TEMA I- Diseño Físico[1]

OBJETIVOS DEL TEMA: Saber responder a las siguientes preguntas:

- ¿Cuál es el propósito del diseño físico de una base de datos?
- ¿Cómo almacena y accede a datos persistentes un SGBD?
- ¿Cómo organiza un SGBD los archivos de datos en disco para minimizar el coste de E/S?
- ¿Qué es un índice y por qué se usa?
- ¿Cómo funcionan los índices Hash y B+, y cuándo son más efectivos?
- ¿Cómo se pueden utilizar índices para optimizar el rendimiento para una carga de trabajo dada?

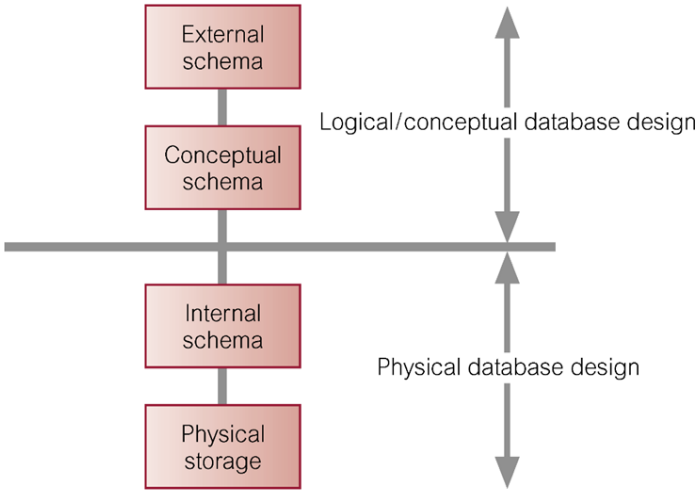
1. Ciclo de vida del desarrollo de sistemas de BD

La base de datos es un componente fundamental de los sistemas de información, y su desarrollo y utilización deben contemplarse desde la perspectiva de los requisitos globales de la organización. La siguiente figura proporciona un resumen de las actividades principales asociadas con cada etapa del ciclo de vida del desarrollo de sistemas de BD.

Stage of database system development lifecycle	Examples of data captured	Examples of documentation produced
Database planning	Aims and objectives of database project	Mission statement and objectives of database system
System definition	Description of major user views (includes job roles or business application areas)	Definition of scope and boundary of database application; definition of user views to be supported
Requirements collection and analysis	Requirements for user views; systems specifications, including performance and security requirements	Users' and system requirements specifications
Database design	Users' responses to checking the logical database design; functionality provided by target DBMS	Conceptual/logical database design (includes ER model(s), data dictionary, and relational schema); physical database design
Application design	Users' responses to checking interface design	Application design (includes description of programs and user interface)
DBMS selection	Functionality provided by target DBMS	DBMS evaluation and recommendations
Prototyping	Users' responses to prototype	Modified users' requirements and systems specifications
Implementation	Functionality provided by target DBMS	
Data conversion and loading	Format of current data; data import capabilities of target DBMS	
Testing	Test results	Testing strategies used; analysis of test results
Operational maintenance	Performance testing results; new or changing user and system requirements	User manual; analysis of performance results; modified users' requirements and systems specifications

[1] El resumen presentado aquí ha sido realizado tomando como referencia básica los libros "*Connolly, T.M.; Begg, C. Sistemas de bases de datos: un enfoque práctico para diseño, implementación y gestión*" y "*Ramakrishnan, R.; Gehrke, J. Sistemas de Gestión de Bases de Datos. McGraw-Hill*", indicados en la bibliografía al final de este documento.

El diseño de la base de datos está compuesto por tres fases principales, denominadas diseño conceptual, diseño lógico y diseño físico, que se corresponden con la arquitectura ANSI-SPARC representada en la siguiente figura:



Durante el diseño conceptual y lógico se obtiene un diagrama Entidad-Relación, un esquema relacional (en caso de haber elegido un SGBD relacional) y la documentación de soporte que describe el modelo, como por ejemplo un diccionario de datos. Toda esta información, tomada conjuntamente, representa el punto de partida para el proceso de diseño físico.

Mientras que el diseño lógico se preocupa de ***qué***, el diseño físico de la base de datos está centrado en el ***cómo***.

2. Diseño físico de una base de datos

El diseño físico de una base de datos es el proceso de generar una descripción de la implementación de la BD en almacenamiento secundario; describe las relaciones base, las **estructuras de almacenamiento en disco** y **métodos de acceso a los datos** más adecuados a los requerimientos de la aplicación, así como cualesquiera restricciones de integridad y medidas de seguridad asociadas.

Los pasos de la metodología física del diseño de datos son los siguientes:

1. Traducir el modelo lógico al SGBD seleccionado
    a. Diseñar las relaciones base
    b. Diseñar la representación de los datos variados
    c. Diseñar las restricciones generales
2. Diseñar la organización de archivos e índices
    a. Análisis de las transacciones
    b. Elección de la organización de los archivos
    c. Elección de índices
    d. Estimación de los requisitos de espacio de disco

3. Diseñar las vistas de usuario
4. Diseñar los mecanismos de seguridad
5. Considerar la introducción de una cantidad controlada de redundancia
6. Monitorizar y ajustar el sistema final

El Paso 1 del diseño físico implica el diseño de las relaciones base y de las restricciones generales. Este paso también considera cómo debemos representar los datos derivados presentes en el modelo de datos.

El Paso 2 implica seleccionar las organizaciones de archivo y los índices para las relaciones base. Generalmente, los SGBD ofrecen varias opciones para organizar los datos del usuario, por lo que los administradores y diseñadores de bases de datos responsables de realizar el diseño físico deben conocer cómo opera el sistema informático que alberga el SGBD, y deben ser plenamente conscientes de la funcionalidad del SGBD elegido.

El Paso 3 implica decidir el modo de cómo debe implementarse cada vista de usuario. El Paso 4 implica diseñar los mecanismos de seguridad necesarios para proteger los datos frente a accesos no autorizados, incluyendo los controles de acceso requeridos para las relaciones base.

El Paso 5 considera la posibilidad de relajar las restricciones de normalización impuestas al modelo lógico de los datos, con el fin de mejorar las prestaciones globales del sistema. El último paso es un proceso continuo de monitorización del sistema final, para identificar y resolver los problemas de prestaciones derivados del diseño, y para implementar requisitos nuevos o modificados.

3. Análisis de las transacciones

En el momento del diseño físico es importante conocer la **carga de trabajo** (combinación de consultas y actualizaciones) que la base de datos debe soportar y los requerimientos de rendimiento de usuario. Esta información forma la base para una serie de decisiones que habrá que durante este paso.

Una de las actividades que componen este paso es el análisis de las transacciones (conjuntos de operaciones de acceso a la base de datos, como inserción, eliminación, modificación o recuperación). Es necesario conocer la funcionalidad asociada a cada una de las transacciones que se ejecutarán en la base de datos, e identificar cuáles son las transacciones prioritarias. Para ello, habrá que tratar de identificar criterios de rendimiento, tales como:

a) Las transacciones que se ejecutan frecuentemente y que tendrán un impacto significativo sobre las prestaciones.

b) Las transacciones que resultan críticas para la operación de la empresa.

c) Los momentos del día y de la semana en los que la demanda de procesamiento será mayor en la base de datos (i.e, el pico de carga).

Esta información se utiliza para identificar las partes de la base de datos que pueden causar problemas de rendimiento.

También se necesita identificar la funcionalidad de alto nivel de las transacciones, como por ejemplo los atributos que son actualizados en una transacción de actualización, o los criterios utilizados para restringir las tuplas que se extraen en una consulta.

