Diseño físico de bases de datos

Jordi Conesa Caralt
M. Elena Rodríguez González

PID_00203528

UOC
Universitat Oberta de Catalunya

Índice

Introducción

Hasta ahora nos hemos centrado en cómo realizar el diseño conceptual y lógico de una base de datos para que satisfaga los requisitos de información del usuario, creando un esquema de base de datos que permita representar toda la información necesaria, de forma correcta y sin redundancias. Pero en cuanto pensamos en el sistema gestor de bases de datos (SGBD) donde se implementará dicha base de datos, en el hardware donde el SGBD estará instalado, en qué dispositivos de almacenamiento no volátil se guardan sus datos, en qué programas accederán a la base de datos y en el volumen de datos que deberá almacenar la misma, surge la pregunta de ¿cómo ajustar la base de datos para que funcione eficientemente en el entorno para el que se ha diseñado? El diseño físico de bases de datos es el proceso que se encarga de responder a esta pregunta.

El diseño físico forma parte del proceso general de diseño de una base de datos. En particular, se realiza después del diseño lógico de la misma. En el diseño lógico se genera un conjunto de tablas normalizadas a partir del esquema conceptual creado en la fase de diseño conceptual de la base de datos. El diseño físico empieza en el momento en que dichas tablas son creadas, y se centra en definir los métodos de almacenamiento y acceso a los datos que permitan operar con la base de datos con una eficiencia esperada. Una vez el diseño físico se ha realizado, el siguiente paso es implementar la base de datos y monitorizarla. Esta tarea de monitorización se encargará de comprobar si la base de datos satisface el rendimiento esperado. De no ser así, se deberán realizar modificaciones para mejorarlo. También puede ser necesario modificar el esquema para ajustar la base de datos a nuevos requerimientos. El responsable de realizar las tareas previamente descritas es el administrador de la base de datos, quien tiene a su disposición diferentes herramientas suministradas por los vendedores de SGBD.

Modelo lógico

Algunos fabricantes de SGBD utilizan el término *modelo lógico* para referirse al esquema conceptual y el término *modelo físico* para referirse a la implementación de las tablas normalizadas en el SGBD específico. En este caso, la creación de las tablas normalizadas a partir del esquema estaría dentro de la fase de diseño físico.

Tabla normalizada

Una tabla está normalizada cuando representa un concepto único del mundo real. Si el diseño conceptual de la base de datos es correcto, las tablas resultantes en el diseño lógico de la misma cumplen esta condición. Una tabla no normalizada padece redundancias, es decir, repeticiones de los datos que serían evitables.

En este módulo se introduce el diseño físico y sus características principales. Para ello se muestran los elementos que se utilizan en él para mejorar el rendimiento de la base de datos. Para cada uno de estos elementos se muestra, a modo de ejemplo, cómo gestionarlos en Oracle y PostgreSQL.

Objetivos

Estos materiales permiten que el estudiante adquiera las siguientes competencias:

1. Entender qué es el diseño físico de bases de datos, su objetivo, motivación y cómo llevarlo a cabo.

2. Conocer, de forma general, las distintas técnicas y elementos utilizados en el diseño físico de bases de datos y saber en qué casos tiene sentido aplicarlos.

3. Conocer la diferencia entre espacio virtual y físico de la base de datos y cómo gestionarlos.

4. Conocer los tipos de índices más habituales, cómo funcionan y en qué casos son necesarios.

5. Saber cómo funcionan los métodos de optimización de consultas utilizados en los SGBD relacionales y cómo utilizar índices para optimizar consultas.

6. Saber qué son las vistas materializadas, cuándo son necesarias y cómo gestionarlas.

1. Diseño físico de bases de datos

Conceptualmente, podemos definir el **diseño físico** de una base de datos como un proceso que, a partir de su diseño lógico y de información sobre su uso esperado, creará una configuración física de la base de datos adaptada al entorno donde se alojará y que permita el almacenamiento y la explotación de los datos de la base de datos con un rendimiento adecuado.

A continuación, vamos a estudiar en detalle esta definición para entender a fondo el concepto de diseño físico:

a) Entradas: el diseño físico parte de la siguiente información:

- **El esquema lógico**: es el resultado del diseño lógico de la base de datos y contiene la lista de tablas necesarias para almacenar la información relevante, incluyendo al menos, sus columnas, claves primarias y foráneas.

