IIC3413 - Implementación de Sistemas de Bases de Datos

Andrés Cabezas

2 de abril de 2025

Índice general

Capítulo 1

Introducción a los Sistemas de Bases de Datos

1.1. Motivación

Primero que nada, sería bueno contestar la mítica pregunta que harían algunas personas: *¿Y por qué no simplemente almacenas todo en un Excel / documento de texto / papel?* A veces puede no ser inmediatamente obvio el propósito de un sistema de bases de datos.

Figura 1.1: Almacenamiento de datos análogo

Muchas veces los métodos pueden ser demasiado desorganizados, lentos para realizar búsqueda, ineficientes con el uso del espacio y en los peores casos, poco confiables. Por esto, resulta importante usar bases de datos, que se deben enfocar en resolver estos problemas.

Sin embargo, considerando todo esto, aún faltan cosas para implementar un *buen* sistema de bases de datos. Perfectamente un sistema de bases de datos que resuelve todas las cosas que hemos descrito anteriormente, que por ejemplo, se puede basar en almacenar los datos en un `.txt` va a presentar los siguientes problemas

- Organización ineficiente de los datos.
- Búsqueda costosa de los datos por falta de uso de índices.

- Fuerza bruta para procesar las consultas.
- No hay manejo de buffer: Todo se va a la RAM, colapsando el sistema muy rápido.
- Falta de control de concurrencia: Si más de un actor usa la base de datos simultáneamente, **caos**.
- No es capaz de recuperarse de una falla: Se perderán datos si, por ejemplo, se corta la luz.
- Falta de seguridad.
- No paralelizable.

Algunos buenos motivos para tratar de comprender muy bien el funcionamiento de los sistemas de bases de datos (que nos permitirá crear un buen sistema de bases de datos) son los siguientes:

- Big Data y Data Deluge: Hoy en día estamos produciendo datos muy rápido... Al punto en que estamos teniendo problemas para poder procesarlos e interpretarlos de manera correcta. No sirve de nada tener un montón de datos si no podemos hacer nada con ellos.
- Manejar datos es muy importante: ¿No me crees? Pregúntale a las plataformas grandes de internet, como redes sociales masivas como X (twitter jeje), Instagram, Facebook, etc. Manejar volúmenes gigantes de datos es algo muy importante para ellos (si no, sus plataformas simplemente no funcionarían). Practicamente cualquier negocio hoy en día se basa en datos. Si hace mucho tiempo atrás existió la fiebre del oro, hoy en día estaríamos en algo así como la fiebre de los datos.
- La herramienta más usada para trabajar con datos: Saber trabajar con sistemas de bases de datos es un skill que se usa mucho hoy en día en el mundo laboral. Es extremadamente solicitado.

Mi abuelo siempre me decía...

"Las técnicas para el manejo y gestión eficiente de datos representan hoy en día una herramienta fundamental que puede ayudarles a ustedes, los futuros ingenieros o científicos de datos, para utilizar el poder inherente en los datos y transformarlos en conocimiento."

1.2. El Modelo Relacional

1.2.1. Anatomía de una Relación

Una relación R se compone de

- Esquema: $\texttt{name(att}_1\texttt{: } D_1\texttt{, ..., att}_n\texttt{: } D_n\texttt{)}$, donde
  - `name`: Nombre de la relación. Por ejemplo, el nombre de la `TABLE` en PostgreSQL.
  - $\texttt{att}_i$: Nombre del atributo i.
  - D_i: Dominio del atributo i. Que datos son los que almacenamos en ese atributo (números, strings, fechas, vectores, etc).
  - n: Cantidad de atributos de la relación R.
- Instancia: Subconjunto finito de $D_1 \times \ldots \times D_n$. Esto es al final la tupla en cuestión con los datos.

Observación

Dada una relación R, usaremos las siguientes expresiones para referirnos a ciertos aspectos.

- R: El nombre y la instancia de la relación R.
- schema(R): El esquema de la relación R.
- arity(R): La cantidad de atributos de la relación R.
- dom($\texttt{att}_i$): El dominio D_i del atributo $\texttt{att}_i$.

Una relación R se puede representar por medio de una tabla.

$\texttt{att}_1$	$\texttt{att}_2$	...	$\texttt{att}_n$
v_1	v_2	...	v_n
v_1'	v_2'	...	v_n'
...	...	...	...

Figura 1.2: Tabla representante de una relación R.

En este ejemplo, $\vec{v}$; $\vec{v'}$ corresponden a tuplas de R. Cabe destacar que aquí el orden de las filas no es importante y cada tupla es diferente al resto. Claramente se pueden repetir datos entre las tuplas, pero siempre tiene que haber por lo menos un atributo que sea diferente.

1.2.2. Base de Datos Relacional

Una base de datos relacional $\mathcal{D}$ consiste en un conjunto de relaciones, cada una con un nombre distinto.

$$\mathcal{D} = \{R_1, R_2, \ldots, R_m\}$$

Un esquema relacional $\mathcal{S}$ de una base de datos relacional $\mathcal{D}$ consiste en el conjunto de esquemas de todas las relaciones que componen a $\mathcal{D}$.

$$\mathcal{S} = \text{schema}(\mathcal{D}) = \{\text{schema}(R_1), \text{schema}(R_2), \ldots, \text{schema}(R_m)\}$$

Otras definiciones

- $dom(\mathcal{S})$: Todas las bases de datos que tienen como esquema relacional a $\mathcal{S}$.
- $|R|$: Número de tuplas en R (cardinalidad).

1.2.3. Restricciones de Integridad y Validez de una Base de Datos

Una restricción de integridad consiste en una condición φ que restringe los datos que pueden ser almacenados en una base de datos relacional.

De esto, podemos derivar el concepto de validez de una base de datos. Se dice que una base de datos es válida respecto a φ si es que satisface la restricción impuesta por φ.

$$\mathcal{D} = \varphi$$

1.2.4. Extracción de datos a base de consultas

Primero que nada, debemos dejar en claro *qué es* una consulta. Para nosotros, una consulta consiste en una función de tipo

$$f: \ dom(\mathcal{S}) \to D$$

donde S es un esquema relacional y D es un dominio cualquiera.

De aquí podemos extrapolar el concepto de lenguaje de consultas. Esto es un conjunto de expresiones sintácticas que definen una consulta por medio de una semántica. Aquí nos enfocaremos en SQL y Algebra Relacional.

1.3. SQL (Structured Query Language)

Es actualmente el estándar mundial para consultas a bases de datos relacionales. Esto se usa en todos lados (como mi ejemplo favorito, PostgreSQL). Este es un lenguaje de consultas declarativo, basado en cálculo relacional. Este lenguaje tiene varios componentes.

- Data Definition Language (DDL): `CREATE`, `ALTER`, etc.
- Data Manipulation Language (DML): `SELECT`, `INSERT`, `DELETE`, etc.
- Data Control Language (DCL): `GRANT`, `REVOKE`, etc.

Una consulta SQL sigue una estructura general que se ve más o menos así:
<MINTED>

De aquí aparecen las consultas anidadas. Estas son consultas que contienen dentro de sí, una consulta embebida. Usualmente esto se da en expresiones como `WHERE`, `FROM` o `HAVING`

Ejemplo

<MINTED>
Resultado: El nombre y la cantidad de goles del jugador con más goles a su nombre.

Existe un tipo particular de consulta anidada, llamada consulta correlacionada, la cual contiene una subconsulta por medio de una referencia a una consulta externa.

Ejemplo

<MINTED>
Resultado: El nombre y el año de los jugadores que meten en promedio más de un gol en un partido.

1.4. Álgebra Relacional

Este consiste en un lenguaje para realizar consultas en bases de datos relacionales basado en operadores relacionales.

1.4.1. Operadores Relacionales

Selección $\sigma_\theta(R)$

Operador que filtra las tuplas que se obtendrán de la consulta. Es un equivalente al `WHERE` en SQL.
Aquí, θ es una combinación booleana de términos:

$$\text{atributo}_1 \text{ op } \text{atributo}_2$$
$$\text{atributo op constante}$$

donde op $\in \{=, \leq, \geq, <, >\}$.

Proyección $\pi_L(R)$

Operador que elige los atributos que se obtendrán al final de la consulta. Es un equivalente al `SELECT` en SQL.

En este operador, L corresponde a una lista de atributos presentes en la relación R.

Joins

Este tipo de operador se encarga de mezclar dos relaciones.

- Producto cruz: $R_1 \times R_2$
- θ-join: $R_1 \bowtie_\theta R_2 = \sigma_\theta(R_1 \times R_2)$
- Equi-join: $R_1 \bowtie_\phi R_2 = \sigma_\phi(R_1 \times R_2)$, donde ϕ solo contiene igualdades.
- Natural-join: $R_1 \bowtie_\phi R_2 = \sigma_\phi(R_1 \times R_2)$, donde $\phi = \bigwedge_{a \in \texttt{att}(R_1) \cap \texttt{att}(R_2)} R_1.a = R_2.a$

1.4.2. Álgebra Relacional Extendida

Se agregan algunos operadores extra que no son parte del álgebra relacional tradicional con el fin de permitir hacer algunos procesos que son necesarios hoy en día.

- Renombrar: $\rho_{\texttt{old\_att} \to \texttt{new\_att}}(R)$
- Eliminar duplicados: $\delta(R)$
- Group-by con agregación: $\gamma_{G,A}(R)$, donde G es la lista de atributos a agrupar, y A es una lista de elementos de la forma $f(\texttt{agg\_att} \to \texttt{new\_att})$ con $f \in \{\texttt{MIN}, \texttt{MAX}, \texttt{SUM}, \texttt{AVG}, \ldots\}$
- Sorting: $\tau(R)$

1.4.3. Plan lógico

El plan lógico consiste en un árbol de parsing, el cual da el primer paso para poder evaluar una consulta.