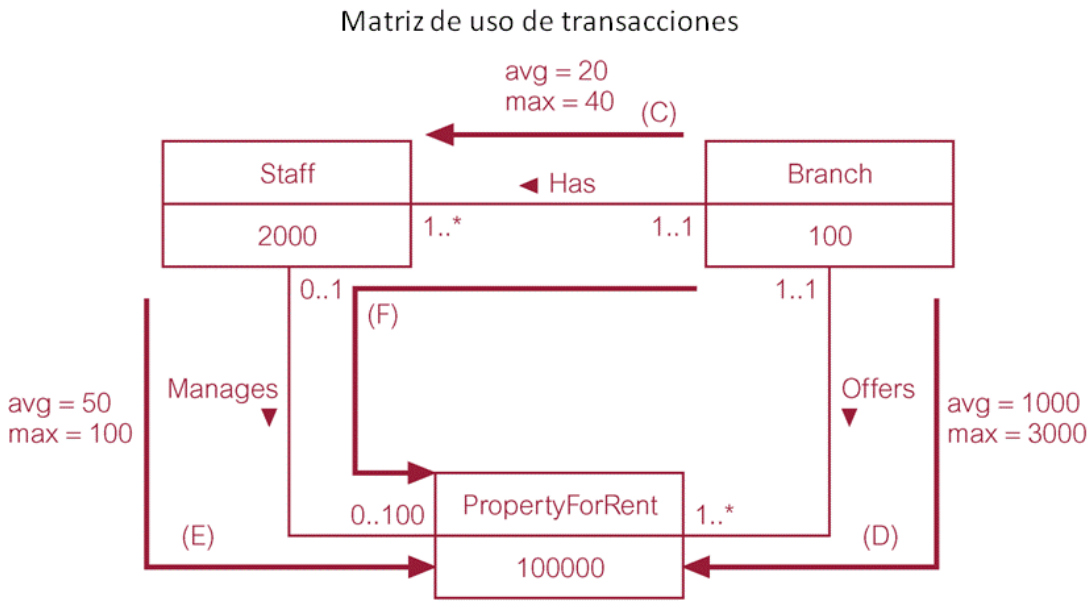
En muchas situaciones, no es posible analizar todas las transacciones esperadas, pero sí que se debe investigar al menos las más importantes. Se ha sugerido que el 20% de las consultas de usuario más activas representan el 80% del acceso total a los datos. Esta regla 80/20 puede utilizarse como directriz a la hora de llevar a cabo el análisis. Como ayuda para la identificación de las transacciones que hay que investigar, se utiliza una *matriz cruzada de transacciones/relaciones* que muestra las relaciones a las que accede cada transacción, y/o un *mapa de uso de las transacciones* que indica cuáles son las relaciones que van a ser potencialmente más utilizadas.

Matriz cruzada de transacciones/relaciones

Transaction/ Relation	(A)				(B)				(C)				(D)				(E)				(F)			
	I	R	U	D	I	R	U	D	I	R	U	D	I	R	U	D	I	R	U	D	I	R	U	D
Branch										X				X								X		
Telephone																								
Staff		X				X				X								X				X		
Manager																								
PrivateOwner	X																							
BusinessOwner	X																							
PropertyForRent	X					X	X	X						X				X				X		
Viewing																								
Client																								
Registration																								
Lease																								
Newspaper																								
Advert																								

I = Insert; R = Read; U = Update; D = Delete

La matriz anterior indica que la transacción (A) lee de la tabla *Staff* y también inserta tuplas en las relaciones *PropertyForRent* y *PrivateOwner*/*BusinessOwner*. Para ser más útil, la matriz debe indicar en cada celda el número de accesos en un cierto intervalo de tiempo (p.ej, cada hora, día o semana). La matriz muestra que tanto la relación *Staff* como *PropertyForRent* son utilizadas por cinco de las seis transacciones, por lo que es importante un acceso eficiente a estas relaciones para evitar la aparición de problemas de rendimiento. Por tanto, concluiremos que es necesaria una inspección más detallada de estas transacciones y relaciones.



La figura anterior muestra un mapa de uso de las transacciones para las transacciones (C), (D), (E) y (F), todas las cuales acceden al menos a una de las dos relaciones *Staff* y *PropertyForRent*.

Al considerar cada transacción, es importante conocer no sólo el número máximo y medio de veces que se ejecuta por hora, sino también el día y la hora en que la transacción se ejecuta, incluyendo cuándo se esperan los picos de carga. Además, debemos determinar:

- Las relaciones y atributos a los que accede la transacción y el tipo de acceso (inserción, actualización, borrado o consulta). *Para una transacción de actualización, debe anotarse los atributos que son actualizados, ya que estos atributos pueden ser candidatos para evitar una estructura de acceso.*

- Los atributos usados en la parte WHERE. Comprobar si implican correspondencia de patrones (*name LIKE '%Smith%'*), búsquedas de rango (*salary between 10000 and 20000)*, o extracción de claves especificadas exactamente (*salary = 20000)*.

  Esto se aplica también a actualizaciones y borrados que puedan restringir las tuplas que hay que actualizar/borrar en una relación.

  Estos atributos pueden ser candidatos al establecimiento de estructuras de acceso.

- Para las consultas, los atributos implicados en la combinación de dos o más relaciones.

  Estos atributos pueden ser candidatos al establecimiento de estructuras de acceso.

- La frecuencia esperada con la que se ejecutarán dichas transacciones; por ejemplo, puede que una transacción vaya a ejecutarse aproximadamente 50 veces al día.

- Los objetivos de rendimiento para la transacción; por ejemplo, que la transacción deba completarse en menos de 1 segundo.

  Los atributos utilizados en los predicados de las transacciones muy frecuentes o críticas deben tener una mayor prioridad a la hora de establecer estructuras de acceso.

La siguiente figura muestra un ejemplo de **formulario de análisis de transacciones** para la transacción (D) que será ejecutada por las 4 oficinas (Branch) de Glasgow.

Transaction Analysis Form **1-Sept-2004**

Transaction (D) List the property number, address, type, and rent of all properties in Glasgow, ordered by rent

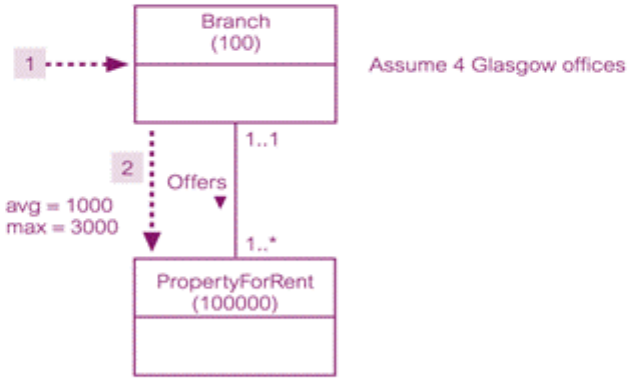
Transaction volume

Average: 50 per hour
Peak: 100 per hour (between 17.00 and 19.00 Monday–Saturday)

```
SELECT propertyNo, p.street, p.postcode, type, rent
FROM Branch b INNER JOIN PropertyForRent p ON
     b.branchNo = p.branchNo
WHERE p.city = 'Glasgow'
ORDER BY rent;
```

Predicate: p.city = 'Glasgow'
Join attributes: b.branchNo = p.branchNo
Ordering attribute: rent
Grouping attribute: none
Built-in functions: none
Attributes updated: none

Transaction usage map



Access	Entity	Type of Access	No. of References		
			Per Transaction	Avg Per Hour	Peak Per Hour
1	**Branch** (entry)	R	100	5000	10000
2	**PropertyForRent**	R	4000–12000	200000–600000	400000–1200000
Total References			**4100–12100**	**205000–605000**	**410000–1210000**

Figure 17.4 Example transaction analysis form.

4. Índices

Un **índice** es una estructura de datos auxiliar que ayuda en la localización de datos bajo una cierta condición de selección. Cada índice posee una **clave de búsqueda** asociada, conjunto de uno o más atributos de un archivo de datos para el que se construye el índice.

La existencia de un índice no afecta a la forma de realizar las consultas SQL, sólo a la velocidad de ejecución. Es la base de datos la que se encarga de mantener y utilizar los índices, una vez que han sido creados, y de reflejar los cambios de los datos (inserción, modificación y borrado de filas) en todos los índices relacionados, sin intervención del usuario.

4.1. Formato de las entradas de datos de índices

El índice está formado por un conjunto de entradas, denominadas **entradas de datos del índice (k*),** que apuntan a los registros de datos que tienen un valor de clave de búsqueda ***k*** concreto.

A la hora de almacenar las entradas de datos del índice, existen tres **alternativas** principales:

[1] ***k****: almacena el registro de datos completo con valor de clave *k*. En este tipo de organización, los registros de datos se almacenan en un índice creado sobre la clave primaria.

Se puede pensar como una organización de archivos especial, denominada *index-organized table* en Oracle, que puede utilizarse en lugar de un *archivo ordenado* o de un *archivo heap*.

[2] **(*k*, *rowid*)**: en este caso, se almacena el valor de la clave y el identificador (*rowid*) del registro con clave de búsqueda *k*.

[3] **(*k*, *lista-rowid*)** donde *lista-rowid* es una lista de identificadores de registro con clave de búsqueda *k*. En este caso, las entradas para un mismo índice son variables en longitud, dependiendo del número de registros de datos que tengan un cierto valor de clave de búsqueda.

La Alternativa [1] se implementa bajo una estructura de árbol balanceado (B-tree). Proporciona un acceso rápido a los registros de datos por clave primaria o por un prefijo válido de la clave. La presencia del resto de atributos del registro en el índice evita una operación extra de E/S para acceder a los datos, así como la necesidad de almacenar el *rowid*.