- **Información sobre el uso esperado de la base de datos**: estimación sobre los volúmenes de datos y transacciones que el sistema deberá procesar. También deberá contener una estimación sobre las operaciones SQL que se utilizarán, sobre qué datos, con qué frecuencia, y la calidad de servicio esperada.

b) Salida: se tomará un conjunto de decisiones sobre las estructuras físicas más adecuadas a utilizar, generando una configuración física de la base de datos. Esta configuración podrá estar compuesta por una combinación adecuada de los espacios de almacenaje, un conjunto de índices para mejorar el rendimiento de las consultas, un particionamiento de tablas adecuado, un conjunto de vistas materializadas y otros elementos adicionales, como por ejemplo disparadores que regulen reglas de negocio complejas o procedimientos almacenados en la base de datos.

c) Adaptado al entorno: el diseño físico puede verse influido por el SGBD donde se implemente la base de datos, los dispositivos de almacenamiento no volátil donde se guarden los datos de la base de datos, y el entorno hardware donde se aloje el SGBD.

- **SGBD**: el paso de un diseño lógico a físico requiere un conocimiento profundo del SGBD donde se vaya a implementar la base de datos. En particular, entre otros, se deberá tener un conocimiento de los siguientes elementos:
  - Soporte ofrecido a la integridad referencial
  - Tipos de índices disponibles
  - Tipos de datos disponibles
  - Tipos de restricciones de integridad disponibles
  - Parámetros de configuración que puedan afectar al rendimiento, como por ejemplo el tamaño de página y la gestión de datos y de concurrencia utilizados
  - Construcciones SQL disponibles de soporte al diseño físico
  - Particularidades del SGBD utilizado (y en consecuencia del lenguaje SQL) en relación a la definición de elementos relacionados con el diseño físico de la base de datos

- **Entorno de almacenamiento**: no es lo mismo almacenar la base de datos en un PC, que en un servidor con múltiples discos u otros dispositivos de almacenamiento no volátiles, que en un sistema distribuido. El hardware disponible y sus capacidades (velocidad de acceso, sistemas de replicación, etc.) permitirán definir distintas configuraciones físicas para mejorar el rendimiento de la base de datos.

Sentencias SQL

A diferencia de las sentencias SQL relacionadas con la manipulación de los datos y con el diseño lógico de las tablas, las sentencias SQL de soporte al diseño físico de la base de datos no aparecen en el estándar SQL. Por ese motivo, hay mucha variación entre las sentencias SQL utilizadas en los SGBD para gestionar los elementos relacionados con el diseño físico.

d) Que permita el almacenamiento y explotación de los datos con un rendimiento adecuado: el objetivo del diseño físico es obtener un buen rendimiento de la base de datos en un entorno real. Con *rendimiento* nos referimos básicamente al tiempo de respuesta a operaciones de consulta o actualización, a la carga de transacciones a procesar y a la disponibilidad de la base de datos.

1.1. Elementos de diseño físico

Los elementos ofrecidos por el SGBD para afinar el modelo físico de la base de datos son principalmente los siguientes:

- Espacios para tablas
- Índices
- Vistas materializadas
- Particiones

1.1.1. Espacios para tablas

Un **espacio para tablas**, *tablespace* en inglés, es un componente del SGBD que indica dónde se almacenarán los datos de la base de datos y en qué formato.

El espacio para tablas permite asociar un fichero físico a un conjunto de elementos de la base de datos (tablas, índices, etc.). Dicho fichero contendrá los datos correspondientes a los elementos de la base de datos asociados.

El número de espacios para tablas a crear en una base de datos dependerá de las necesidades del diseño físico. Los espacios para tablas podrán ser simples cuando afecten a un solo elemento de la base de datos, o compuestos cuando afecten a distintos elementos de la base de datos.

Además de indicar el fichero donde se almacenarán los datos, los espacios para tablas también permiten definir cómo será dicho fichero: espacio asignado inicialmente, espacio incremental cuando el espacio asignado se agota, memoria intermedia disponible, si se utilizarán técnicas de compresión al almacenar los datos, cómo gestionar el espacio libre del fichero, etc.

1.1.2. Índices

Los **índices** son estructuras de datos que permiten mejorar el tiempo de respuesta de las consultas sobre los datos de una tabla.

De manera intuitiva, se puede establecer un paralelismo entre los índices usados en una base de datos con los índices de conceptos clave que podemos encontrar en la mayoría de libros de texto. La idea es organizar (de manera alfabética en el caso del índice del libro) una serie de valores de interés (los conceptos clave elegidos en el caso del libro), conjuntamente con las páginas físicas (las páginas del libro) que contienen datos sobre el valor que está siendo indexado (en el caso del libro, se indican las páginas del libro que tratan sobre cada concepto clave que se decide indexar).

Los índices mantienen los valores de una o más columnas de una tabla en una estructura que permite el acceso a las filas de la tabla. Cada posible valor v tiene correspondencia con la dirección (o direcciones) física que contiene las filas de la tabla que tienen a v como valor de la columna indexada. Esta estructuración de los datos permite un acceso rápido a los datos cuando se realizan búsquedas por valor o cuando se requiere la ordenación de las filas de una tabla de acuerdo a los valores de la columna indexada.