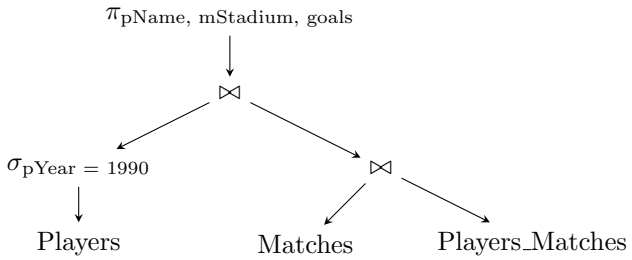


Figura 1.3: Ejemplo de un plan lógico

Capítulo 2

Arquitectura de un Sistema de Bases de Datos Relacional

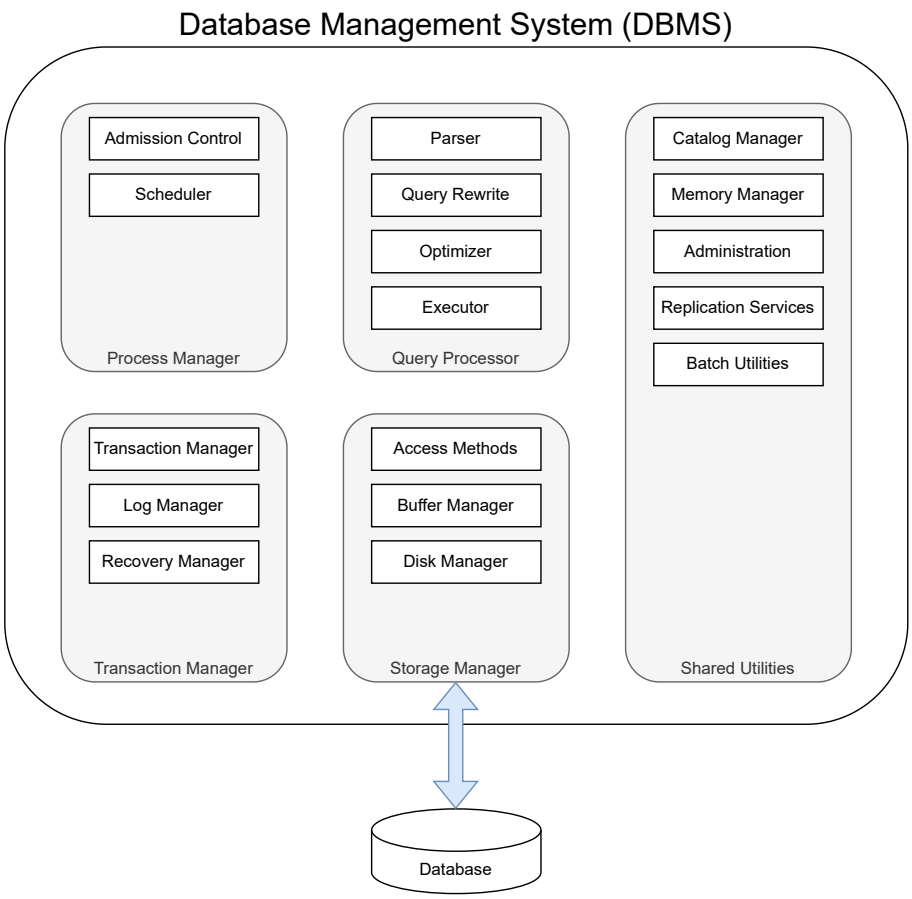


Figura 2.1: Arquitectura de un DBMS convencional

2.1. Query Processor

Para procesar una consulta, es necesario realizar una serie de pasos.

1. Parser:
   - Valida la consulta (correctitud, autorización, etc).
   - Convierte la consulta en un formato interno (algebra relacional).
   - Optimizaciones menores.
2. Reescritura de la consulta (Query Rewrite):
   - "Desanidación"de la consulta (flattening).
   - Reescritura de la consulta aplicando reglas de algebra relacional.
   - Creación de un set de planes lógicos optimizados
3. Optimizador:
   - Encuentra el plan físico más eficiente para ejecutar la consulta.
   - Se consideran varios factores, como el tamaño de cada relación, la distribución de sus datos, distintos métodos de acceso a las relaciones, distintos planes lógicos posibles para la misma consulta, etc.
4. Ejecutor:
   - Ejecutar el plan óptimo generado por el optimizador (pipelining).

2.1.1. Plan físico

El plan físico es similar al plan lógico del álgebra relacional, pero este decide que algoritmo usar para cada operador, planifica la secuencia y el orden a ejecutar de los operadores, además de la forma de acceder a los datos (access path).

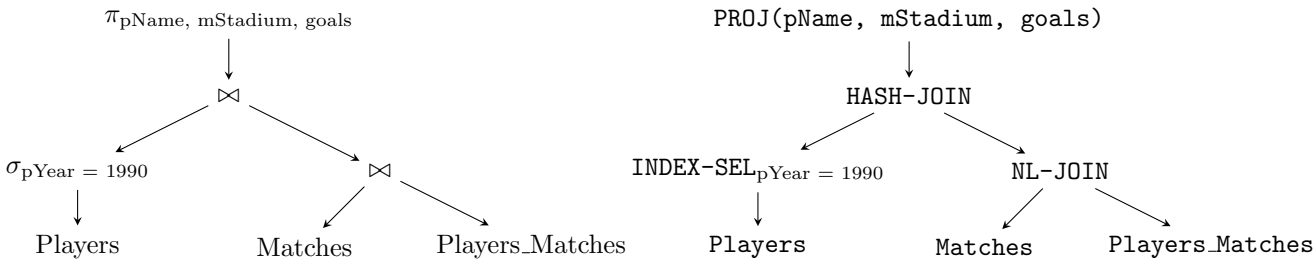


Figura 2.2: Plan lógico a plan físico

2.1.2. Ejecución

Se empieza a ejecutar en serie el plan físico. Aquí, cada operador retorna tuplas a su padre en el árbol, todo ASAP.

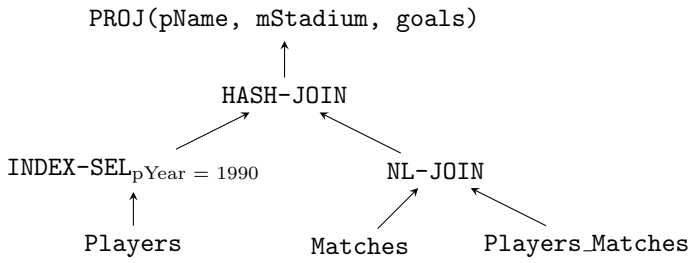


Figura 2.3: Pipelining

2.2. Storage Manager

2.2.1. Disk Manager

Esto se encarga del acceso al almacenamiento secundario (Disco duro), y de organizar las tuplas dentro del disco, por medio de heapfiles.

2.2.2. Buffer Manager

Esto se encarga del manejo de los datos en memoria. También se encarga de optimizar la cantidad de accesos que ocurren en el disco duro, ya que el acceso al disco duro es más lento que la RAM. Idealmente se deseará hacer todo en RAM y pasar al almacenamiento persistente al final. Este componente también es importante para todo lo relacionado con transacciones y ACID.

2.2.3. Access Methods

Esto maneja todos los indices y se encarga de mantener una organización eficiente de los datos, tanto en términos de almacenamiento como ordenado para hacer que su acceso sea más rápido.

2.3. Transaction Manager

Este macrocomponente trata de mantener las propiedades ACID (Atomicity, Consistency, Isolation, Durability) en los datos. Particularmente, el Transaction Manager asegura Isolation y Consistency, mientras que el Log Manager y el Recovery Manager aseguran Atomicity y Durability.

2.4. Process Manager

2.4.1. Admission Control

Este componente se encarga de manejar usuarios, permisos, autentificación y otras cosas relacionadas. También es el encargado de asegurar la cantidad de recursos necesarios para que el sistema funcione correctamente.

2.4.2. Scheduler

Este componente se encarga del manejo de procesos y threads dentro de la base de datos, además de paralelizar las consultas.

2.5. Shared Utilities

- Catalog Manager: Mantiene la estructura y metadata de los datos. Le permite a la base de datos saber que datos se están almacenando en las distintas relaciones para saber como trabajar con ellos.
- Memory Manager: Asigna memoria para distintos componentes de la base de datos.

Capítulo 3

Almacenamiento

3.1. Jerarquía de Memoria

Existen distintos niveles de almacenamiento, todos con sus pros y contras.

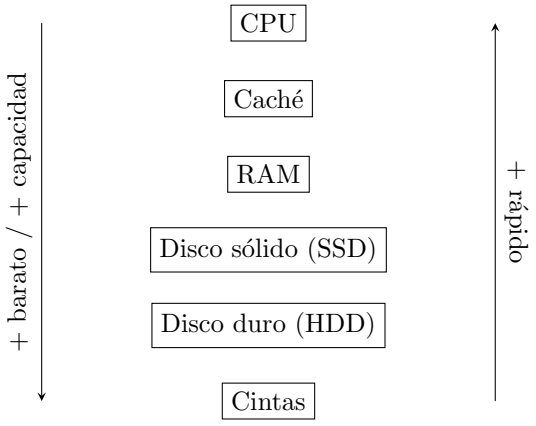


Figura 3.1: Jerarquía de Memoria

Cabe destacar que en la jerarquía tenemos memoria volátil y no volátil. La memoria volátil es la memoria que una vez se desconecta su energía o se apaga, pierde todos sus datos, mientras que la no volátil, una vez desconectada su energía es capaz de mantener sus datos.

- Volátil:
  - CPU
  - Caché
  - RAM (Memoria principal)
- No volátil:
  - Disco sólido (SSD)
  - Disco duro (HDD)
  - Cintas

Para las bases de datos, claramente la parte más importante de esta jerarquía sería el almacenamiento persistente que se usa hoy en día: Los discos duros magnéticos y los discos sólidos.