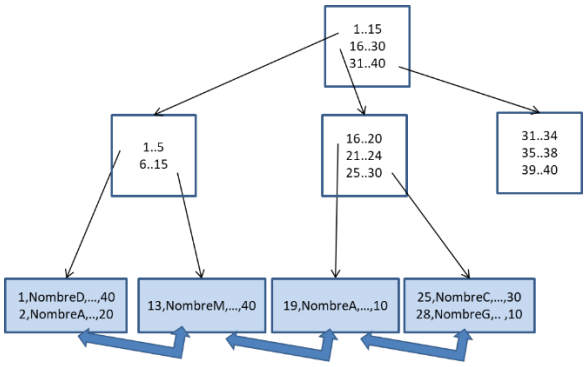
Las Alternativas [2] y [3] son independientes de la organización del archivo de datos. Las entradas de datos del índice son, en general, mucho más pequeñas que los registros de datos. Por ello, son alternativas mejores que la [1], especialmente si las claves de búsqueda son pequeñas. La Alternativa [3] garantiza una mejor utilización del espacio que la Alternativa [2], pero las entradas de datos son variables en longitud, dependiendo del número de registros de datos con el mismo valor de clave de búsqueda.

Ejemplo: Dado un archivo de datos compuesto por registros cuyos campos son NE y NDPTO, entre otros, de la forma:

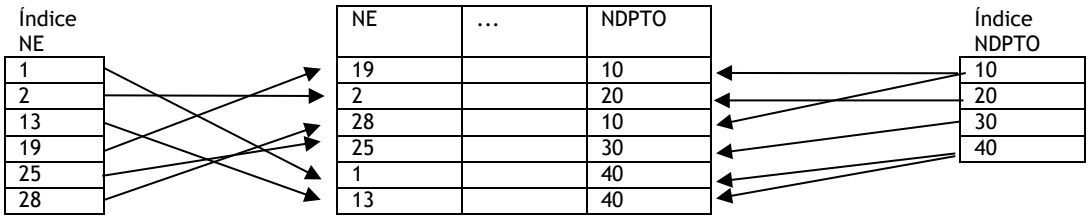
NE	NOMBRE	…	NDPTO
19	NombreA		10
2	NombreA		20
28	NombreG		10
25	NombreC		30
1	NombreD		40
13	NombreM		40

Veamos ejemplos de indexación con las 3 Alternativas:

Alternativa [1] por el campo NE: Las hojas del árbol contienen los registros completos de datos, ordenados secuencialmente por la clave primaria



Alternativa [2] por el campo NE (izquierda) y Alternativa [3] por NDPTO (derecha):



4.2. Estructuras de datos de índices

Los tipos de organización de datos más empleados para la implementación de los índices son las estructuras *basadas en árbol*, las *basadas en hash* y los índices *bitmap***.** La elección de la estructura de datos puede combinarse con las tres alternativas de entradas de datos vistas en el apartado anterior.

4.2.1. Indexación basada en árboles

Las estructuras **basadas en árbol** son un soporte eficiente para las consultas por rango y, excepto en el caso de que los archivos de datos estén ordenados, también son eficientes en inserciones y borrados. Aunque soportan las selecciones de igualdad, no resultan ser estructuras tan adecuadas como las basadas en hash. Dentro de las estructuras basadas en árbol destacan los **árboles B+**.

Una estructura de *árbol-B+* consiste en repartir los valores del índice sobre un bloque (página, unidad mínima de transferencia entre memoria y disco) raíz, unos bloques intermedios y unos bloques hojas. El bloque raíz y los intermedios contienen las direcciones de los otros bloques. Los bloques hoja contienen todos los valores del índice y, para cada valor del índice, el *rowid* del registro que contiene dicho valor del índice (en caso de que el índice siguiese una Alternativa [3], cada entrada tendría varios *ids* en función del número de entradas de datos que tuviesen el mismo valor para el índice).

La técnica del árbol-B+ está basada en los principios siguientes:

- Un árbol-B+ está siempre equilibrado: hay tantos bloques (intermedios y hojas) a la izquierda como a la derecha del bloque raíz. Estos bloques están generalmente equitativamente rellenos.
- Los bloques hoja están todos al mismo nivel, es decir, a igual profundidad en el árbol. Esto hace que los tiempos de respuesta sean los mismos independientemente del valor del índice.
- Los bloques hoja están colocados en orden creciente (o decreciente, según se especifique en el momento de su creación) de izquierda hacia derecha.
- La búsqueda de un valor comienza por la raíz y se termina a nivel de las hojas.
- Los nodos internos se encargan de dirigir la búsqueda a través de las entradas de datos del índice, siendo los nodos hoja los que contienen las entradas de datos del índice (con el valor de la clave y el rowid al registro de datos). Los nodos hoja se enlazan entre sí mediante punteros de página. Mediante una lista doblemente enlazada, es posible atravesar fácilmente la secuencia de nodos hoja en cualquier dirección.

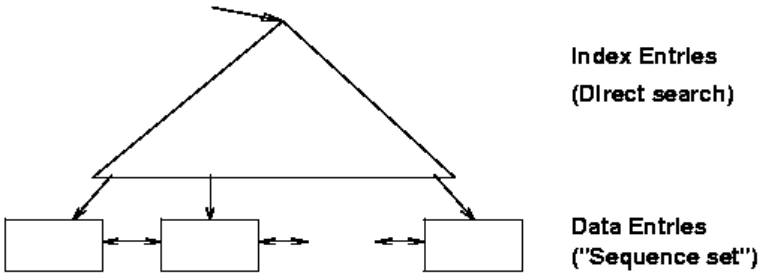


Figura 1.- Estructura de un árbol B+

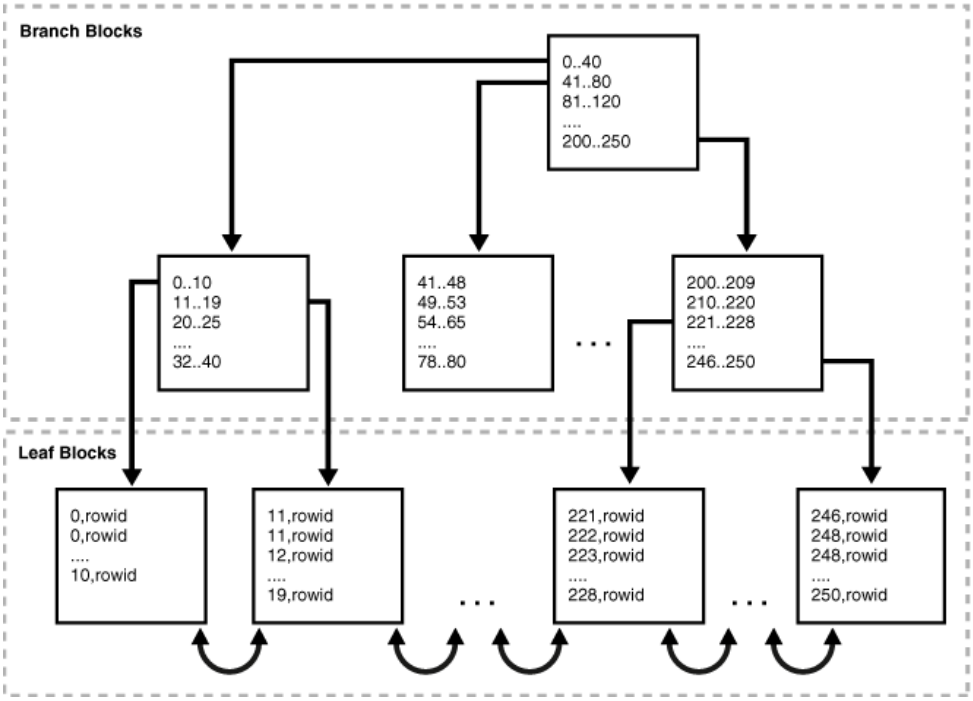


Figura 2.- Estructura interna de un árbol B+

El número de operaciones de E/S realizadas durante una búsqueda en un índice con esta estructura en árbol B+ es igual a la longitud del camino desde la raíz a una hoja, más el número de páginas hoja con entradas de datos que satisfacen la condición de búsqueda.

Buscar la página hoja correcta en un árbol B+ es más rápido que la búsqueda binaria de las páginas en un archivo ordenado porque cada nodo que no es hoja puede alojar un gran número de punteros a nodos, y la altura del árbol raramente suele ser más de 3 o 4. En la práctica, el número medio de punteros almacenados en cada nodo intermedio es por lo menos 100, lo que quiere decir que un árbol de altura 4 contiene 100 millones de páginas hoja. Por tanto, es posible buscar en un archivo con 100 millones de páginas hoja y encontrar la buscada utilizando 4 operaciones de E/S; una búsqueda binaria del mismo archivo llevaría $log_2\ 100\ 000\ 000$ (más de 25) operaciones de E/S.