En caso de no disponer de índices, cualquier consulta sobre una tabla de la base de datos requerirá, en el caso peor, un recorrido completo del contenido de la tabla.

Los índices se pueden utilizar también como sistema de control de las restricciones de integridad de unicidad (claves primarias y alternativas), definiendo un índice único sobre las columnas con dicha restricción. De manera similar, también sirven para garantizar las restricciones asociadas a las claves foráneas.

A pesar de que el uso de índices mejora el rendimiento de la consulta de datos de una base de datos, es necesario tener en cuenta que tienen asociado un coste de mantenimiento ante cambios en los datos. En consecuencia, es necesario estudiar cuidadosamente cuántos y qué índices crear.

Existen distintos tipos de índice, como por ejemplo, los índices basados en árboles B+, en mapas de *bits*, en técnicas de dispersión (*hash* en inglés), en árboles R, etc. El índice más utilizado es el basado en árboles B+, ya que funciona relativamente bien en todo tipo de situaciones. No obstante, no hay un índice universal que funcione bien en todos los casos. En función de cada caso se deberá escoger entre un tipo de índice u otro, y con una configuración adecuada.

1.1.3. Vistas materializadas

Los resultados de las consultas sobre una o más tablas se pueden almacenar en **vistas materializadas**. A diferencia de las vistas convencionales, el conjunto de filas que conforma el contenido de una vista materializada se almacena físicamente en la base de datos, como las filas que conforman el contenido de una tabla. Este tipo de vistas pueden ser muy útiles para mejorar el rendimiento de consultas que se ejecutan repetidamente o de consultas basadas en datos agregados. En ambos casos, el uso de una vista materializada permite calcular la información a priori, ahorrándonos así calcular el contenido asociado a la vista (o a parte de ella) cada vez que el usuario lo solicite.

El tiempo de respuesta de la base de datos puede disminuir significativamente cuando se implementan las vistas materializadas adecuadas. No obstante, en su definición es necesario tener en cuenta sus costes: requiere espacio extra para almacenar el contenido asociado a las vistas materializadas y requiere mantener actualizado su contenido. Este último hecho, por ejemplo, desaconseja el uso de las vistas materializadas en casos donde los datos de origen tengan una frecuencia de actualización alta.

1.1.4. Particiones

Las **particiones** permiten distribuir los datos de una tabla en distintos espacios físicos. Las particiones se pueden hacer de acuerdo a multitud de criterios: distribuyendo las filas de la tabla en función de los valores de una columna, o de un conjunto de ellas (fragmentación horizontal), distribuyendo los datos

de las filas de una tabla entre distintos espacios en función de las columnas a las que pertenecen (fragmentación vertical), e incluso agrupando datos con características comunes, como es el caso de la distribución de los datos agrupados por dimensiones en los *data warehouse*.

Un buen diseño de las particiones permite reducir la carga de trabajo de los componentes hardware del sistema. Los motivos son que el particionamiento de tablas facilita que diferentes transacciones se ejecuten concurrentemente, incrementando así el rendimiento del SGBD y que las consultas a tablas de gran tamaño se puedan resolver mediante consultas más simples que se ejecutan de forma concurrente. Adicionalmente, en el caso de que la base de datos esté distribuida, el particionamiento permite acercar y adecuar los datos a las necesidades de los usuarios/aplicaciones, fomentando el paralelismo y disminuyendo el tráfico de datos a través de la red de comunicaciones. También permite distribuir los datos en dispositivos de almacenamiento más o menos rápidos según la importancia de los datos.