3.2. El disco duro

El interior de un disco duro se ve así

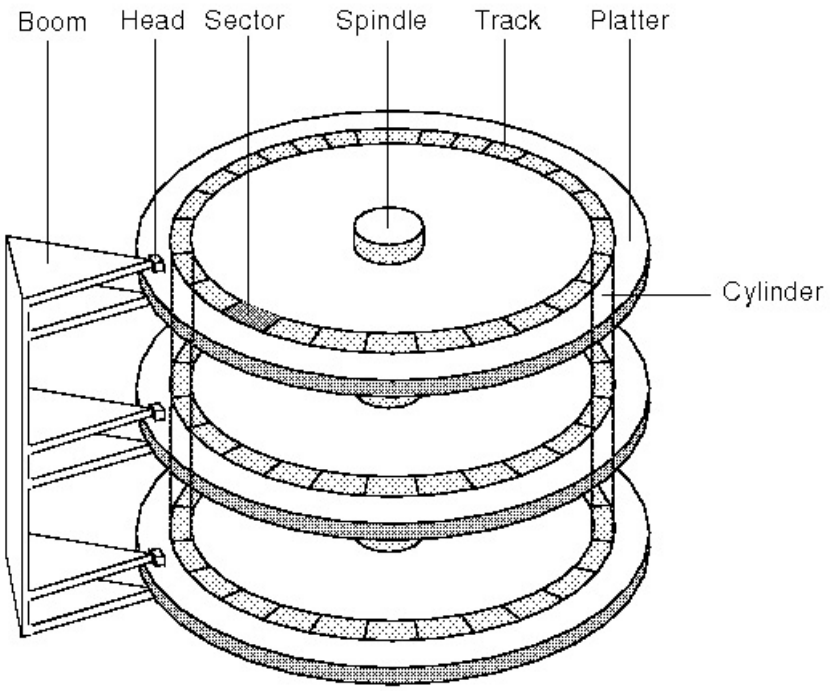


Figura 3.2: Interior de un disco duro

3.2.1. Sectores, bloques y páginas

Cuando trabajamos con un disco duro, se usan estos términos para hablar de la ubicación de datos dentro de este.

- Sector: Un sector es la unidad física mínima de almacenamiento.
- Bloque: Un bloque es la unidad lógica mínima de almacenamiento en disco.
- Página: Una página es la unidad lógica mínima de almacenamiento en el sistema.

3.2.2. Demora en búsqueda de datos (y escritura)

Si se desea obtener un cierto dato desde un disco duro mecánico, es esperable que tome un poco de tiempo. Esto debido a los distintos pasos que se deben realizar para llegar al dato deseado. El tiempo total esta dado por

$$\text{Tiempo total} = \text{Tiempo de búsqueda} + \text{Retraso rotacional} + \text{Velocidad de transferencia}$$

- Tiempo de búsqueda: Tiempo que se demora el disco en posicionar la cabeza en el track correspondiente (3ms - 15ms promedio).
- Retraso rotacional: Tiempo que se demora en rotar el disco al primer sector (2ms - 7ms promedio).

- Velocidad de transferencia: Tiempo que toma leer los sectores, incluyendo espacios entremedio (depende del tamaño de los datos).

A la hora de escribir en el bloque, la demora es similar (o un poco mayor). Para modificar un bloque se debe primero leer el bloque, luego hacer las modificaciones a dicho bloque en memoria, para finalmente escribir el bloque usando el contenido presente en memoria.

3.2.3. Optimizaciones a la hora de usar el disco

Considerando que hacer cosas en el disco es bastante lento, lo ideal seria optimizar lo más posible el proceso. De esta forma, nuestro sistema tardará menos en responder consultas o escribir información nueva.

Localidad de los datos

Si las tuplas t y t' son parte de la misma relación R, entonces se debe almacenar t:

- En el mismo bloque que t'. Si no...
- En el mismo track que t'. Si no...
- En el mismo cilindro que t'. Si no...
- En el cilindro contiguo a t'.

Pre-fetch

La idea es adivinar y traer bloques que se predice que serán solicitados en el futuro. Para esto, si por ejemplo, se solicita el bloque k, traer los bloques contiguos a k, como $k+1, k+2, k+3, \ldots, k+N$, donde N es definido por el usuario (cuanto mayor N, mayor será el costo de la optimización. Usar con cautela).

Planificación del disco

Dada una secuencia de solicitudes $o_1, \ldots, o_n$ al disco, reordenar las solicitudes a $o_{i_1}, \ldots, o_{i_n}$ de tal forma que se minimice el tiempo total.
Esto por ejemplo, se puede hacer reordenando las solicitudes de forma tal que se escriban todas las cosas en lugares cercanos del disco, para moverse una sola vez a otro sitio, en lugar de ir y volver varias veces (mover el cabezal consume mucho tiempo)

Múltiples discos

Distribuir tuplas en múltiples discos, para así tener acceso concurrente a las relaciones y tuplas.
En otras palabras, la idea es que podamos usar varios discos a la vez para leer la información más rápido. Esto es posible solamente si los datos se encuentran distribuidos en múltiples discos. En un solo disco debemos esperar a que el cabezal lea un dato y después avance al siguiente. Si tenemos, por ejemplo, 8 discos, es posible leer 8 tuplas en el tiempo que nos toma leer una desde un solo disco.

Múltiples copias (mirroring)

Principio muy similar al anterior. Consiste en mantener múltiples copias de los datos en varios discos. Esto genera redundancia, por lo que ante fallas se logra mantener disponibilidad de los datos (o incluso hacer que una falla no sea catastrófica). Esto también aumenta la velocidad de lectura de los datos, aunque genera mayor lentitud a la hora de escribir (ya que todos los discos que tengan copias tendrán que sincronizar los datos. Escribir un dato implicará escribir en varios lugares.).

3.3. Organizando los datos con páginas

Ahora que comprendemos como usar el disco correctamente, toca responder la pregunta siguiente: *¿Cómo almacenamos las distintas tuplas de una relación en el disco?*. ¡Aquí es donde las páginas juegan un rol fundamental!

Una página (sistema) corresponde a un bloque en el disco. Cada relación corresponde a un set de páginas que contienen un subconjunto de tuplas. Esto entrega un manejo granular del contenido del disco, optimiza el acceso al disco y facilita el manejo de las transacciones.

3.3.1. Heapfiles

Un heapfile es una estructura de datos diseñada para almacenar relaciones en páginas.

Para almacenar una tupla en un heapfile, podemos escribir primero un directorio, el cual nos indica a que offset del disco dirigirnos para obtener un cierto atributo de la tupla. Eso se ve más o menos así.

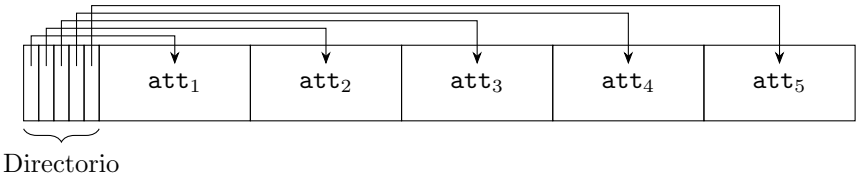


Figura 3.3: Ejemplo de una tupla en un heapfile.

Luego, como deseamos almacenar varias tuplas, lo ideal sería darles un identificador. A esto se le llama *"Record ID"*. En general, una buena elección de RID es (PageID, NumSlot). Ahora, para almacenar varias tuplas, hay dos approachs: Almacenar tuplas de largo fijo y almacenar tuplas de largo variable.

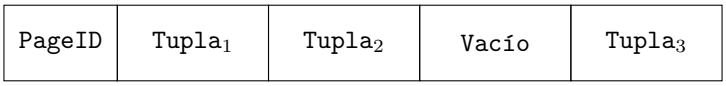


Figura 3.4: Ejemplo de una página con tuplas de largo fijo.

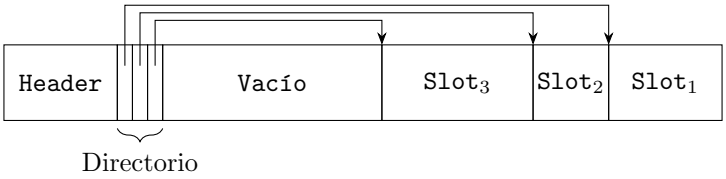


Figura 3.5: Ejemplo de una página con tuplas de largo variable.

La última pregunta que queda por responder es como almacenar una misma relación en varias páginas. Para mantener el orden, una forma de hacerlo es por medio de overflow pages. Esto puede conducir a una base de datos lenta e ineficiente, por lo que se recomienda usar con discreción.

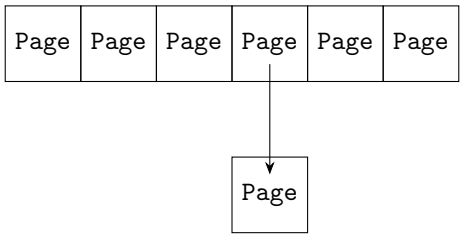


Figura 3.6: Ejemplo de overflow page.

3.4. Buffer Manager

El buffer manager es el mediador entre el disco y la memoria principal. Cuenta con una cantidad restringida de la memoria RAM del sistema en donde se ejecuta la base de datos. Este componente se encarga de llevar las páginas del disco a la memoria bajo demanda, y es también responsable de decir que páginas deben ser eliminadas de la memoria cuando se encuentra lleno.

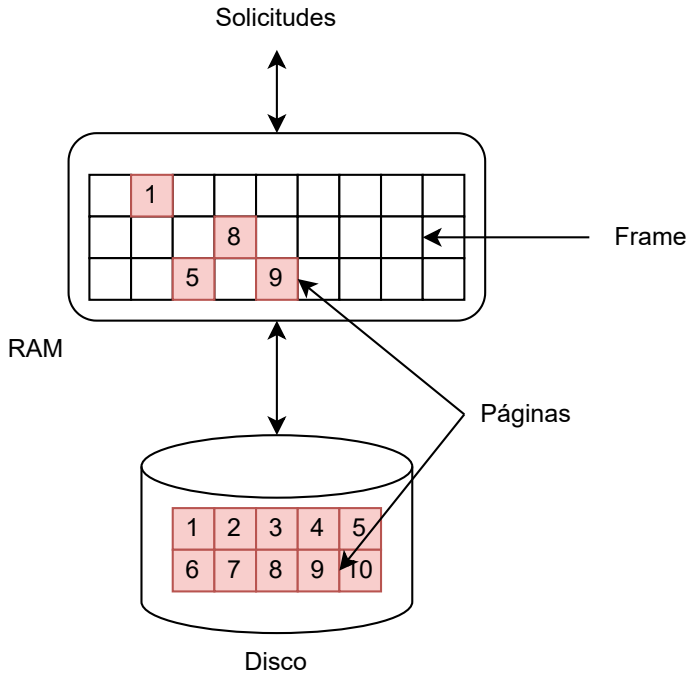


Figura 3.7: Buffer Manager

Aqui, cada frame tiene dos variables:

- `#pin`: Cantidad de procesos que están usando actualmente la página.