4.2.2. Indexación basada en Hash

Una estructura **basada en *hash*** está compuesta por un conjunto de *buckets*. Cada bucket está formado por una **página** (unidad mínima de transferencia entre memoria y disco; es similar al concepto de bloque en Oracle) **primaria**, y 0 o más **páginas de overflow** enlazadas mediante punteros.

Puede determinarse el bucket al que pertenece un registro aplicando una función *hash* a la clave de búsqueda. Dado un número de bucket, una estructura hash permite recuperar la página primaria del bucket en 1 ó 2 accesos E/S en disco.

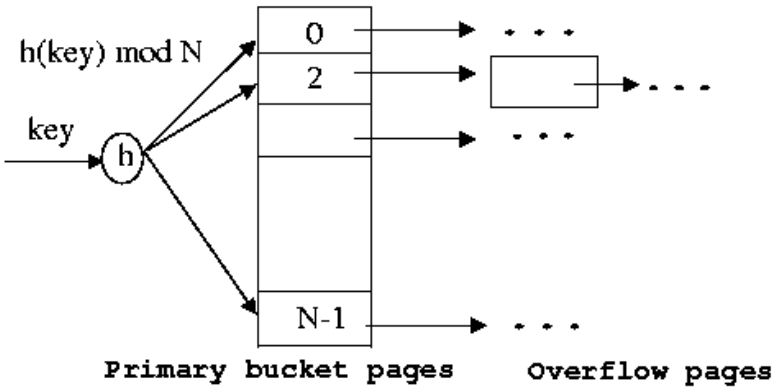


Figura 3.- Estructura basada en hash

Desafortunadamente, las técnicas de indexación basadas en *hash* no soportan las búsquedas por rango. Las técnicas basadas en árbol soportan búsquedas por rango de forma eficiente, y son casi tan eficientes en las selecciones de igualdad que en el caso de indexación basada en *hash*. Por ello muchos sistemas comerciales soportan únicamente índices basados en árbol.

4.2.3. Índices bitmap

En un índice bitmap, la base de datos almacena un bitmap para cada valor de clave. A diferencia de un índice B+, donde cada entrada de datos del índice apunta a un único registro de datos, en un índice bitmap, cada entrada de datos del índice almacena punteros a varios registros de datos.

Cada bit en el bitmap se corresponde a un posible rowid. Si el bit está activado, significa que el registro con el rowid correspondiente contiene ese valor clave. Existe una función de mapeo que convierte la posición del bit al rowid actual. De este modo, el índice bitmap proporciona la misma funcionalidad que un índice B+ pero utiliza una representación interna diferente.

Ej: SELECT ID, LAST_NAME, MARITAL_STATUS, GENDER
FROM CUSTOMERS
ORDER BY id;

ID	LAST_NAME	MARITAL_STATUS	GENDER
1	Kessel		M
2	Koch		F
3	Emmerson		M
4	Hardy		M
5	Gowen		M
6	Charles	single	F
7	Ingram	single	F

Un índice bitmap para la columna Gender sería de la forma:

Value	Rowid1	Rowid2	Rowid3	Rowid4	Rowid5	Rowid6	Rowid7
M	1	0	1	1	1	0	0
F	0	1	0	0	0	1	1

De este modo, la entrada de datos del índice para *M* se representa como *1011100*, y la entrada de datos para *F* se representa como *0100011*.

La función de mapeo asociada convierte cada bit del bitmap a un rowid en la tabla CUSTOMERS. El valor de cada bit depende del valor del campo en el registro de datos correspondiente. Por ejemplo, el primer bit del bitmap para el valor *M* es *1* porque el género para el primer registro de datos es *M*. Los bits 2, 6 y 7 son *0* porque el género para esos registros de datos no es *M*.

Este tipo de índices se utiliza cuando:

- El fichero de datos tiene un número elevado de registros de datos. La indexación de un fichero con muchos registros empleando un índice B+ puede ser prohibitivamente caro en términos de espacio, porque el índice podría ser mayor que los datos del fichero. Por el contrario, los índices bitmap son habitualmente una pequeña fracción del tamaño de los datos indexados.

- Las columnas indexadas tienen baja cardinalidad; es decir, el número de valores distintos es pequeño comparado con el número de registros de datos en el fichero. Por ejemplo, el género de los empleados.

  En general, si el número de valores distintos para la columna indexada es menos de un 1% del número de registros de datos, o si los valores de la columna se repiten más de 100 veces, entonces la columna es candidata para un índice bitmap.

- El fichero indexado es de solo de lectura, o no está sujeto a modificaciones significativas.

- Las consultas son por igualdad, especialmente en combinación con los operadores AND, OR y NOT. Los registros de datos que satisfacen algunas, pero no todas, las condiciones se filtran antes de acceder al fichero de datos. Esto mejora el tiempo de respuesta, en ocasiones de forma muy elevada.

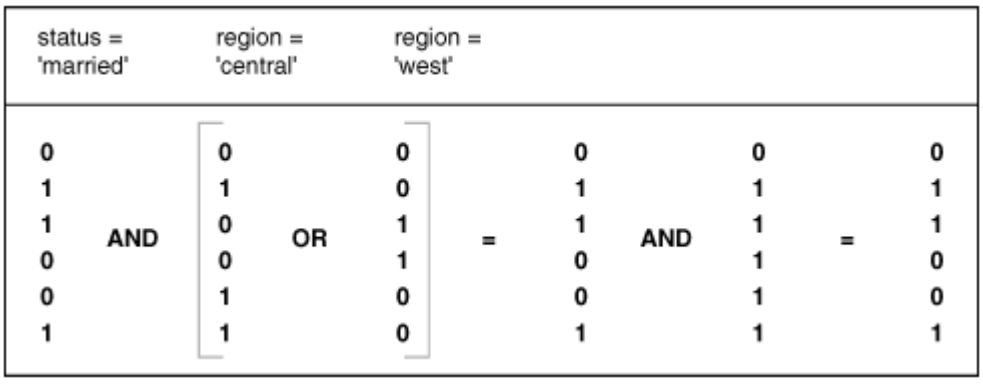
  Ej: Supongamos una tabla con registros como se muestran a continuación:

CUSTOMER #	MARITAL_ STATUS	REGION	GENDER	INCOME_ LEVEL
101	single	east	male	bracket_1
102	married	central	female	bracket_4
103	married	west	female	bracket_2
104	divorced	west	male	bracket_4
105	single	central	female	bracket_2
106	married	central	female	bracket_3

Supongamos que existen índices bitmap para las columnas MARITAL_STATUS y REGION. Entonces, dada la consulta:

```
SELECT COUNT(*) FROM CUSTOMER
WHERE MARITAL_STATUS = 'married' AND REGION IN ('central', 'west');
```

Los índices bitmap pueden procesar esta consulta con gran eficiencia contando el número de unos en el bitmap resultante, como se muestra a continuación:



Para localizar los clientes que cumplen la condición, bastará con utilizar el bitmap final para acceder a la tabla.

4.3. Tipos de índices

4.3.1. Índices únicos y no únicos

Un índice único garantiza que no existen dos filas en la tabla de datos que tengan valores duplicados en la columna (o columnas) clave. Por ejemplo, no existen dos empleados que tengan el mismo ID de empleado. En este caso, existe un único identificador (rowid) para cada entrada de datos.

Un índice no único permite valores duplicados en la columna o columnas indexadas. Por ejemplo, en el campo *salario* de una tabla de empleados podría repetirse el mismo sueldo para varios empleados. En este caso, las entradas de datos del índice estarán ordenadas (en caso de una estructura en árbol B+) por el valor de la clave y el rowid, de forma ascendente.

4.3.2. Agrupados vs. No Agrupados

Se dice que un índice está **agrupado** (**clustered**) si los registros de datos del archivo indexado están ordenados del mismo modo que las entradas de datos del índice. Un índice que usa la Alternativa [1] está agrupado, por definición. Los índices que siguen las Alternativas [2] y [3] estarán agrupados si los registros de datos en el archivo indexado están ordenados por la clave de búsqueda.