1.1.5. Otros métodos

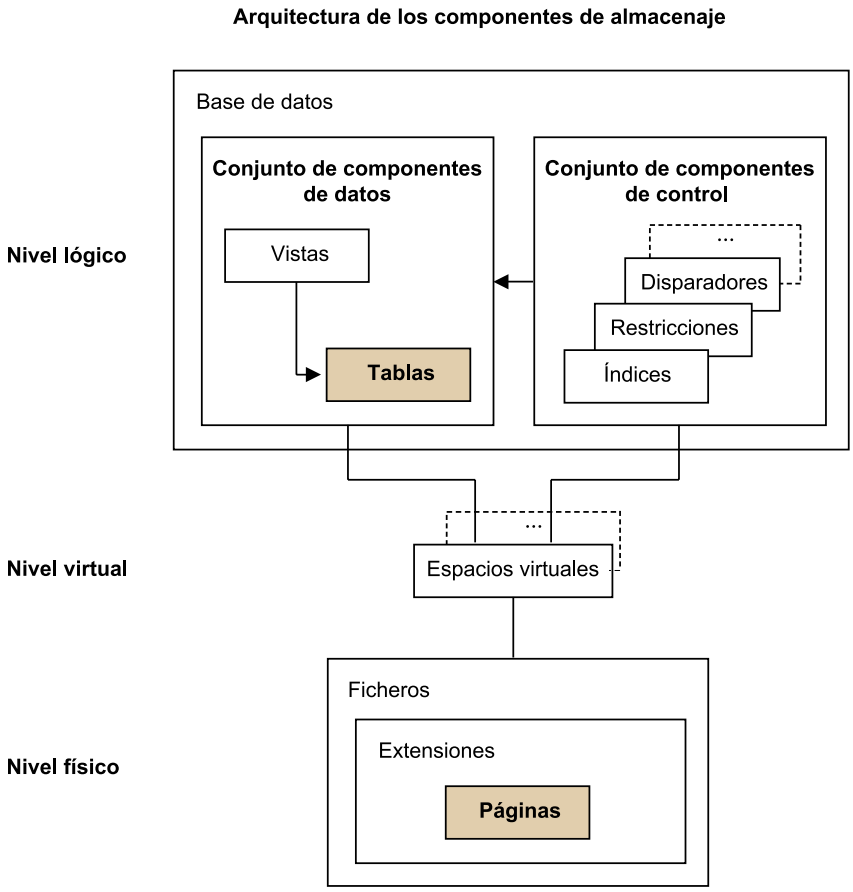
Hay también otros métodos para hacer el acceso a los datos más eficiente. Además, es importante mantenerse debidamente actualizado respecto a las nuevas versiones de SGBD, ya que cada nueva versión puede añadir más elementos para realizar un buen diseño físico y un posterior ajuste de la base de datos.

Otro método que puede ser utilizado en el diseño físico es la compresión de datos, que permite reducir el tamaño requerido para almacenar los datos y, en consecuencia, mejorar el tiempo de respuesta al recuperarlos. Al utilizarlo, se debe tener en cuenta que su uso implica una pequeña sobrecarga al descomprimir los datos cuando estos se recuperan y al comprimirlos cuando se actualizan. Otro de los métodos utilizados es el *data striping*, que permite distribuir datos que deben consultarse de forma atómica en distintos dispositivos de almacenamiento no volátil para permitir un nivel más elevado de paralelismo y de balanceo de carga. Normalmente, este método permite incrementar el número de transacciones por segundo. Otra forma de mejorar la disponibilidad de la base de datos es el *mirroring*, que permite reproducir (o replicar) la base de datos. Dicha reproducción obliga a que se mantengan las copias actualizadas y necesita más espacio de almacenaje. Por ello, el *mirroring* es conveniente cuando disponemos de datos que se actualizan muy raramente.

1.2. Componentes de almacenaje de una base de datos

Antes de poder abordar el diseño físico de una base de datos es importante conocer cómo almacena y gestiona los datos un SGBD. La siguiente figura muestra, de forma simplificada, la arquitectura de almacenaje que utiliza.

Figura 1. Arquitectura de los componentes de almacenaje de un SGBD



Tal y como se puede ver en la figura, el almacenaje de datos persistentes en una base de datos se realiza en una arquitectura de tres niveles:

- **Nivel lógico**: son los elementos de la base de datos con los que trabaja el usuario: tablas, vistas, índices, restricciones de integridad, etc.

- **Nivel físico**: son las estructuras de datos utilizadas para almacenar los datos de la base de datos en dispositivos no volátiles. Este nivel incluirá los ficheros, las extensiones y las páginas.

- **Nivel virtual**: es un nivel intermedio entre los niveles físico y lógico que proporciona al SGBD una visión simplificada del nivel físico. Esto facilita el diseño físico de la base de datos.

Desde un punto de vista teórico, en el modelo relacional, los datos se almacenan en **relaciones**, que normalmente corresponden a conceptos del dominio de interés. Cada relación contiene **tuplas**, que se corresponden a las instancias (u ocurrencias) de un concepto. Las tuplas están compuestas por **atributos**, que son los que permiten representar las características (o propiedades) de los conceptos. Al implementar el modelo anterior en un SGBD relacional, las relaciones se representan mediante **tablas**, que contienen distintas **filas**, que corresponden a las tuplas de la relación. Cada tabla tiene un conjunto

de **columnas**, que representan a los atributos del concepto a representar. Al implementar las tablas en **ficheros**, las filas se representan mediante **registros** y las columnas mediante **campos**. La siguiente tabla muestra un resumen de esta nomenclatura y la relación entre términos. A lo largo del módulo usaremos principalmente los términos correspondientes a las dos últimas filas.