- `dirty`: Si el contenido de la memoria ha cambiado con respecto al contenido en el disco.

Luego, se deben exponer dos funciones para que los procesos de la base de datos puedan usar el buffer manager.

- `pin(pageno)`
  1. Solicitar la página `pageno` al buffer manager.
     - Si la página no se encuentra en memoria, se debe seleccionar un frame vacío, traer la página a la memoria y cargar en el frame vacio. Finalmente, al frame se le asigna `#pin = 1` y `dirty = false`.
     - Si la página ya está en memoria, simplemente `#pin += 1`.
  2. Retornar referencia al frame que contiene a `pageno`
- `unpin(pageno, dirty)`
  1. Solicitar la liberación de la página `pageno` al buffer manager.
  2. `#pin -= 1`
  3. Actualizar `dirty = true` si la página ha sido modificada.

Para que esto funcione correctamente, es necesario que cada proceso haga un uso correcto de las funciones de la interfaz. Esto es, que todo proceso haga `pin`, trabaje con los datos, y finalmente haga `unpin`. También es importante que cada proceso que hace `pin` a una página, haga `unpin` lo antes posible.

3.5. Índices

Un índice es un método de acceso que optimiza el acceso a los datos para una consulta o conjunto de consultas en particular.

En un sistema de bases de datos, un índice tiene el objetivo de optimizar algunas consultas. Particularmente, es posible optimizar búsquedas de valores (*value query*), búsquedas por rango (*range query*) y búsquedas por match (*pattern matching*).

3.5.1. Composición de un índice

Un índice está compuesto por los siguientes elementos

- **Search Key**: Parámetros de búsqueda.
- **Index Entry**: Valor o puntero guardado dentro del índice
- **Data Entry**: Record o dirección en donde se almacena el record al que apunta el índice. Esto puede ser, por ejemplo (dada una search key k):
  - Un record (que satisface a la search key k).
  - Una tupla (k, RID) (Ej.: MilleniumDB de IIC3413).
  - Una tupla $(k, \text{lista de RID})$.

3.5.2. Clasificación de índices

Clustered vs. Unclustered Indexes

- **Clustered Index / Índice primario**: Índice en donde el orden de las entradas es igual al orden en el que se encuentran los records en el disco.
- **Unclustered Index / Índice secundario**: Índice en el que el orden de las entradas no es necesariamente el mismo orden de los datos en el disco. Si la salida a obtener es numerosa, este se vuelve ineficiente.

Denso vs. Disperso

- **Dense / Denso**: Existe un index entry por cada record en la relación.
- **Disperse / Disperso**: No todo record posee una entrada en el índice.

3.6. Índices basados en árboles

Estos índices usan una estructura de árbol que les permite mantener los datos ordenados de manera sencilla usando directorios que apuntan finalmente a los elementos del índice.

Existen dos tipos fundamentales de índices con estructura de árbol: ISAM y B+-Trees

3.6.1. ISAM - Indexed Sequential Access Method

ISAM corresponde a un tipo de índice con estructura de directorio estático. Cada página del directorio toma una estructura en donde hay punteros y valores intercalados, en la siguiente forma.

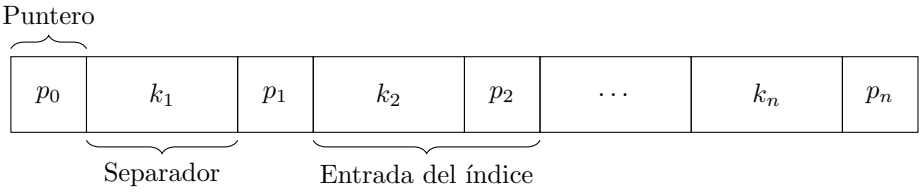
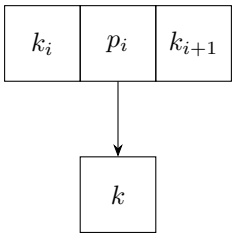


Figura 3.8: Ejemplo de página de un índice ISAM

Aquí siempre se tiene que cumplir que cuando un puntero p_i apunta a un valor k.



se debe dar que $k_i \leq k \leq k_{i+1}$.

Búsqueda en ISAM

Hacer búsqueda en ISAM es muy similar a las búsquedas que se hacen en árboles.

Range Queries en ISAM

Para hacer una range query con valores entre `x` e `y`, considerando que tenemos el algoritmo escrito anteriormente

1. Llamar $P =$ `busquedaEnArbol(x, root)`.
2. Realizar una busqueda binaria del mayor elemento k^* en P tal que $k^* \leq$ `x`.
3. Hacer scan desde k^* sobre todos los valores menores o iguales a `y`

Algorithm 1: busquedaEnArbol(k, P)

Input: Una search key k y una página P del índice
Output: Una página de datos que puede contener a k
if P *es una página de datos* **then**
  return P
$P = p_0 k_1 p_1 \dots k_n p_n$;
switch k **do**
  case $k < k_1$ **do return** busquedaEnArbol(k, p_0);
  case $k_i \leq k < k_{i+1}$ **do return** busquedaEnArbol(k, p_i);
  case $k_n < k$ **do return** busquedaEnArbol(k, p_n);

Inserción en ISAM

Para insertar un valor k en la página de índice P

1. Llamar $P =$ `busquedaEnArbol(`k`, root)`.
2. Si P tiene espacio para k, insertar k en P.
3. Si P no tiene espacio para k, insertar k en una pagina de overflow.

Eliminación en ISAM

Para eliminar un valor k

1. Llamar $P =$ `busquedaEnArbol(`k`, root)`.
2. Si P contiene a k, eliminar k en todas las páginas de overflow que contengan a k (duplicados).

Costos de ISAM

Considerando

- H: Cantidad de niveles del arbol ISAM.
- V: Largo máximo de las cadenas de páginas de overflow.

Los costos para las operaciones vistas quedan en

- Búsqueda: $\mathcal{O}(H + V)$
- Inserción: $\mathcal{O}(H + V)$
- Eliminación: $\mathcal{O}(H + V)$

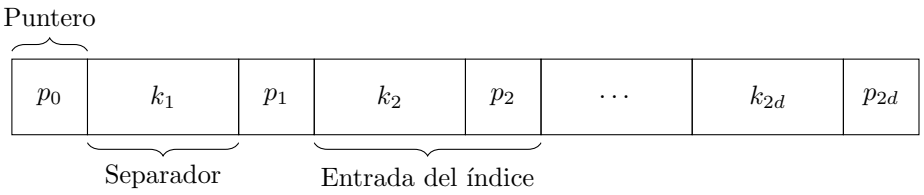
Pros y contras de ISAM

- **Pros**
  - Rendimiento pobre si se realizan muchas modificaciones a los datos guardados.
  - Deficiente si la cantidad de datos aumenta.
- **Contras**
  - Eficiente en búsqueda dentro de bases de datos con pocas modificaciones a sus datos.
  - Sencillo de implementar.
  - Útil para acceso concurrente.

3.6.2. B+-Trees

Este es un tipo de índice muy similar a ISAM, pero a diferencia de este, B+-Trees no usa páginas de overflow y el árbol siempre está balanceado, logra mantener la eficiencia de la búsqueda ($\mathcal{O}(\#\text{tuplas})$) y usa procedimientos eficientes para insertar y eliminar elementos.

La forma que tienen los nodos es casi idéntica a ISAM



La diferencia es que la cantidad máxima de llaves y punteros (n) está definida por el orden del árbol (d) B+-Tree

$$d \leq n \leq 2d$$
$$1 \leq n \leq 2d \quad \text{para root}$$

Este orden d es siempre el orden del tamaño de una página.

$$d \approx \frac{B}{2}$$

Para buscar en un B+-Tree, el algoritmo de búsqueda es exactamente el mismo que en ISAM.

Range Queries en B+-Trees

Para hacer una range query entre `x` e `y`, considerando que tenemos `busquedaEnArbol(`k`, `P`)`.

1. Llamar $P =$ `busquedaEnArbol(x, root)`.
2. Realizar una busqueda binaria del mayor elemento k^* en P tal que $k^* \leq$ `x`.
3. Hacer scan desde k^* sobre todos los valores menores o iguales a `y`

Supuesto

Se asume que todos los data keys son diferentes (sin duplicados)

Inserción en B+-Trees

Para insertar un valor k

1. Llamar $P =$ `busquedaEnArbol(`k`, root)`.
2. Si el espacio libre de P es menor o igual a $2d$, insertar k en P.
3. Si no es así, se debe insertar k en P, y luego hacer split de $[P + k]$, donde $[P + k] = k_1^* k_2^* \dots k_{2d}^* k_{2d+1}^*$ (es una página de datos).

Para hacer split de una página de datos $[P + k]$ de tamaño $2d + 1$

1. Asuma $[P + k] = k_1^* k_2^* \dots k_{2d}^* k_{2d+1}^*$ con $k_i < k_{i+1}$.

2. Divida $[P + k]$ en dos páginas

$$P_1 = k_1^* \dots k_d^* \quad P_2 = k_{d+1}^* \dots k_{2d+1}^*$$

y reemplace P por P_1, P_2 en la lista doble-ligada de datos.

3. Seleccione el valor k_{d+1} como divisor de P_1 y P_2.
4. Reemplace el puntero p en la página P' que apuntaba a la página P por $p_1 k_{d+1} p_2$ donde p_1 apunta a P_1 y p_2 apunta a P_2.
5. Itere sobre la página de directorio P' que apuntaba a P (split de P').

Para hacer split de una página de directorio P de tamaño $2d + 1$

1. Asuma $P = k_1 k_2 \dots k_{2d} k_{2d+1}$ con $k_i < k_{i+1}$.
2. Divida P en dos páginas

$$P_1 = k_1 \dots k_d \quad P_2 = k_{d+1} \dots k_{2d+1}$$

y reemplace P por P_1, P_2 en el directorio.

3. Seleccione el valor k_{d+1} como divisor de P_1 y P_2.
4. Reemplace el puntero p en la página P' que apuntaba a la página P por $p_1 k_{d+1} p_2$ donde p_1 apunta a P_1 y p_2 apunta a P_2.
5. Itere sobre la página de directorio P' que apuntaba a P (split de P').