Un archivo de datos solo puede agruparse por una clave de búsqueda, lo que significa que solo puede existir un índice agrupado por archivo. Un índice que no está agrupado se denomina **unclustered.**

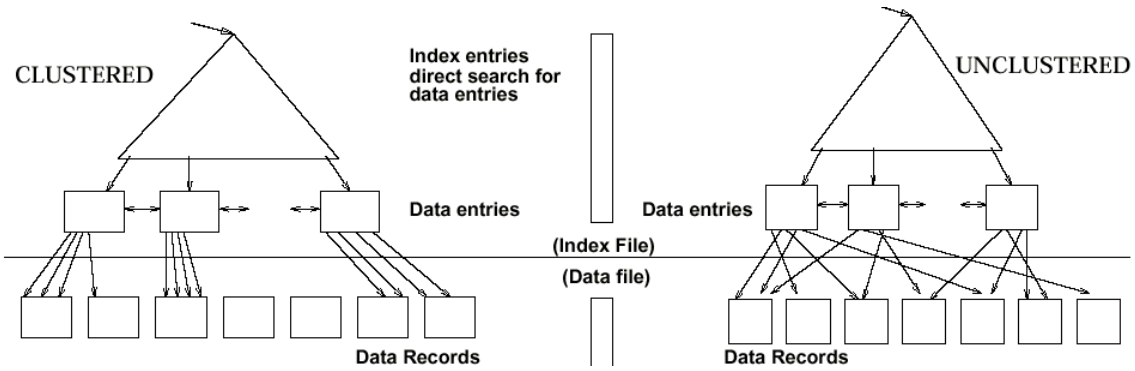


Figura 4.- a) Índice B+ agrupado; b) Índice B+ NO agrupado

El coste en la recuperación de los registros de datos en base a índices varía enormemente en función de si el índice está agrupado o no, sobre todo en el caso de las consultas por rango. Así, si el índice está agrupado, los *ids* apuntan a registros adyacentes, como se muestra en la Figura 4(a), por lo que se recuperan pocas páginas. Si el índice no está agrupado, cada entrada de datos del índice podría contener *ids* que apunten a páginas diferentes (ver Figura 4(b)), lo que acaba en tantas E/S de páginas de datos como el número de entradas de datos del índice que corresponden al rango.

El agrupamiento debe utilizarse pocas veces y solamente cuando esté justificado por consultas frecuentes por rango que se beneficien de él. En particular, generalmente no hay suficientes razones para construir un archivo agrupado utilizando un índice Hash pues, tal como se ha comentado anteriormente, las consultas por rango no pueden realizarse utilizando este tipo de índices. La única excepción es el caso en el que se realice una igualdad con muchos duplicados.

4.3.3. Densos vs. Dispersos (Sparse)

Un índice es **denso** si contiene una entrada de datos del índice **por cada valor de clave de búsqueda** que aparece en un registro del archivo indexado, al menos (porque si se utiliza la Alternativa [2] podría haber duplicados).

Un índice **disperso (sparse)** contiene una entrada de datos del índice **por cada página de registros** en el archivo de datos.

- La Alternativa [1] siempre corresponde a un índice denso.
- La Alternativa [2] puede utilizarse para construir un índice denso o disperso.
- La Alternativa [3] generalmente se utiliza para construir un índice denso.

Ejemplo: En la Figura 5 se muestra un archivo de datos con tres campos (*nombre*, *edad* y *sal*) y 2 índices simples que emplean la Alternativa [2]. El índice de la izquierda es agrupado disperso por *nombre* (el orden de las entradas en el índice y en el archivo de datos coinciden, y hay una entrada de datos del índice por página de registros de datos). El índice de la derecha es no agrupado denso por el campo *edad* (el orden de las entradas del índice es distinto al del archivo de datos, y hay una entrada de datos en el índice por cada registro en el archivo de datos).

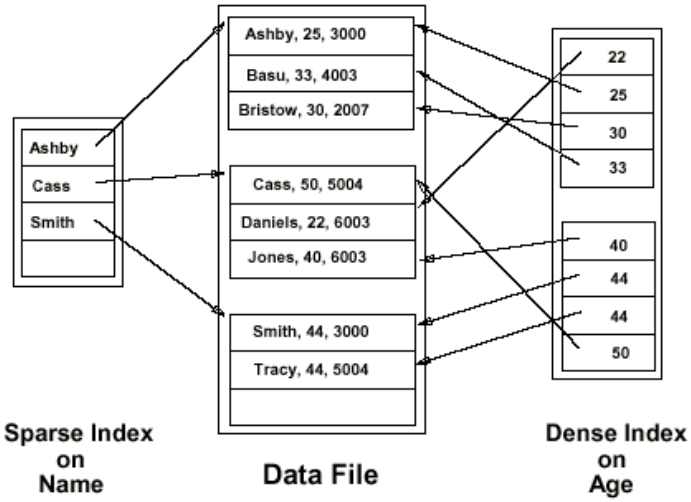


Figura 5.- Ejemplo de índices disperso y denso

No es posible construir un índice disperso que no esté agrupado; esto significa que solo podremos tener como máximo un índice disperso por archivo de datos. Un índice disperso suele ser mucho más pequeño que uno denso.

Los índices hash no pueden ser dispersos. Por definición, la distribución de los datos en un índice hash es disperso, de modo que en cada bucket se asignan registros cuyo valor de clave de búsqueda no es consecutivo necesariamente. Por lo tanto, es necesario que las entradas de datos en el índice hash contengan siempre punteros a registros de datos.

4.3.4. Índices que utilizan claves de búsqueda compuestas

Una clave **compuesta** (o **concatenada**) es aquella formada por varios campos. En la Figura 6 se muestra la diferencia entre índices compuestos y simples. Todos los índices de la figura utilizan la Alternativa [2] para las entradas de datos.

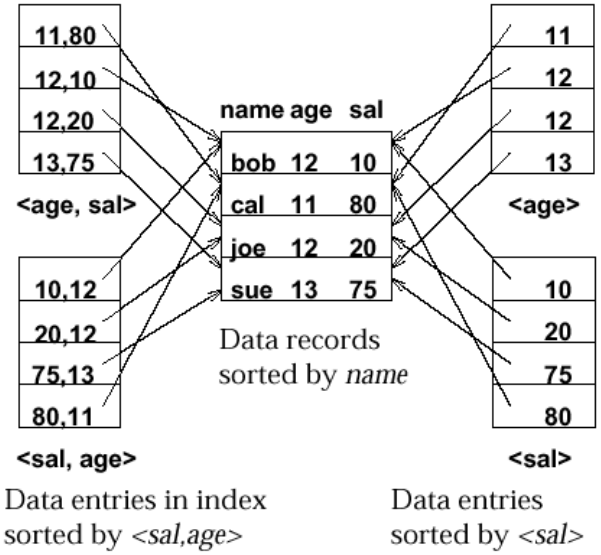


Figura 6.- Índices compuestos

Cuando la clave de búsqueda es compuesta, una **consulta de igualdad** es aquella en la que *todos* los campos son constantes (Ejemplo: *age=20 y sal=10*). **La organización de archivos *hash* únicamente soporta consultas de igualdad, ya que la función *hash* identifica el *bucket* que contiene los registros deseados solo si se especifica un valor para cada campo de la clave de búsqueda.**

Una **consulta por rango** es aquella en la que *al menos uno* de sus campos no es constante (Ejemplo: *age<30 y sal>40*) (Ejemplo: *age=20*; esta consulta implica que se acepta cualquier valor para el campo *sal*). En este caso resulta más adecuada la organización de archivo agrupado y los índices basados en árbol.

Para el siguiente ejemplo:

```
SELECT E.eid
FROM EMP E
WHERE E.age BETWEEN 20 AND 30
        AND E.sal BETWEEN 3000 AND 5000
```

Un índice compuesto <*age, sal*> resulta más adecuado que un índice por *age* o por *sal*. En caso de que se creasen 2 índices, uno por *age* y otro por *sal*, se podrían utilizar ambos para responder la consulta recuperando e intersectando los resultados. Sin embargo, si consideramos qué índices crear para esta consulta concreta, es mejor definir un índice compuesto.

Si las condiciones del WHERE son selectivas equitativamente (es decir, si el número de tuplas resultantes de cada condición del WHERE son similares), crear un índice B+ compuesto agrupado <*age, sal*> es tan efectivo como un índice similar con <*sal, age*>.

No obstante, el orden de los atributos de la clave de búsqueda puede afectar a la eficiencia del resultado:

Supongamos la siguiente consulta:

```
SELECT E.eid
FROM EMP E
WHERE E.age = 25
        AND E.sal BETWEEN 3000 AND 5000
```

Ahora un índice B+ compuesto agrupado <*age*, *sal*> tendrá un mejor rendimiento porque los registros se clasifican, en primer lugar, por *age* y posteriormente (si 2 registros tienen el mismo valor de *age*) por *sal*. Así, todos los registros con *age = 25* se encuentran juntos.

En caso de crear un índice B+ compuesto agrupado <*sal, age*>, los registros se clasifican en primer lugar por *sal*, por lo que 2 registros con el mismo valor de *age* podrían estar muy alejados. De hecho, este índice únicamente permite utilizar la selección del rango de *sal*, pero no la selección por igualdad en *age* para recuperar las tuplas.

4.4. Pautas para la selección de índices

La elección de un conjunto de índices adecuado requiere conocer las técnicas de indexación disponibles, como las presentadas aquí, además de la forma en que trabaja el optimizador de consultas propio del SGBD.