Tabla 1. Nomenclatura

Nivel	Concepto	Instancia	Característica
Modelo relacional	Relación	Tupla	Atributo
SQL	Tabla	Fila	Columna
Físico	Fichero	Registro	Campo

Ya conocéis los componentes del nivel lógico. A continuación introduciremos los componentes de los niveles virtual y físico, a los que conjuntamente llamaremos componentes de almacenaje, porque controlan la disposición física de los datos en el soporte del almacenamiento no volátil (en general, disco).

1.2.1. Nivel físico

Los datos se almacenan en soportes no volátiles controlados por el sistema operativo (SO) de la máquina donde se alojan. Es el SO el encargado de efectuar las operaciones de lectura y escritura física de los datos. Los SGBD aprovechan las rutinas existentes de los SO para leer y escribir los datos de las bases de datos.

Los SO gestionan los datos a partir de unas estructuras globales llamadas **ficheros**. Normalmente el SO no reserva una gran cantidad de espacio destinada a satisfacer todas las necesidades futuras de almacenamiento en el dispositivo de almacenamiento no volátil, sino que realiza una asignación inicial y va adquiriendo más espacio a medida que lo necesita. La unidad de adquisición se denomina **extensión**. Una extensión es una agrupación de páginas consecutivas en el dispositivo de almacenamiento no volátil. A su vez, las páginas son el componente físico más pequeño. Las **páginas** son los elementos que finalmente contienen y almacenan los datos del nivel lógico.

La página es el componente más importante del nivel físico ya que es la unidad mínima de acceso y de transporte del sistema de entrada y salida (E/S) de un SGBD. Eso quiere decir que:

- La página es la unidad mínima de transferencia entre la memoria externa (no volátil) y la memoria interna (o memoria principal).

- En los SGBD el espacio en el dispositivo de almacenamiento no volátil siempre se asigna en un número múltiplo de páginas.

- Cada página puede ser direccionada individualmente.

El hecho de que la página sea la unidad mínima de transferencia tiene implicaciones importantes en el diseño físico. Cuando trabajamos con un SGBD, es importante saber con qué tamaño de página trabaja, para conocer cuántas filas de una tabla o entradas de un índice cabrán en cada página. Hacer una buena gestión de las páginas de una base de datos puede ahorrar muchos accesos de E/S innecesarios, incrementando enormemente el rendimiento. Veremos algunos ejemplos de ello a lo largo de este módulo.

Las páginas pueden clasificarse, principalmente, en páginas de datos y páginas de índice. Las primeras contienen datos (entre otros) de tablas o vistas materializadas, mientras que las segundas contienen datos sobre los índices. Cada tipo de página requerirá una estructura interna de página diferente.

La extensión es una agrupación de páginas consecutivas que el SO proporciona para almacenar datos. Cuando el SGBD detecta que se necesita más espacio, el SO otorga una nueva extensión. Las diferentes extensiones de un mismo fichero no tienen por qué estar consecutivas en el dispositivo de almacenamiento no volátil. La adquisición de nuevas extensiones es automática y transparente para los usuarios de la base de datos.

Un fichero está formado por un conjunto de extensiones y es la unidad lógica que usan los SO para gestionar el espacio en los dispositivos de almacenamiento no volátil.

1.2.2. Nivel virtual

En este punto nos podríamos preguntar: ¿es realmente necesario el nivel virtual en una base de datos? ¿Qué proporciona realmente este nivel?

En una primera aproximación podríamos pensar que cada tabla se almacena en un fichero, y que cada fichero solo almacena datos de una tabla. Es decir, podríamos pensar que siempre existe una relación biunívoca entre tabla y fichero, con lo cual desaparecería la necesidad de un nivel intermedio que haga corresponder componentes lógicos con componentes físicos, ya que dicha correspondencia sería fija.

La realidad, sin embargo, es más compleja que la suposición anterior. A continuación presentamos algunos casos:

a) Puede haber tablas muy grandes que nos puede interesar particionar, de tal manera que cada partición se almacene en un fichero diferente (quizás localizado en un dispositivo de almacenamiento diferente) para mejorar el acceso.

b) Por el contrario, podemos encontrar un conjunto de tablas muy pequeñas, que convenga guardar en un mismo fichero con el objetivo de no consumir tantos recursos del sistema.

c) En ocasiones, una tabla puede estar tan relacionada con otra u otras que en la mayoría de los casos el acceso a la primera, por parte de los usuarios o de las aplicaciones, comporta también un acceso a la segunda, o a las demás. En esos casos, nos interesará que los datos de estas tablas se encuentren tan cerca físicamente como sea posible para minimizar el tiempo de acceso global y aumentar el rendimiento de la gestión de sus datos. En definitiva, tras esta opción, la idea es tener las operaciones de combinación (en inglés, *join*) entre las tablas relacionadas preconstruidas en el dispositivo de almacenamiento no volátil.