Para hacer split del nodo raíz de un B+-Tree, se comienza haciendo split en las hojas y continúa hasta llegar a la raíz. Si es necesario hacer split de todos los nodos y el nodo raíz está lleno, entonces será necesario crear un nuevo nodo raíz. Una consecuencia de esto, es que la profundidad del árbol aumentará.

Eliminación en B+-Trees

Para eliminar un valor k

1. Llamar $P =$ `busquedaEnArbol(`k`, root)`.
2. Si el espacio usado en P es mayor o igual a $d + 1$, eliminar k en P.
3. Si no, se debe eliminar k en P y rebalancear $[P + k]$

Costos de B+-Trees

Considerando

- T: Número de túplas.
- B: Tamaño de una página.

Los costos para las operaciones vistas quedan en

- Búsqueda: $\mathcal{O}(\log_{\frac{B}{2}}(\frac{2 \cdot T}{B}))$
- Inserción: $\mathcal{O}(\log_{\frac{B}{2}}(\frac{2 \cdot T}{B}))$
- Eliminación: $\mathcal{O}(\log_{\frac{B}{2}}(\frac{2 \cdot T}{B}))$

Lidiando con elementos duplicados en B+-Trees

Antes se habia hecho la suposición de que no estabamos trabajando con elementos duplicados jamás. Sin embargo, esto es algo que claramente puede ocurrir. Si consideramos elementos duplicados, se tienen varias opciones.

- Usar páginas de overflow.
- Entradas con llaves compuestas.
- Permitir duplicados y flexibilizar los intervalos del directorio.

$$k_i \leq k < k_{i+1} \Rightarrow k_i \leq k \leq k_{i+1}$$

3.7. Índices basados en Hashing

Los índices basados en hashing almacenan la información del índice por medio de una tabla de hash y una función de hashing. Estos índices son muy eficientes para hacer consultas de valores, aunque no son tan eficientes para range queries como lo es B+-Trees. A día de hoy, la mayoría de motores de bases de datos incluyen soporte para hash indexes.

3.7.1. Static Hashing

En Static Hashing, el índice es almacenado usando una tabla de hash, en donde se tienen N buckets (páginas en disco). Cada bucket tiene una lista ligada de overflow pages. Para relacionar las entradas del índice con su valor correspondiente, se usa una función de hash $h : \texttt{datatype}(X) \rightarrow [0, \ldots, N-1]$.

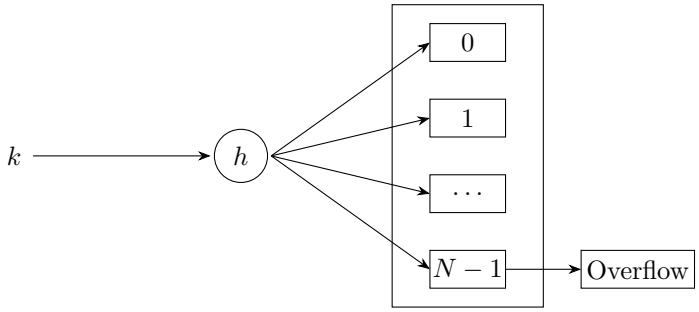


Figura 3.9: Funcionamiento de la función de hashing

Operaciones en static hashing

Para buscar, insertar y eliminar una llave k

1. Computar el valor $h(k)$.
2. Acceder al bucker en la posición $h(k)$.
3. Buscar / Insertar / Eliminar la tupla en la página $h(k)$ o sus overflow pages.

Colisiones

Puede llegar a pasar que para la función de hashing $h(k)$, existan dos valores $k_1 \neq k_2$, pero que $h(k_1) = h(k_2)$. Idealmente se debe encontrar una función de hashing en donde las colisiones sean disminuidas al mínimo. Idealmente, la función de hash se debería comportar de forma aleatoria para $k_1, \ldots, k_n$.

Costo de operaciones

Dada

- Una buena selección de la función de hash h
- `#(data-entries)` $\propto B \cdot N$, con B tamaño de página y N buckets.

se tiene que el costo de todas las operaciones es constante.

$$\approx \frac{\texttt{\#(data-entries)}}{B \cdot N}$$

3.7.2. Extendable Hashing

La idea es usar un directorio de buckets principales, y luego duplicar el tamaño del directorio cuando haya un overflow de un bucket.

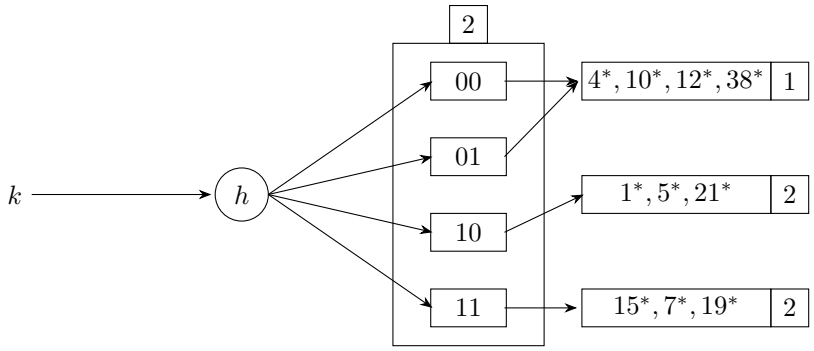


Figura 3.10: Extendable hashing

Inserción en Extendable Hashing

1. Computamos $h_n(k)$.
2. Buscamos en el directorio el puntero p para la entrada $h_n(k)$.
3. Seguimos el puntero p al buscket B correspondiente.
4. Si hay espacio en B, insertamos k en B.
5. Si no hay espacio en B con profundidad local d, se debe realizar un bucket split de B. Así, se aumenta la profundidad local de B_0 y B_1 a $d+1$.
   - Si $n > d$, insertamos B_0 y B_1 en el directorio.
   - Si $n = d$, hacemos un doblamiento del directorio.

Bucket Split

Dado un bucket B con profundidad local d, se dividen sus elementos en dos buckets según el bit $d+1$.

Doblamiento del Directorio

Dado un bucket B con profundidad local $d = n$, hacemos un bucket split de B y duplicamos el tamaño del directorio a 2^{n+1}.

Eliminación en Extendable Hashing

Para eliminar un elemento $k*$, este se debe buscar y eliminar del bucket al que pertenezca. Si el bucket queda vacío, extendable hashing hace merge de buckets, aunque esto es poco usado.

Costo de Extendable Hashing

Considerando D el tamaño del directorio en páginas, el costo de cada operación es

- Búsqueda: $\mathcal{O}(1)$
- Inserción: $\mathcal{O}(D)$

3.8. Índices Multidimensionales - Bitmap Index

En los índices multidimensionales, existe más de una search key en una misma tupla, con todas esas search key con el mismo grado de importancia.

Bitmap index es un índice multidimensional bastante diferente al resto de propuestas que existen. Permite realizar consultas multidimensionales y está basado en una codificación alternativa de las tuplas de una relación.

Para una relación R

1. Enumeramos sus tuplas del 1 a $|R|$ de forma permanente, donde $R = \{t_1, t_2, \ldots, t_{|R|}\}$.
2. Para un atributo c de R y un valor $v \in \pi_c(R)$ definimos un string binario b_v^c de largo $|R|$ tal que

$$b_v^c[i] = \begin{cases} 1 & \text{si } t_i(c) = v \\ 0 & \text{en otro caso.} \end{cases}$$

3. Un bitmap index sobre un atributo c se define como el conjunto $B^c = \{b_v^c | v \in \pi_c(R)\}$

Bitmap index es muy útil para consultas con filtros complejos, columnas de baja cardinalidad y datos que sufren pocas modificaciones. Por otro lado, es bastante deficiente cuando los datos requieren ser actualizados frecuentemente.

3.9. Tablas de hash

Implementan un mapeo entre llaves y valores utilizando una funcion de hash que permite acceder a los valores de forma rápida. Las tablas de hash son una estructura de datos muy eficiente para almacenar y recuperar información.

- Espacio: $\mathcal{O}(N)$
- Tiempo de búsqueda: $\mathcal{O}(1)$, o $\mathcal{O}(N)$ en el peor de los casos.

3.9.1. Funciones de hash

Permite mapear un espacio grande de llaves a un dominio pequeño. El principal trade-off es que mientras mas rapida sea de computar, mas colisiones se producen.

Entre más memoria se use, menos colisiones se producen por lo tanto la busqueda es mas rapida, sin embargo esto a su vez produce un mayor costo de almacenamiento.

3.9.2. Linear probe hashing

- Reservo un arreglo grande para mis entradas
- Si hay una colision, busco la siguiente posicion vacia en el arreglo.
- Para buscar una entrada, voy hacia donde me lleva el hash y busco hacia adelante.
- Se tiene un load factor que indica que tan lleno esta el arreglo. Si este factor supera un cierto umbral, duplico el tamaño del arreglo y rehago el hash de todas las entradas.

Inserciones

- Calculo el hash de la entrada a insertar.
- Si la posicion esta vacia, la inserto.
- Si la posicion esta ocupada, busco la siguiente posicion vacia (o eliminada) y la inserto.

Eliminaciones

- Primero busco la entrada a eliminar.
- Si la encuentro, marco la posicion como eliminada.

> **Nota**
>
> Al momento de buscar si encontramos una posicion eliminada, debemos seguir buscando hasta encontrar una posicion vacia o la entrada que buscamos. Esto se hace para evitar que las entradas eliminadas interfieran con la busqueda.

3.9.3. Robin Hood hashing

Tiene una logica similar a linear probe hashing, pero tambien cada elemento guarda que tan lejos esta de la posicion donde apuntaba el hash.

Inserciones

- Calculo el hash de la entrada a insertar.
- Si la posicion esta vacia, la inserto.
- Si la posicion esta ocupada, hago una busqueda lineal donde por cada posicion hago las siguientes comparaciones:
  - Si la distancia del numero que estoy insertando es igual o menor del numero que ya estaba, sigo buscando.
  - Si la distancia del numero que estoy insertando es mayor, lo inserto en la posicion, tomo el numero que estaba y lo inserto hacia adelante con la misma logica.

Busqueda

Lo que nos permite este algoritmo es que al momento de buscar, es que si mientras buscamos encontramos una posicion que tiene un valor mayor a la distancia que tenemos, podemos parar la busqueda. Esto ocurre porque si el elemento buscado lo hubiesemos insertado ahi antes.