A la hora de crear un índice deben tenerse en cuenta los siguientes aspectos:

1. **creación de un índice:** construir un índice sólo si hay consultas que se beneficien de él. En lo posible elegir índices que afecten a varias consultas.

2. **elección de una clave de búsqueda**: los atributos incluidos en el WHERE, GROUP BY, HAVING y ORDER BY son los candidatos a la indexación.

3. **índices compuestos**: los índices con varios atributos son interesantes cuando la sentencia incluye un WHERE con condiciones que afectan a más de un atributo de una misma relación. Si se crean índices sobre claves de búsqueda con múltiples atributos y se realizan consultas por rango, se deben ordenar los atributos en la clave de búsqueda para que emparejen con las consultas.

4. **agrupación**: las consultas por rango, o aquellas en las cuales los atributos tienen muchos duplicados, son las más beneficiadas por la agrupación (ordenación de los registros de datos en función del índice). Dado que solo se puede crear un índice agrupado por relación, si existen varias consultas sobre una relación que involucren diferentes atributos se debe optar por aquella cuya frecuencia de ejecución sea mayor.

5. **índice *hash* o en árbol**: los índices B+ suelen ser más utilizados, porque soportan tanto las consultas por rango como por igualdad. Un índice *hash* resulta, sin embargo, más adecuado cuando el índice debe soportar *joins* anidados, o cuando se emplea en consultas de igualdad.

6. **analizar el coste de mantenimiento del índice**: es necesario considerar el impacto de cada índice en las actualizaciones.

4.5. Ejemplos de selección de índices

Para los ejemplos que se muestran a continuación consideraremos las siguientes tablas:

EMP (eno, ename, dno, age, sal, hobby)
DEPT (dno, dname, mgr)

que representan una relación 1-N donde cada departamento se relaciona con los empleados que trabajan en el mismo. Además, para cada departamento se almacena el código del empleado que lo dirige (*mgr*).

4.5.1. Creación de índices agrupados

En general, la agrupación resulta interesante para un índice donde la clave de búsqueda no sea una clave candidata (es decir, con duplicados). Así, dada la consulta:

```
SELECT E.dno
FROM EMP E
WHERE E.hobby = 'Stamps'
```

se recomienda el siguiente índice:

Campo	Estructura	Agrupado /No Agrupado	Denso / Disperso
E.hobby si tuplas='Stamps'↓↓	Hash (igualdad)	NO Agrupado (↓↓ duplicados, ordenación cara)	Denso (única opción)

Si se recuperan pocas tuplas, el índice podría ser no agrupado. Sin embargo, si se recuperan muchas tuplas (más del 10% de los registros), la utilización de un índice no agrupado por *E.hobby* resultaría muy ineficiente. En este caso, el coste sería inferior si se recuperan todas las tuplas de la relación y se filtran sobre la marcha.

Supongamos ahora que en lugar de *E.hobby='Stamps'* tuviésemos la condición *E.eno=552*:

```
SELECT E.dno
FROM EMP E
WHERE E.eno = 552
```

Ahora, dado que como mucho el resultado de la consulta sería una tupla, no tendría ninguna ventaja crear un índice agrupado. Por lo tanto, el índice recomendado en este caso sería:

Campo	Estructura	Agrupado /No Agrupado	Denso / Disperso
E.eno	Hash (igualdad)	NO Agrupado (se obtiene solo 1 tupla)	Denso (única opción)

Las consultas por rango también son buenas candidatas a mejorar si se emplea un índice agrupado. Veamos un ejemplo:

```
SELECT E.dno
FROM EMP E
WHERE E.age > 40
```

En primer lugar, un índice B+ en *E.age* resulta de interés solo si el porcentaje de tuplas que cumplen la condición es reducido. Debe tenerse en cuenta que, si se recuperan muchas tuplas, el tiempo de acceso se incrementaría al tener que estar accediendo a dos estructuras (el índice y el archivo de datos); una búsqueda secuencial sería prácticamente igual de buena.

Además, si se crea un índice por *E.age*, conviene que sea agrupado. Si no lo está, podríamos tener que acceder a una página de Entrada/Salida por cada empleado, lo que resultaría más caro que un recorrido secuencial sobre la relación, ¡incluso si se selecciona solo el 10% de los empleados!

En resumen, el índice recomendado para este caso sería de la forma:

Campo	Estructura	Agrupado /No Agrupado	Denso / Disperso
E.age si tuplas *age*>40 ↓↓	B+ (rango)	Agrupado (∃ duplicados, consulta por rango)	Disperso (+ pequeño)
E.age si tuplas *age*>40 ↑↑	No se plantea índice porque sería ineficiente		

Supongamos la siguiente consulta, derivada de la anterior:

```
SELECT E.dno, COUNT(*)
FROM EMP E
WHERE E.age > 50
GROUP BY E.dno
```

En este caso, los índices recomendados son:

Campo	Estructura	Agrupado /No Agrupado	Denso / Disperso
E.age si tuplas *age*> 50 ↓↓	B+ (rango)	Agrupado (∃ duplicados, consulta por rango)	Disperso (+ pequeño)
ó E.dno si tuplas *age*> 50 ↑↑	Hash (igualdad) ó B+ (ordenado)	Agrupado (↑↑ duplicados, los empleados se obtienen ordenados por departamento)	Denso (única opción para Hash) Disperso (si B+ porque es más pequeño)

Como se puede observar en la tabla, se proponen dos índices diferentes en función de lo restrictiva que sea la condición *E.age*. Así, un índice B+ en *E.age* resulta de interés solo si el número de tuplas que cumplen la condición es reducido. En este caso, además será necesario ordenar, posteriormente, las tuplas recuperadas sobre el campo *E.dno* y de esta forma contestar la consulta.

Sin embargo, en caso de que el número de tuplas que cumplan la condición WHERE fuese grande, sería más recomendable crear un índice por *E.dno*. Se podría utilizar este índice para recuperar todas las tuplas correspondientes a cada valor de *E.dno*, y para cada uno de estos valores contar el número de tuplas con *E.age>50*. (Esta estrategia se puede utilizar tanto con índices Hash como de árbol B+; solo es necesario que las tuplas estén agrupadas, no necesariamente ordenadas por *E.dno*).

La eficiencia de este índice depende de nuevo de si es o no agrupado. Si el índice no está agrupado, se podría llegar a realizar una operación de Entrada/Salida por cada tupla de EMP. De hecho, si el índice no está agrupado el optimizador elegirá el plan directo basado en ordenar la tabla por *E.dno*.

En conclusión, el impacto de la agrupación depende del número de tuplas recuperadas; es decir, del número de tuplas que satisfacen las condiciones de selección que emparejan con el índice. Un índice no agrupado es tan bueno como uno agrupado en el caso de una selección que recupera una sola tupla (es decir, una selección de igualdad en una clave candidata). Sin embargo, cuando el número de tuplas recuperadas se incrementa, el índice no agrupado se hace más caro incluso que el recorrido secuencial de la relación entera. Esto es así porque, aunque el recorrido secuencial recupera todos los registros, cada página se recupera exactamente una vez, mientras que si se utiliza un índice no agrupado una página podría recuperarse tantas veces como registros contenga.

4.5.2. Otros ejemplos

Dada la consulta:

SELECT *E.ename, D.mgr*
FROM *EMP E, DEPT D*
WHERE *D.dname = 'Toy'* AND *E.dno = D.dno*

En primer lugar, dado que las condiciones WHERE son de igualdad, se recomienda que los índices creados sean *hash*. También parece claro que se debe construir un índice por *D.dname* y otro por *E.dno*, de modo que una vez recuperados los departamentos que cumplan la condición *D.dname='Toy'* se emparejen las tuplas de *Emp* mediante el atributo *E.dno*. (Obsérvese que no tiene sentido crear un índice adicional por *D.dno* ya que las tuplas de DEPT ya se recuperan utilizando el índice *D.dname*).

Si definimos los índices *D.dname* y *E.dno*, entonces *D.dname* debe ser no agrupado (se supone que el número de tuplas que satisfacen *D.dname='Toy'* es pequeño). Por otro lado, dado que *E.dno* no es una clave candidata (y se supone que puede dar lugar a muchos duplicados) su índice debe ser agrupado.