d) Se pueden usar tablas que, además de contener columnas con datos de tipo tradicional (cantidades numéricas, cadenas de caracteres, fechas, etc.), tienen otras que almacenan tipos de datos diferentes, como gráficos, imágenes o algunos minutos de audio o de vídeo. Estos datos, que necesitan muchos más *bytes* para ser almacenados, se denominan objetos grandes (en inglés *Large OBjects*, *LOB*). Estos objetos grandes se almacenan en ficheros diferentes a los que se usan para los datos de tipo más tradicional, con el fin de mejorar los tiempos de acceso.

e) Aunque solo hemos mencionado las tablas, existen otros componentes, como por ejemplo, los índices. Los ficheros que guardan información sobre los índices, de forma similar al caso anterior, están estructurados internamente de manera diferente para poder obtener el máximo rendimiento.

Todos estos ejemplos deben servirnos para ser conscientes de la utilidad de disponer del nivel virtual. Nos proporciona un grado elevado de flexibilidad (o independencia) para asignar componentes lógicos a los componentes físicos, ya que no siempre se desea establecer una correspondencia biunívoca entre tabla y fichero. Concretamente, nos permite decidir dónde se almacenará cada tabla o partición de tabla. De esta manera escogemos en qué máquina, en qué dispositivo de almacenamiento no volátil o en qué trozo del mismo tendremos los diferentes datos de la base de datos. El nivel virtual es normalmente conocido bajo las denominaciones de *tablespace* o *dbspace* en los SGBD comerciales.

2. Espacio para tablas

Los espacios para tablas (*tablespace* en inglés) permiten definir dónde se almacenarán los datos de los objetos de la base de datos. Una vez creado, un *tablespace* puede referenciarse por su nombre cuando se crea un nuevo objeto en la base de datos.

El uso de los *tablespaces* permite un mayor control sobre cómo se distribuirán los datos de la base de datos en los dispositivos de almacenamiento no volátil disponibles (por lo general, discos). Facilitan en gran medida la redistribución de los datos de la base de datos en caso de necesidad, como por ejemplo, cuando un *tablespace* se ubica en un disco que está a punto de llenarse. En este caso, se podría fácilmente cambiar la ubicación de los objetos del *tablespace* simplemente asignando una nueva ubicación física al mismo. Aparte, los *tablespaces* nos permiten distribuir fácilmente los distintos objetos de la base de datos en distintos discos para mejorar el rendimiento. Así pues, podríamos ubicar los datos de un índice que se utilice frecuentemente en un disco de alta velocidad (como por ejemplo en un disco de estado sólido) para acelerar su acceso. Por otro lado, podríamos ubicar las tablas que no se usen muy frecuentemente en un disco más lento.

Tal y como hemos comentado, los *tablespaces* pueden asignarse a cualquier objeto de la base de datos. En caso de asignarse a tablas, vistas materializadas e índices, indica el espacio físico donde se almacenarán los datos de dichos objetos. En caso de asignarse a una base de datos, se puede utilizar para distintos fines: para almacenar los datos del catálogo de la base de datos, para almacenar los datos temporales de la base de datos, como *tablespace* por defecto, etc. La necesidad de almacenar datos temporales por parte del SGBD puede acontecer, por ejemplo, en la resolución de las consultas formuladas por los usuarios y también en la gestión de vistas. Dichos datos temporales se guardan en los denominados espacios de tablas temporales (en inglés, *temporary tablaspaces*).

Al no estar incluidos en el estándar SQL, la definición y uso de cada *tablespace* es distinto en cada SGBD. Por lo tanto, en cada caso se deberá comprobar la semántica y sintaxis concreta en la documentación de cada SGBD. A continuación veremos cómo definirlos y utilizarlos en Oracle y PostgreSQL.

2.1. Definición y uso de espacio de tablas en Oracle

A continuación mostramos una versión simplificada de la sintaxis de creación de *tablespaces* en Oracle:

Notación

En la notación a utilizar a partir de ahora se utilizarán:

- Palabras en mayúsculas para denotar cláusulas SQL
- Palabras en minúsculas para denotar nombres de variables que deben ser informados por el usuario.
- Corchetes [] para indicar opcionalidad (cláusulas que pueden no informarse).
- Llaves {} para indicar un conjunto de cláusulas separadas por "|" de las cuales solo se puede elegir una.

```
CREATE TABLESPACE tablespace_name
   DATAFILE 'filename' [SIZE nn]
   [AUTOEXTEND {OFF|ON [NEXT integer] [MAXSIZE {UNLIMITED|integer}]]
   [BLOCKSIZE integer]
   DEFAULT [{COMPRESS|NOCOMPRESS}]
   ...
```