3.9.4. Filtros Bloom

Un filtro es una estructura de datos que responde a la pregunta de si un elemento pertenece a un conjunto o no.

Los filtros de bloom son probabilisticos, en el sentido de que pueden dar falsos positivos, pero nunca falsos negativos. Esto significa que si el filtro dice que un elemento no pertenece al conjunto, es seguro que no pertenece. Sin embargo, si dice que si pertenece, puede ser que no pertenezca.

Estructura

Es un arreglo de bits de tamaño m y k funciones de hash que mapean un elemento a una posicion en el arreglo. Cada vez que se inserta un elemento, se calculan las k posiciones y se marcan como 1. Para verificar si un elemento pertenece al conjunto, se calculan las k posiciones y si todas son 1, entonces el elemento puede pertenecer al conjunto. Si alguna de las posiciones es 0, entonces el elemento no pertenece al conjunto.

Capítulo 4

Transacciones

Una transacción es una secuencia de una o más operaciones que modifican o consultan una base de datos. Cada transacción se compone de una secuencia de instrucciones y un estado. El estado incluye la posición actual en código y las variables temporales.

Un ejemplo de como se diagrama una transacción es:

T_1	T_2	A	B
`READ(A,x)`		1000	1000
`WRITE(A,x-100)`		900	
	`READ(A,y)`		
	`WRITE(A,y*1.1)`	990	
	`READ(B,y)`		
	`WRITE(B,y*1.1)`		1100
`READ(B,x)`			
`WRITE(B,x+100)`		990	1200

4.1. Propiedades ACID

Consiste en una serie de principios que definen lo que una transacción debería hacer. Esto ayudará un poco a entender el propósito que tienen.

ACID es un acrónimo para *"Atomicity"*, *"Consistency"*, *"Isolation"* y *"Durability"*. Más especificamente...

- **"Atomicity"**: Se ejecutan todos los pasos de la transacción, o no se ejecuta nada.
- **"Consistency"**: Al terminar una transacción, los datos deben estar en un estado consistente.
- **"Isolation"**: Cada transacción se ejecuta sin ser interferida por otras transacciones.
- **"Durability"**: Si una transacción hace commit, sus cambios sobrevivirán a cualquier tipo de falla.

4.1.1. ¿Qué puede salir mal en una transacción?

1. Conflictos Write-Read (WR) - Lecturas sucias: Una transacción hace cambios a un dato, y antes de que la transacción termine, otra transacción está leyendo ese dato modificado y haciendo cosas con él. Esto no se debería permitir, ya que no mantiene el principio de Isolation.

Ejemplo: Conflicto WR

T_1	T_2	A	B
READ(A,x)		1000	1000
WRITE(A,x-100)		900	
	READ(A,y)		
	WRITE(A,y*1.1)	990	
	READ(B,y)		
	WRITE(B,y*1.1)		1100
READ(B,x)			
WRITE(B,x+100)		990	1200

Aquí, T_2 empezó a trabajar con A antes de tiempo, alterando los resultados de los datos.

2. Conflictos Read-Write (RW) - Lecturas irrepetibles: Una transacción hace lectura de un dato para generar cambios en este, pero antes de que comience a hacer cambios, otra transacción hace una lectura y un cambio del mismo dato.

Ejemplo: Conflicto RW

T_1	T_2	A
READ(A,x)		1
IF(x>0)		
	READ(A,y)	
	IF(y>0)	
	WRITE(B,y-1)	0
	ENDIF	
WRITE(B,x-1)		-1
ENDIF		-1

3. Conflictos Write-Write (WW) - Reescritura de datos temporales: Se rompe la secuencia correcta de escritura de datos, generando resultados inesperados.

Ejemplo: Conflicto WW

T_1	T_2	A	B
WRITE(A,1000)		1000	
	WRITE(A,2000)	2000	
	WRITE(B,2000)		2000
WRITE(B,1000)		2000	1000

Posibles soluciones

1. Datos erróneos → Restricciones de integridad, *data cleaning*.
2. Fallas del almacenamiento (disco duro) → RAID, redundancia de datos.

3. Catástrofes → Copias distribuidas.
4. Fallas del sistema → *Log manager*, *Recovery manager*.

4.2. Elementos y Estados de una Base de Datos

Un elemento de una base de datos es un valor o conjunto de valores al que se puede acceder por medio de una transacción. La granularidad queda bajo el criterio del diseñador de la base de datos, y puede ser desde algo tan pequeño como el atributo de una tupla hasta algo tan grande como una relación. La granularidad correcta siempre va a depender del uso que se le dará a la base de datos.

Por otro lado, el estado de una base de datos consiste en el valor de cada uno de sus elementos. De aquí surge el concepto de un estado consistente en la base de datos, que es cuando el estado cumple con las restricciones de integridad y otras restricciones implícitas.

Principio de Correctitud

"Si una transacción comienza con una BD en estado consistente y se ejecuta en ausencia de otras transacciones o errores de sistema, entonces la BD estará en un estado consistente cuando la transacción termine."

4.3. Operaciones de una transacción

Las transacciones interactúan en tres espacios de memoria:

- Almacenamiento no-volátil (disco duro, SSD)
- Memoria RAM (Buffer Manager)
- Variables locales (estado de una transacción)

Las operaciones primitivas de las transacciones son las siguientes

- `PIN(`X`)`: Solicitar X al Buffer Manager.
- `READ(`X, t`)`: Leer X y almacenarlo en la variable local t.
- `WRITE(`X, t`)`: Escribir el contenido de t en X.
- `UNPIN(`X`)`: Soltar X y sacarlo de la memoria.
- `COMMIT`: Guardar los cambios realizados por la transacción
- `ABORT`: Abortar la transacción y no llevar a cabo los cambios.

4.4. Transaction Manager

Recordemos un poco lo que habíamos hablado de ACID. Una de las propiedades de las que habíamos hablado era *"isolation"*.

Propiedad Isolation

Propiedad isolation asegura que, si bien las acciones de varias transacciones pueden ser intercaladas, el resultado final es idéntico a ejecutar todas las transacciones una después de la otra en algún orden secuencial.

Esta es la responsabilidad que tiene el Transaction Manager.

4.4.1. Schedule y concurrencia de transacciones

Un schedule S es una secuencia de operaciones primitivas de una o más transacciones, tal que para toda transacción T_i, las acciones de T_i aparecen en el mismo orden en S.

Un schedule se dice que es serial si todas las acciones de una transacción se ejecutan en grupo, sin intercalar.

Un schedule se dice que es serializable si existe un schedule S' tal que el resultado de S y S' es el mismo dado un mismo estado inicial.

Ejemplos

Se tiene T_1 y T_2

T_1	T_2
`READ(A,x)`	`READ(A,y)`
`x := x + 100`	`y := y * 2`
`WRITE(A,x)`	`WRITE(A,y)`
`READ(B,x)`	`READ(B,y)`
`x := x + 200`	`y := y * 3`
`WRITE(B,x)`	`WRITE(B,y)`

Entonces, un schedule serial sería

Paso	T_1	T_2
1	`READ(A,x)`	
2	`x := x + 100`	
3	`WRITE(A,x)`	
4	`READ(B,x)`	
5	`x := x + 200`	
6	`WRITE(B,x)`	
7		`READ(A,y)`
8		`y := y * 2`
9		`WRITE(A,y)`
10		`READ(B,y)`
11		`y := y * 3`
12		`WRITE(B,y)`

Un schedule serializable sería

Paso	T_1	T_2
1	`READ(A,x)`	
2	`x := x + 100`	
3	`WRITE(A,x)`	
4		`READ(A,y)`
5		`y := y * 2`
6		`WRITE(A,y)`
7	`READ(B,x)`	
8	`x := x + 200`	
9	`WRITE(B,x)`	
10		`READ(B,y)`
11		`y := y * 3`
12		`WRITE(B,y)`

4.4.2. ¿Es serializable?

Es muy dificil concluir si una transacción es o no es serializable. Para tratar de definir esto lo mejor posible, nos enfocaremos únicamente en los `READ` y `WRITE` que realizan las transacciones.

Leyenda

Para facilitar la lectura y escritura, se usará la siguiente notación:

- T_i : `READ(X,t)` $\rightarrow r_i(X)$
- T_i : `WRITE(X,t)` $\rightarrow w_i(X)$

Así, podemos escribir el schedule serializable de antes de esta forma:

$$S: \ r_1(A)\ w_1(A)\ r_2(A)\ w_2(A)\ r_1(B)\ w_1(B)\ r_2(B)\ w_2(B)$$

4.4.3. Conflict Serializable

Dos acciones t_1 y t_2 son **NO** conflictivas si para toda secuencia de acciones u y v, se tiene que

$$u\ t_1\ t_2\ v \quad \text{es equivalente a} \quad u\ t_2\ t_1\ v$$

Cuando esto ocurre, se denota como

$$u\ t_1\ t_2\ v \vdash u\ t_2\ t_1\ v$$

Para toda secuencia de acciones u y v se dice que $u \vdash^* v$ si $u \vdash v$ o si existe una secuencia de acciones w tal que

$$u \vdash^* w \quad \text{y} \quad w \vdash^* v$$

Con todo esto, llegamos a la definición de Conflict Serializable. Se dice que un schedule S es conflict serializable si existe un schedule S' tal que

$$S \vdash^* S'$$

Si S es conflict serializable, entonces S también es serializable. Esto **NO** funciona al revés.

Grafo de Precedencia

Sea S un schedule de transacciones $T_1, \ldots, T_n$. Para dos transacciones T_i, T_j se dice que T_i precede a T_j en S ($T_i <_S T_j$) si $S = u\ p_i(X)\ w\ q_j(Y)\ v$, donde u, v y w son secuencias de acciones de S y $p_i(X)$ y $q_j(X)$ son acciones conflictivas.

Esta relación $<_S$ define el grafo de precedencia. Así, se dice que un schedule es conflict serializable si y solo si el grafo de precedencia de S, $\mathcal{G}_S$, es acíclico.

4.5. Control de concurrencia basado en locks

Los locks consisten en un sistema en donde, en el momento que una transacción solicita acceso a algun elemento de la base de datos, este se bloquea, impidiendo que otras transacciones hagan cosas en ese elemento.

Para esto, es necesario implementar una interfaz que incluya `requestLock(element)` y `releaseLock(element)`.