Por lo tanto, los índices recomendados para mejorar la eficiencia de la consulta son:

Campo	Estructura	Agrupado /No Agrupado	Denso / Disperso
D.dname si tuplas='Toy' ↓↓	Hash (igualdad)	No Agrupado (↓↓ duplicados, ordenación cara, No consulta por rango)	Denso (única opción para índices hash)
E.dno	Hash (igualdad)	Agrupado (No clave, ∃ duplicados)	Denso (única opción para índices hash)

Como variante a la consulta anterior, veamos ahora el siguiente caso:

SELECT *E.ename, D.mgr*
FROM *EMP E, DEPT D*
WHERE *D.dname = 'Toy'* AND *E.dno = D.dno* ***AND E.age = 25***

En este caso, podría considerarse la posibilidad de crear un índice por *E.age* en lugar de *E.dno*. De este modo, podrían recuperarse las tuplas de *DEPT* que satisfagan la selección de *D.dname* (utilizando el mismo índice por *D.dname* que en el caso anterior), recuperar los empleados que satisfagan la selección por *E.age* utilizando dicho índice, y unir ambos conjuntos de tuplas, quedándose únicamente con las comunes a ambos conjuntos.

El plan propuesto es una variante al anterior, especialmente si el índice por *E.age* ya existiese (creado para alguna otra consulta que forme parte de la carga de trabajo). En este caso, no tendría ningún sentido crear un índice por *E.dno*.

Los índices para esta segunda consulta quedarían entonces de la forma:

Campo	Estructura	Agrupado /No Agrupado	Denso / Disperso
D.dname si tuplas = 'Toy' ↓↓	Hash (igualdad)	No Agrupado (↓↓ duplicados, ordenación cara, No consulta por rango)	Denso (única opción)
E.age si tuplas = 25 ↓↓	Hash (igualdad)	No Agrupado (↓↓ duplicados, ordenación cara, No consulta por rango)	Denso (única opción)

4.6. Optimización utilizando índices

Los SGBD disponen de un **optimizador de consultas** que se encarga de crear un plan para la ejecución de las consultas de usuario. El optimizador genera planes alternativos y entre ellos elige aquel que supone un menor coste de ejecución.

Los índices pueden utilizarse de diferentes modos y puede dar lugar a planes que son significativamente más rápidos que otros que no los utilicen:

1. **Plan utilizando un índice**: En caso de que haya varios índices por los campos que forman parte del WHERE, cada uno de ellos ofrecerá un modo diferente de acceso a los registros de datos. El optimizador entonces puede elegir aquel que dé lugar a un menor nº de accesos a páginas de disco.

2. **Plan utilizando múltiples índices**: Si hay varios índices que emparejan con la condición WHERE, se pueden utilizar todos ellos para recuperar conjuntos de identificadores de registros. Después se realiza una intersección de estos conjuntos y se ordena el resultado por el identificador de página (asumiendo que el identificador de registro incluye un identificador de la página donde se ubica).

   Ej: Supongamos la consulta:

   SELECT *E.ename, D.dname*
   FROM *EMP E, DEPT D*
   WHERE *E.sal* BETWEEN 10000 AND 20000
   AND *E.hobby = 'Stamps'* AND *E.dno = D.dno*

   Los índices a construir son:

Campo	Estructura	Agrupado /No Agrupado	Denso / Disperso
D.dno	Hash (igualdad)	No Agrupado (suponiendo pocas tuplas en la tabla)	Denso (única opción)
E.sal	B+ (rango)	Agrupado (No clave, ∃ duplicados, consulta por rango)	Disperso (+ pequeño
y E.hobby	Hash (igualdad)	NO Agrupado (solo puede haber 1 índice agrupado por tabla)	Denso (única opción)

En la tabla DEPT se recomienda la creación de un índice *hash* por *D.dno*. Por otro lado, en la relación EMP se debe construir un índice B+ por el atributo *E.sal* que ayudará en la selección del rango, sobre todo si es agrupado, y/o un índice *hash* por el atributo *E.hobby* para la selección de igualdad.

En la tabla EMP, si se ha definido uno solo de estos índices (E.sal o E.hobby) se pueden recuperar las tuplas de EMP utilizando dicho índice, recuperar las tuplas de DEPT que emparejen utilizando el índice *D.dno*, y aplicar las restantes selecciones y proyecciones "*on-the-fly*".

Por el contrario, si se dispone de ambos índices en EMP y el SGBD aplica la opción 1, elegirá el camino de acceso más selectivo para la consulta; es decir, considerará qué selección (la condición por rango en *E.sal* o la igualdad por *E.hobby*) da como resultado menos tuplas, en función de los datos que existan en el archivo.

Si el SGBD optase por aplicar la opción 2, intersectará los identificadores de los resultados dados por cada índice.

En cualquier caso, *E.hobby* deberá ser No Agrupado, dado que en una tabla solo puede haber un índice agrupado y, en general, los rangos dan lugar a más tuplas que las igualdades.

3. **Plan solo-índice**: Si todos los atributos que forman la consulta (en el SELECT, WHERE, GROUP BY o HAVING) forman parte de un índice *denso* de la relación que aparece en el FROM, algunos gestores (como Oracle) utilizan una **aproximación solo-índice** para obtener el resultado.

   En este caso, dado que las entradas de datos del índice ya contienen todos los atributos necesarios para resolver la consulta, y hay una entrada de datos del índice por tupla, la BD lee directamente del índice los datos que precisa y ya no se recuperan los registros de datos de disco.

   Consecuentemente, bajo esta aproximación generalmente se definirán índices compuestos, formados incluso por atributos que no aparecen en el WHERE pero sí en la parte SELECT. Los índices **nunca serán agrupados**, ya que la ordenación no tiene sentido si no se van a recuperar los registros del archivo de datos. Además, en la estrategia solo-índice estudiada aquí solo se pueden definir **índices densos**, como se ha mencionado.

4.6.1. Ejemplos de índices con planes solo-índice

Veamos ahora ejemplos de consultas que pueden dar lugar a planes eficientes si se utiliza la aproximación solo-índice.

Ejemplo: La siguiente consulta muestra para cada empleado el director de su departamento:

```
SELECT D.mgr, E.eno
FROM EMP E, DEPT D
WHERE E.dno = D.dno
```

En este caso, si el índice en la tabla EMP es un árbol B+ denso por <*E.dno, E.eno*>, en lugar de un índice simple por *E.dno*, ya disponemos de toda la información sobre el empleado en la entrada de datos del índice. Así, podemos utilizar el índice para encontrar la primera entrada para un *E.dno* determinado, y recorrer todas las entradas consecutivas para extraer los números de empleado de ese departamento, sin tener que acceder al archivo indexado. Fíjese que en este caso NO podría utilizarse un índice tipo *hash*, ya que no permite localizar una entrada de datos del índice con un *E.dno* dado, puesto que la función hash se aplicaría sobre ambos atributos a la vez.

Los índices recomendados para EMP y DEPT bajo una aproximación solo-índice de esta consulta serían entonces:

Campo	Estructura	Agrupado /No Agrupado	Denso / Disperso
<E.dno, E.eno>	B+	NO Agrupado (NO tiene sentido agrupar)	Denso (única opción)
<D.dno, D.mgr>	B+	NO Agrupado (NO tiene sentido agrupar)	Denso (única opción)

Como se puede observar, el índice correspondiente a la tabla DEPT debe ser de tipo B+, ya que, como en el caso de la tabla EMP, es necesario acceder a cada valor de *D.dno* para realizar el join. Esto no sería posible si se optase por un índice Hash compuesto. En este último caso, el resultado de la función hash aplicada al mismo D.dno pero diferente D.mgr podría dar lugar a un bucket diferente.

Veamos ahora cómo pueden afectar las funciones de agregación en la elección de los índices:

SELECT *E.dno, COUNT(*)*
FROM *EMP E*
GROUP BY *E.dno*

Si no se hubiesen definido índices sería necesario ordenar la tabla EMP por *E.dno* para poder calcular el nº de empleados de cada departamento. Sin embargo, si se define un índice (*hash* o B+) no agrupado denso por *E.dno*, es posible responder a la consulta utilizando únicamente el índice. Para cada valor *E.dno*, simplemente es necesario contar el nº de entradas de datos del índice con ese valor. En este caso, no tiene sentido que el índice sea agrupado, ya que nunca se recuperan las tuplas de *EMP*, tal como se ha comentado anteriormente.

Por lo tanto, el índice recomendado sería:

Campo	Estructura	Agrupado /No Agrupado	Denso / Disperso
E.dno	Hash (igualdad) o B+ (ordenado)	NO Agrupado (NO tiene sentido agrupar)	Denso (única opción)

Veamos ahora una variante de la consulta anterior:

SELECT *E.dno, COUNT(*)*
FROM *EMP E*
WHERE *E.sal = 10000*
GROUP BY *E.dno*

Ahora un índice simple por *E.dno* no permite evaluar esta consulta con una aproximación *solo-índice*, porque se necesita realizar el filtro *E.sal=10000*. Sin embargo, se puede aplicar dicha aproximación si optamos por crear un índice B+ compuesto denso no agrupado por <*E.sal, E.dno*> o <*E.dno, E.sal*>.