Oracle permite una gran capacidad de personalización al crear los espacios de tablas. En particular, los elementos que permite definir son:

- `Tablespace_name`: nombre del *tablespace*. Es el nombre que se utilizará para identificar el *tablespace* y asignarlo a los distintos objetos de la base de datos.
- `Filename`: permite definir el fichero donde se almacenará la información asociada al *tablespace* y el tamaño del mismo.
- `Autoextend`: indica qué hacer cuando el fichero asignado al *tablespace* se llene. Se puede no hacer nada (`OFF`), extender el fichero para que permita nuevos datos (`ON NEXT nn`, donde `nn` indica el tamaño de la extensión). También se puede indicar el tamaño (`size`) máximo del fichero.
- `Blocksize`: permite definir el tamaño de página que utilizará el *tablespace* para almacenar la información.
- `Compress/nocompress`: indica si los datos se deben almacenar comprimidos, o no.

Acrónimos de tamaños

Cuando se indican tamaños se debe indicar la unidad de tamaño utilizando un acrónimo. Las unidades que se pueden utilizar y sus acrónimos son: K para indicar *kilobytes*, M para indicar *megabytes*, G para indicar *gigabytes* y T para indicar *terabytes*.

Definición de tamaño de página

En Oracle el tamaño de página se define al crear la base de datos. No obstante, ese tamaño de página por defecto puede cambiarse para tener más flexibilidad al trabajar con *tablespaces*.

Por ejemplo, la siguiente sentencia crearía un *tablespace* para almacenar los elementos de la base de datos que no requieran de un acceso muy rápido. Como podemos ver, se indica la ubicación del fichero, el nombre del *tablespace* (`tbs_slow_access`), el tamaño inicial del fichero (20 *megabytes*) y que el sistema debe extender el fichero en incrementos de 10 *megabytes* cuando este se llene hasta llegar a un máximo de 200 *megabytes*.

```
CREATE TABLESPACE tbs_slow_access
   DATAFILE '/dev/slowdisks/disc1/tbs_slow_access.dat'
```

```
       SIZE 20M
       AUTOEXTEND ON NEXT 10M MAXSIZE 200M;
```

Una vez creado el *tablespace*, si quisiéramos asignar una tabla al mismo, simplemente tendríamos que indicarlo en su definición. Así, por ejemplo, la siguiente sentencia asigna la tabla ciudad (`City`) al *tablespace* denominado `tbs_slow_access`.

```
CREATE TABLE City (
   cityZip  CHAR(5),
   cityName VARCHAR(50)
) TABLESPACE tbs_slow_access;
```

2.2. Definición y uso de espacio de tablas en PostgreSQL

La definición de *tablespaces* en PostgreSQL es más simple y permite menos personalización.

```
CREATE TABLESPACE tablespacename [OWNER username] LOCATION 'directory'
```

Como podemos ver, en este caso solo se permite definir el nombre del *tablespace*, su propietario y su ubicación. Muchos de los parámetros que hemos visto en el *tablespace* de Oracle también pueden configurarse en PostgreSQL, pero usando sentencias separadas. Por ejemplo, el tamaño de página puede modificarse al instalar (o recompilar) el SGBD mediante el parámetro `with-blocksize`.

Si tuviéramos que crear un *tablespace* como el de la sección anterior y asignarlo a la tabla ciudad (`City`) lo haríamos de la siguiente manera:

```
CREATE TABLESPACE tbs_slow_access
  LOCATION '/dev/slowdisks/disc1/tbs_slow_access.dat';

CREATE TABLE City (
   cityZip  CHAR(5),
   cityName VARCHAR(50)
) TABLESPACE tbs_slow_access;
```

3. Índices

La gran mayoría de operaciones sobre una base de datos se realizan por valor. A su vez, el acceso por valor puede ser directo o secuencial. Más concretamente:

- El **acceso directo por valor** consiste en obtener todas las filas que contienen un determinado valor para una columna.

- El **acceso secuencial por valor** consiste en obtener diversas filas por el orden de los valores de una columna.