4.5.1. Principios del uso de locks

- Consistencia: Las transacciones pueden leer o escribir un elemento solo si tienen su lock correspondiente. También, las transacciones deben hacer release del lock en algún momento (idealmente hacerlo apenas terminen de trabajar con el elemento en cuestión).
- Validez: Solo una transacción puede tener el lock en cualquier momento.

4.5.2. Two-Phase Locking (2PL)

Toda transacción pasa por dos fases:

1. Lock phase: Los locks son pedidos a medida que se necesitan.
2. Release phase: Los locks son liberados si ya no se necesitan.

Teorema: 2PL

2PL produce únicamente conflict serializable schedules.

En el papel este sistema suena bastante bonito, pero en realidad tiene varios problemas...

- Cascading Rollback: 2PL permite lecturas sucias. → Solución: Strict 2PL.
- Interferencia entre `READ`s: Hay esperas innecesarias ocasionadas por lecturas. Una lectura no debería bloquear. → Solución: Lock modes.
- Deadlocks: 2PL puede generar un schedule que no puede avanzar y se queda atascado indefinidamente. → Solución: Detección y prevención de deadlocks.

Strict 2PL

Los locks no se sueltan si ya no se necesitan. Se sueltan solamente cuando la transacción hace `ABORT` o `COMMIT`. Esto es más sencillo de implementar y no requiere de cascading rollback.

Lock modes

Se pueden usar dos tipos de locks: Shared locks (slock, para leer), y exclusive locks (xlock, para escribir).

	Lock solicitado	
Locks Actuales	S	X
S	Si	No
X	No	No

Teorema: Strict 2PL + Lock modes

2PL y Lock modes produce solamente conflict serializable schedules.

Detección de Deadlocks

- Detección de ciclos en grafo de espera.
- Timeout.
- Prevención de deadlocks (timestamps): wait-die; wound-wait.

4.6. Control de Concurrencia basado en Timestamps

Cada transacción T recibe un timestamp TS(T). Este puede ser asignado por un clock interno del sistema, o por medio de un contador. Así, toda transacción que inicia más tarde tiene un timestamp mayor.

4.6.1. Elementos de la Base de Datos

Para cada elemento de la base de datos se define

- $RT(X)$ (Read-time de X): El timestamp mayor de todas las transacciones que han leído X.
- $WT(X)$ (Write-time de X): El timestamp mayor de todas las transacciones que han escrito X.
- $C(X)$ (Commit de X): 1 si y solo si la última transacción que escribió en X hizo commit.

4.6.2. Manejo de concurrencia

Cuando T selecciona $r(x)$ o $w(x)$, el control de concurrencia debe tomar algunas decisiones

- Otorgar el permiso de X a T.
- Rollback de T (y empezar T con un nuevo timestamp).
- Retrasar T hasta que $C(X) = 1$.

Se define el siguiente protocolo

- T desea leer X:
  - Si $TS(T) \geq WT(X)$:
    - Si $C(X) = 1$: Permitir la lectura de X.
    - Si $C(X) = 0$: Retrasar T hasta que $C(X) = 1$.
  - Si $TS(T) < WT(X)$: Rollback de T.
- T desea escribir X:
  - Si $TS(T) \geq RT(X)$ y $TS(T) \geq WT(X)$: Permitir la escritura de X y actualizar $C(X) = 0$, $WT(X) := TS(T)$.
  - Si $TS(T) \geq RT(X)$ y $TS(T) < WT(X)$:
    - Si $C(X) = 1$: Ignorar la escritura de X.
    - Si $C(X) = 0$: Retrasar T hasta que $C(X) = 1$.
  - Si $TS(T) \geq RT(X)$: Rollback de T.
- T desea hacer `COMMIT`:
  - Buscar todos los elementos X actualizados por T.
  - Actualizar $C(X) := 1$ para cada X modificado.
- T desea hacer `ROLLBACK`:
  - Buscar todas las transacciones T' que están a la espera de T.
  - Otorgarles permiso.
  - Verificar si la acción de T es válida para el schedule.

4.6.3. Multiversion Timestamps

Consiste en mantener múltiples versiones de un mismo elemento, por medio de timestamps. Esto nos permitirá ahorrarnos algunos rollbacks.

A esto también se le conoce como MVCC o Multiversion Concurrency Control.

4.7. Snapshot Isolation

Control de concurrencia optimista y multiversión. Aquí, cada transacción utiliza durante su ejecución, la última versión de los datos disponible al momento de iniciar.

Capítulo 5

Recuperación de fallas

5.1. Políticas Steal y Force para el Buffer Manager

- Steal frame: *¿Puede un elemento X en memoria ser escrito en el disco antes de que la transacción T termine?*
  - Steal: Se puede mover al disco tan pronto como sea posible.
  - No-Steal: No se puede mover al disco hasta que se haga `COMMIT`.
- Force page: *¿Es necesario que una transacción haga `FLUSH` de todos los elementos modificados inmediatamente antes o después de un `COMMIT`?*
  - Force: Debe ser movido al disco antes de hacer `COMMIT`.
  - No-Force: Puede ser movido en cualquier momento, incluso después del `COMMIT`.

No-Steal + Force es fácil de recuperar, pero tiene el peor rendimiento. Por otro lado, Steal + No-Force no es tan fácil de recuperar, pero entrega mejor rendimiento. Este último es el más usado.

5.2. Log Manager

El Log Manager lleva un registro de todas las acciones de las transacciones. Todas las transacciones pueden escribir en el log de manera recurrente, y el log va llenando secuencialmente una página en el buffer. Luego, cuando esta página se llena, se almacena en el disco (secuencialmente también).

Existen varios tipos de Log Manager, pero todos comparten en común los siguientes tipos de entrada.

- `<START` T `>`
- `<COMMIT` T `>`
- `<ABORT` T `>`
- `<`T *`update`* `>`

5.2.1. Undo-Logging

Aquí, <T *update* > $\rightarrow$ < T , X , t >, donde t es el valor antiguo de X.

Reglas

- Regla U_1: "Si T modifica X, el log record < T , X , t > debe ser escrito en el disco antes de que X sea escrito en el disco."
- Regla U_2: "Si T hace COMMIT, el log record <COMMIT T > debe ser escrito en el disco justo después que todos los datos modificados por T sean guardados en el disco."

Procedimiento

Para recuperarse de una falla usando Undo-Logging, se debe leer el log desde el final hasta el inicio.

- <START T >: Ignorar
- <COMMIT T >: Marcar T como realizada.
- <ABORT T >: Marcar T como realizada.
- < T , X , t >: Restituir $X := t$ en el disco si T no fue realizada.

Checkpoints

Hacemos undo recovery, hasta que llegamos a un checkpoint. Una vez que se llega a un checkpoint, se asume que todo lo que está antes está en el disco, y que todo funcionó como corresponde, por lo que no es necesario seguir.

5.2.2. Redo-Logging

Aquí, <T *update* > $\rightarrow$ < T , X , t >, donde t es el valor nuevo de X.

Reglas

- Regla R_1: "Antes de modificar cualquier elemento X en disco, es necesario que todos los logs records estén almacenados en disco, incluido el <COMMIT T > log record."

Procedimiento

Para recuperarse de una falla usando Redo-Logging, se debe leer el log desde el inicio hasta el final.

- Se identifican las transacciones que hicieron commit.
- Se hace un scan desde el inicio.
- Para cada log record < T , X , t >, no se hace nada si T no hizo COMMIT. Si T logró hacer COMMIT, se reescriben los cambios en el disco acorde al log.
- Para cada transacción que quedó incompleta, se escribe <ABORT T >.

Checkpoints

Al usar checkpoints, se sabe que al leer <END CKPT >, se garantiza persistencia de las transacciones que hicieron COMMIT.

5.2.3. Undo/Redo-Logging

Aquí, `<T update >` $\rightarrow$ `<` T`, ` X`, ` t_O`, ` t_N `>`, donde t_O es el valor antiguo de X y t_N es el valor nuevo de X.

Reglas

- Regla U_1: "Si T modifica X, el log record `<` T`, ` X`, ` t `>` debe ser escrito en el disco antes de que X sea escrito en el disco."
- Regla U_2: "Si T hace `COMMIT`, el log record `<COMMIT T >` debe ser escrito en el disco justo después que todos los datos modificados por T sean guardados en el disco."
- Regla R_1: "Antes de modificar cualquier elemento X en disco, es necesario que todos los logs records estén almacenados en disco, incluido el `<COMMIT` T `>` log record."
- Regla UR_1: "Antes de modificar cualquier elemento X en disco, es necesario que el log record `<` T`, ` X`, ` t_O`, ` t_N `>` esté registrado en el log."

Procedimiento

Para recuperarse de una falla usando Undo/Redo-Logging, se debe hacer redo de todas las transacciones que hicieron commit, desde el principio hasta el final del log y también se debe hacer undo de todas las transacciones que quedaron incompletas, desde el final hasta el principio del log.

Capítulo 6

Anexo: Algoritmos

6.1. Operadores Relacionales

Cada operador físico implementa un operador relacional lógico, y está orientado a desempeñar una tarea específica.

Existen múltiples operadores físicos, como la selección (σ), la proyección (π), y sorting (τ).

6.1.1. Sorting

Se considera que un conjunto de tuplas $\{t_1, \ldots, t_n\}$ está ordenado respecto a una secuencia $(k_1, \ldots, k_m)$ si

$$t_i \leq_{k_1,\ldots,k_m} t_{i+1} \Rightarrow t_i(k_1) < t_{i+1}(k_1) \vee (t_i(k_1) = t_{i+1}(k_1) \wedge t_i \leq_{k_2,\ldots,k_m} t_{i+1})$$

Existen varios algoritmos para ordenar. Nos vamos a enfocar en MergeSort. MergeSort se basa en ordenar un par de listas que ya están ordenadas, para mezclarlas y dejarlas como una sola lista ordenada.