Con un índice de la forma <*E.sal, E.dno*> (el más recomendable) todas las entradas para *E.sal=10000* son contiguas. Además, las entradas están ordenadas por *E.dno*, lo que facilita el cálculo del número de empleados para cada departamento.

Campo	Estructura	Agrupado /No Agrupado	Denso / Disperso
<E.sal, E.dno>	B+ (rango)	NO Agrupado (NO tiene sentido agrupar)	Denso (única opción)

Sin embargo, este índice no es óptimo si la condición WHERE fuese ***E.sal>10000***. Aunque se puede seguir aplicando la aproximación *solo-índice,* es necesario ordenar las entradas de datos del índice por *E.dno* para identificar los grupos, ya que dos entradas con el mismo valor *E.dno* pero diferente *E.sal* podrían no estar contiguas.

Un índice de la forma <*E.dno, E.sal*> es mejor para esta consulta. Ahora las entradas de datos del índice para cada *E.dno* se almacenan consecutivamente, y cada grupo a su vez está ordenado por *E.sal*. Entonces, para cada *E.dno* se pueden eliminar las entradas con *E.sal* menor a 10 000 y contar el resto.

Supongamos ahora una consulta para encontrar el salario mínimo de cada departamento:

SELECT *E.dno, MIN(E.sal)*
FROM *EMP E*
GROUP BY *E.dno*

Un índice formado únicamente por *E.dno* no permite evaluar esta consulta con una aproximación *solo-índice*. Sin embargo, sí sería posible creando un índice compuesto B+ no agrupado denso por <*E.dno, E.sal*>. En este caso, todas las entradas de datos del índice para un *E.dno* dado se almacenan consecutivamente (independientemente de que el índice sea o no agrupado). Además, cada grupo está ordenado por *E.sal*.

Un índice formado por <*E.sal, E.dno*> evitaría también la recuperación de los registros de datos, pero haría necesario ordenar las entradas de datos del índice por *E.dno*.

Campo	Estructura	Agrupado /No Agrupado	Denso / Disperso
<E.dno, E.sal>	B+ (rango)	NO Agrupado (NO tiene sentido agrupar)	Denso (única opción)

Por último, consideremos la siguiente consulta:

SELECT *AVG(E.sal)*
FROM *EMP E*
WHERE *E.age = 25*
AND *E.sal BETWEEN 3000 AND 5000*

Un índice B+ compuesto no agrupado denso por <*E.age, E.sal*> permite resolver la consulta utilizando una aproximación *solo-índice* con un buen rendimiento ya que los registros están ordenados primero por *E.age* y después (si dos registros tienen el mismo valor para *E.age*) por *E.sal*. Un índice por <*E.sal, E.age*> también lo permite, aunque no resulta tan eficiente pues es necesario recuperar más entradas de datos del índice que en el caso anterior dado que dos registros con el mismo valor para *E.age* pueden estar muy separados.

Campo	Estructura	Agrupado /No Agrupado	Denso / Disperso
<E.age, E.sal>	B+ (rango)	NO Agrupado (NO tiene sentido agrupar)	Denso (única opción)

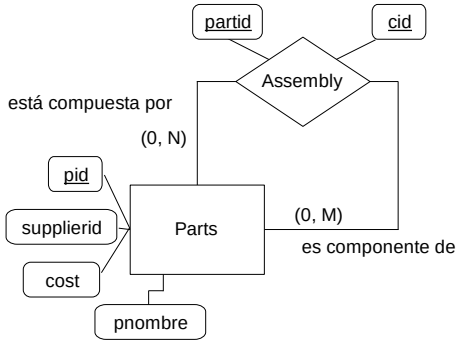
4.7. Agrupación (Clustering) de relaciones

Algunos SGBD (como ORACLE) permiten almacenar los registros de varias relaciones físicamente juntos, en la denominada **agrupación de relaciones** (***clustering***). Las agrupaciones sirven para guardar datos de distintas tablas en los mismos bloques de datos físicos. Se deben utilizar si con frecuencia se consultan de forma conjunta los registros de esas tablas.

Al estar almacenados dichos registros en los mismos bloques (páginas) de datos, se reduce el número de lecturas necesarias para llevar a cabo las consultas y, como consecuencia, se mejora el rendimiento. Sin embargo, estas agrupaciones ejercen un efecto negativo sobre las transacciones de manipulación de datos y sobre las consultas que solo hagan referencia a una de las tablas de la agrupación.

Cada agrupación guarda los datos de las tablas, además de mantener un **índice de clúster** que sirve para ordenar los datos. Las columnas del índice de clúster se denominan ***clave de clúster***; se trata del conjunto de columnas que tienen en común las tablas del clúster. Dado que las columnas de la clave de cluster son las que determinan la ubicación física de las filas en el clúster, estas columnas no deberían actualizarse con mucha frecuencia. La clave de cluster suele ser la clave foránea de una tabla que hace referencia a la clave primaria de otra tabla de cluster.

Veamos su interés bajo el siguiente ejemplo: Supongamos una entidad de piezas (PARTS) y una relación reflexiva ASSEMBLY que asocia cada pieza con sus componentes. Una pieza puede tener muchos componentes, y cada pieza puede ser componente de otras piezas. Su modelo E-R es de la forma:



Supongamos la siguiente consulta que permite conocer los componentes inmediatos de cada pieza:

```
SELECT P.pid, P.pnombre, A.cid
FROM PARTS P, ASSEMBLY A
WHERE P.pid = A.partid
ORDER BY P.pid
```

En este caso, se podría optar por crear índices agrupados por los campos *A.partid* y *P.pid* para obtener las tuplas ordenadas. Los índices serían de la forma:

Campo	Estructura	Agrupado /No Agrupado	Denso / Disperso
P.pid	B+ (ordenado)	Agrupado (las tuplas se obtienen ordenadamente por código de pieza)	Disperso (+ pequeño)
A.partid	Hash (igualdad)	Agrupado (No clave, ∃ duplicados)	Denso (única opción para índices hash)

Otra opción es crear un cluster; es decir, agrupar las dos tablas, almacenando cada registro de PARTS seguido por los registros de ASSEMBLY tales que *P.pid=A.partid*. Esta aproximación es mejor porque no necesita ningún índice.

Lo mismo sucede si la consulta permitiese localizar los componentes (inmediatos) de todas las piezas de un determinado suministrador *X*:

SELECT *P.pid, P.pnombre, A.cid*
FROM *PARTS P, ASSEMBLY A*
WHERE *P.pid=A.partid* AND *P.supplierid='X'*

En este caso, podrían crearse los siguientes índices:

Campo	Estructura	Agrupado /No Agrupado	Denso / Disperso
P.supplerid (si tuplas='X' ↓↓)	Hash (igualdad)	No Agrupado (↓↓ duplicados, ordenación cara)	Denso (única opción)
A.partid	Hash (igualdad)	Agrupado (No clave, ∃ duplicados)	Denso (única opción para índices hash)

De este modo podría aplicarse, en primer lugar, la condición de selección sobre PARTS y luego recuperar las tuplas de ASSEMBLY en base a un índice sobre *A.partid*.

Una opción alternativa sería hacer un join utilizando una organización *clustered*, que daría lugar a la recuperación del conjunto de tuplas de PARTS y ASSEMBLY (que se almacenan juntas). En este caso, el índice por A.partid no sería necesario, pero se mantendría P.supplerid.

La aproximación mediante *clustering* es especialmente interesante si quisiéramos movernos en diferentes niveles de la jerarquía pieza-componente. Por ejemplo, una consulta típica es conocer el coste total de una pieza, lo que requiere la necesidad de realizar joins entre *PARTS* y *ASSEMBLY* de forma repetitiva. Además, si no sabemos el número de niveles en la jerarquía *a priori*, ni siquiera se podría expresar esta consulta directamente en SQL dado que el número de joins varía.

En resumen, el *clustering*:

- puede acelerar los *joins*, en particular los establecidos entre una clave foránea y una principal correspondientes a relaciones 1:N
- la recuperación secuencial de las relaciones es más lenta
- las inserciones, borrados y actualizaciones que alteran las longitudes de los registros son más lentas debido a la sobrecarga producida por el *clustering*.

BIBLIOGRAFÍA:

- **Connolly, T.M.; Begg, C. Sistemas de bases de datos: un enfoque práctico para diseño, implementación y gestión.** Pearson Educación
- Date, C.J. Introducción a los sistemas de bases de datos. Prentice Hall
- Elmasri, R.; Navathe, S. Fundamentos de Sistemas de Bases de Datos. Addison-Wesley
- **Ramakrishnan, R.; Gehrke, J. Sistemas de Gestión de Bases de Datos**. McGraw-Hill
- Silberschatz, A.; Korth, H.; Sudarshan, S. Fundamentos de bases de datos. McGraw-Hill