Algorithm 2: merge(L_1, L_2)

Input: Dos listas ordenadas L_1 y L_2
Output: Una lista ordenada L
L = lista vacía;
while L_1 *y* L_2 *son ambas no vacias* **do**
    Quitar el menor elemento m entre L_1 y L_2;
    Agregar m al final de L
if L_1 *está vacía* **then**
    Remover elementos de L_2 y agregar al final de L.;
else if L_2 *está vacía* **then**
    Remover elementos de L_1 y agregar al final de L.;
return L

Algorithm 2: mergesort(L)

Input: Una lista de numeros $L = a_1, \ldots, a_n$.
Output: L en orden creciente.
if $n = 1$ **then**
    return L
else
    $m = \lfloor n/2 \rfloor$;
    $L_1 = a_1, a_2, \ldots, a_m$;
    $L_2 = a_{m+1}, a_{m+2}, \ldots, a_n$;
    return *merge(mergesort(L_1), mergesort(L_2))*

Existe una alternativa llamada External Merge Sort, la cual hace una tarea similar a Merge Sort, pero guardando los elementos del disco en un buffer de memoria, para en ese momento fusionar ambas listas y guardarlas al disco.

Algorithm 3: externalMergeSort(P)

Input: Páginas $P = \{p_1, \ldots, p_n\}$ con tuplas.
Output: Lista de páginas P' con las tuplas ordenadas.
foreach $p \in P$ **do**
    Leemos p a memoria.;
    Ordenamos p en r.;
    Escribimos r en disco.
$R = \{r_1, \ldots, r_n\}$ **while** $|R| > 1$ **do**
    $R' = \emptyset$;
    foreach $r_1, r_2 \in R$ **do**
        Leemos r_1 y r_2 a memoria. ($R = R - \{r_1, r_2\}$);
        Mergeamos r_1 y r_2 ($r = \text{merge}(r_1, r_2)$);
        Escribimos r en disco ($R' = R' \cup \{r\}$)
    $R = R'$
return *Único run en R*

El costo I/O de este algoritmo, considerando N el número de páginas del archivo, es de $2 \cdot N \cdot (1 + \lceil \log_2(N) \rceil)$. Esto significa que un archivo de 8GB con 1.048.576 páginas tomará 1.2 horas en hacer sort. Esto es mucho.

El algoritmo puede ser optimizado si es que se aumenta el número de runs en el merge, se reduce el número de runs iniciales y se evita escribir el último resultado. Con estas cosas el algoritmo queda así

Algorithm 4: externalMergeSort(P) optimizado

Input: Páginas $P = \{p_1, \ldots, p_n\}$ con tuplas y $B + 1$ páginas en buffer.
Output: Lista de páginas P' con las tuplas ordenadas.
while $P \neq \emptyset$ **do**
    Leemos $\{p_1, \ldots, p_{B+1}\}$ a memoria ($P = P - \{p_1, \ldots, p_{B+1}\}$);
    Ordenamos $\{p_1, \ldots, p_{B+1}\}$ con quicksort y creamos un run r;
    Escribimos r en disco
$R = \{r_1, \ldots, r_{\frac{n}{B+1}}\}$;
foreach $|R| > 1$ **do**
    $R' = \emptyset$;
    while $R \neq \emptyset$ **do**
        Leemos $r_1, \ldots, r_B$ a memoria ($R = R - \{r_1, \ldots, r_B\}$);
        Hacemos merge de $r_1, \ldots, r_B$ ($r = merge(r_1, \ldots, r_B)$);
        Escribimos r en disco ($R' = R' \cup \{r\}$)
    $R = R'$
return *Único run en R*

Ahora su costo total en I/O es de $2 \cdot N \cdot (1 + \lceil \log_B(\lceil \frac{N}{B+1} \rceil) \rceil)$. Con eso, el mismo ejemplo de antes, y considerando un buffer de 2GB de RAM, es posible ordenar en tan solo 6.7 minutos.

Como nota, los parámetros de costo de sorting son los siguientes

- $\text{cost}(\tau(R)) = \text{cost}(R) + 2 \cdot \text{pages}(R)$
- $\text{pages}(\tau(R)) = \text{pages}(R)$
- $|\tau(R)| = |R|$
- $\text{rsize}(\tau(R)) = \text{rsize}(R)$

6.1.2. Selección

La selección es el proceso de elegir ciertos datos de una relación que calcen con un cierto criterio.

La selección se puede dar en varios casos: sin índices, con índice primario, con índice secundario y con distintos filtros. Cada uno de estos casos genera pequeños cambios en el algoritmo, tal como se mostrará ahora.

Para el caso sin índices, la única forma de filtrar es leer cada una de las tuplas, y retornarla si satisface el predicado.

Algorithm 5: select(p, R) no index

Input: Predicado p y relación R
Function `open()`:
  R.open()
Function `close()`:
  R.close()
Function `next()`:
  $t = R$.next()
  while $t \neq$ *NULL* **do**
    if $p(t) =$ *true* **then**
      return t
    $t = R$.next()
  return *NULL*

Costo y parámetros de $\sigma_p(R)$

- $\text{cost}(\sigma_p(R)) = \text{cost}(R)$
- $\text{pages}(\sigma_p(R)) = \text{sel}_p(R) \cdot \text{pages}(R)$
- $|\sigma_p(R)| = \text{sel}_p(R) \cdot |R|$

Para el caso de índice primario, se busca la primera tupla que satisface $A = v$. Luego se retornan las siguientes tuplas haciendo next en el índice.

Algorithm 6: select(p, R) primary index

Input: Predicado $p := A = v$, relación R e índice I
Function `open()`:
  I.open()
  I.search($A = v$)
Function `close()`:
  I.close()
Function `next()`:
  $t = I$.next()
  if $t \neq$ *NULL* **then**
    return t
  return *NULL*

Costo y parámetros de $\sigma_p(R)$

- $\text{cost}(\sigma_p(R)) = \text{cost}(I) + \text{sel}_p(R) \cdot \text{pages}(R)$
- $\text{pages}(\sigma_p(R)) = \text{sel}_p(R) \cdot \text{pages}(R)$
- $|\sigma_p(R)| = \text{sel}_p(R) \cdot |R|$

Para el caso de índice secundario, se busca la primera tupla que satisface $A = v$. Luego, por cada data entry $k*$ que satisface $A = v$, se busca la pagina en $k*$.RID. Finalmente, se retorna la tupla con RID igual a $k*$.RID.

Algorithm 7: select(p, R) secondary index

Input: Predicado $p := A = v$, relación R e índice I

Function `open()`:
- I.open()
- I.search($A = v$)

Function `close()`:
- I.close()

Function `next()`:
- $k* = I$.next()
- **if** $k* \neq$ *NULL* **then**
  - **return** $R.get(k*.RID)$
- **return** *NULL*

Costo y parámetros de $\sigma_p(R)$

- $\text{cost}(\sigma_p(R)) = \text{cost}(I) + \text{sel}_p(R) \cdot |R|$
- $\text{pages}(\sigma_p(R)) = \text{sel}_p(R) \cdot \text{pages}(R)$
- $|\sigma_p(R)| = \text{sel}_p(R) \cdot |R|$

6.1.3. Proyección

La proyección tiene como fin mostrar cierta columna L de una relación R.

Algorithm 8: project(p, R)

Input: Predicado $p := A = v$, relación R e índice I

Function `open()`:
- I.open()
- I.search($A = v$)

Function `close()`:
- I.close()

Function `next()`:
- $t = R$.next()
- **if** $t \neq$ *NULL* **then**
  - **return** $t.project(L)$
- **return** *NULL*

Costo y parámetros de $\pi_L(R)$

- $\text{cost}(\pi_L(R)) = \text{cost}(R)$
- $\text{pages}(\pi_L(R)) = \frac{\text{rsize}(\pi_L(R))}{\text{rsize}(R)} \cdot \text{pages}(R)$
- $|\pi_L(R)| = |R|$
- $\text{rsize}(\pi_L(R)) = \sum_{\text{att} \in L} \mathbb{E}(|\pi_{\text{att}}(R)|)$

6.1.4. Join

Nested loops join

Algorithm 9: join(R, S)

Input: Predicado p, operadores R y S

Function `open()`:
- R.open()
- S.open()
- $r = R$.next()

Function `close()`:
- R.close()
- S.close()

Function `next()`:
- **while** $r \neq$ *NULL* **do**
  - $s = S$.next()
  - **if** $s =$ *NULL* **then**
    - S.close()
    - $r = R$.next()
    - S.open()
  - **else if** (r, s) *satisfacen* p **then**
    - **return** (r, s)
- **return** *NULL*

Costo y parámetros de $R \bowtie_p S$

- $\text{cost}(R \bowtie_p S) = \text{cost}(R) + |R| \cdot \text{cost}(S)$
- $\text{pages}(R \bowtie_p S) = \text{sel}_p(R \times S) \cdot \text{pages}(R) \cdot \text{pages}(S)$
- $|R \bowtie_p S| = \text{sel}_p(R \times S) \cdot |R| \cdot |S|$
- $\text{rsize}(R \bowtie_p S) = \text{rsize}(R) + \text{rsize}(S)$

6.2. Estimación de Costos

Para estimar que tan computacionalmente costosa puede llegar a ser una operación, se tienden a usar los siguientes parámetros

- $\text{cost}(R)$: Costo en I/O para computar R[1]
- $\text{pages}(R)$: Cantidad de páginas necesarias para almacenar R.
- $|R|$: Cantidad de tuplas en R.
- $\text{rsize}(R)$: Tamaño de una tupla en promedio en R.
- $\text{distinct}(R)$: Cantidad de elementos distintos en R.
- $\text{distinct}_a(R)$: Cantidad de elementos distintos en R, según la columna a.
- $|\text{page}|$: Tamaño de una página.
- $\text{sel}_p(R)$: Fracción de tuplas en R que satisfacen p.[2]

[1] R: Relación que está participando en la operación.

[2] Se habla de alta selectividad cuando pocos elementos de R satisfacen a p, osea, cuando $\text{sel}_p(R) \sim 0$

Capítulo 7

Notas finales

A la fecha actual, este resumen está incompleto. Faltan varios contenidos del curso, por lo que no se aconseja utilizar este apunte como un sustituto para las clases.

Agradecimientos especiales para el profesor Domagoj Vrgoč por compartir con nosotros su conocimiento e inspirarnos a comprender bien este tópico. Clases de buena calidad ocurren gracias a docentes como usted. Nunca deje de tratar de hacer reir a sus alumnos en clase, y jamás se venda a la maligna entidad corporativa llamada Oracle.

Y también, gracias a ti por leer.

Hecho con ♡ por Andrés Cabezas en LaTeX.