A continuación se muestran ejemplos sobre una tabla de clientes que contiene las siguientes columnas: código del cliente (clave primaria), nombre del cliente, población y edad.

a) Ejemplos de acceso directo por valor:

```
SELECT * FROM clientes WHERE poblacion='Arenys de Mar'

UPDATE cliente SET edad=50 WHERE codigo_cliente=20

DELETE FROM cliente WHERE codigo_cliente=100
```

b) Ejemplos de acceso secuencial por valor:

```
SELECT * FROM cliente ORDER BY edad

SELECT * FROM cliente WHERE edad>=40 AND edad<=50

DELETE FROM cliente WHERE poblacion IN ('Barcelona','Tarragona')
```

Los ejemplos previos muestran accesos por valor (secuenciales o directos) sobre una única columna. Pero también se puede acceder por el valor que muestran diversas columnas. Estos accesos se denominan **accesos por diversos valores**, y pueden ser directos, secuenciales y mixtos (en el caso de que una misma operación o sentencia combine a la vez accesos por valor directo y secuencial). A continuación se muestran algunos ejemplos sobre la tabla de clientes:

a) Acceso directo por diversos valores:

```
SELECT * FROM cliente WHERE poblacion='Arenys de Mar' AND edad=30

UPDATE cliente SET edad=edad+1 WHERE codigo_cliente='Barcelona' AND edad=40
```

b) Acceso secuencial por diversos valores:

```
SELECT * FROM clientes ORDER BY edad, poblacion

SELECT * FROM clientes WHERE edad>30 AND codigo_cliente<100
```

c) Acceso mixto por diversos valores:

```
SELECT * FROM cliente WHERE ciudad='Arenys de Mar' ORDER BY edad

DELETE FROM cliente WHERE edad<25 AND ciudad='Barcelona'
```

Por defecto, y a falta de estructuras de datos que den soporte a las operaciones mostradas en los ejemplos, cualquier operación sobre una tabla de la base de datos significará hacer un recorrido de toda la tabla y preguntar, para cada fila recuperada, si cumple o no las condiciones de búsqueda. Si este sistema ya puede resultar muy costoso por sí mismo, la situación se agrava en función del número de filas que contiene la tabla (por ejemplo, podrían ser decenas de miles), o si en la petición se incluyen combinaciones de tablas (operaciones de *join*) y/o agrupaciones (consultas con cláusula `GROUP BY`). En todos estos casos, la eficiencia puede caer en picado.

Para dar solución a esta problemática se han creado los **índices**, que son estructuras de datos que permiten mejorar el tiempo de respuesta de las peticiones que impliquen un acceso por valor.

El concepto de índice no es nuevo, sino que son estructuras centenarias que forman parte de nuestra vida cotidiana. Ejemplos de índices fuera del ámbito de las bases de datos serían el índice de capítulos de un libro, el índice de conceptos clave de un libro mencionado al inicio de este módulo y el mapa de sitio de una web.

Cabe destacar que los SGBD crean automáticamente algunos índices para gestionar las restricciones de integridad de la base de datos. Entre los índices creados por defecto por los SGBD encontramos los creados sobre la clave primaria y las claves alternativas.

En su versión más simple, un índice permite acceder a las filas de una tabla a partir de los valores de una de sus columnas. La columna sobre la cual se construye el índice se denomina columna indexada. Cada valor distinto *v* de la columna indexada se hace corresponder con el identificador de registro (RID[1]) que apunta a la fila de la tabla que tiene a *v* como valor de la columna indexada. Las diferentes parejas (*v*, *RID*) que se almacenan en el índice reciben el nombre de entradas del índice. Esta estructuración de los datos permite un acceso rápido a los mismos cuando se realizan accesos directos o secuenciales por valor, o se requiere la ordenación de las filas de una tabla de acuerdo a los valores de la columna indexada. El índice descrito en este párrafo responde a

(1)El RID (o *Record Identifier* en inglés) es un identificador que apunta al registro físico donde se almacena la fila de una tabla.

uno de los tipos de índices más simples: el índice único, donde la columna indexada es una clave candidata, es decir, su valor identifica unívocamente cada fila de la tabla con independencia de que esta sea clave primaria o alternativa de la tabla.

A partir de aquí los índices pueden ir sofisticándose según el tipo y número de columnas indexadas y el tipo de estructura de datos utilizada en su implementación. A continuación describimos los distintos tipos de índice en función del tipo y número de columnas indexadas y en las siguientes secciones veremos algunas de las estructuras de datos más utilizadas para crear índices en las bases de datos.

En algunos casos interesará indexar una tabla por una columna que no tome valores únicos, como, por ejemplo, la población de los clientes, que puede repetir valor en diferentes filas de la tabla. Estos casos implican modificar ligeramente la estructura del índice, por ejemplo, haciendo que la entrada del índice asociada a cada posible valor v de la columna indexada tenga asociados N *RID* ($\{RID_1, ..., RID_N\}$, $N>0$), donde cada RID_i apuntará hacia una fila de la tabla que tiene el valor v en la columna indexada.

También nos podremos encontrar con la necesidad de definir índices sobre múltiples columnas: por ejemplo, un índice sobre las columnas nombre y apellidos de nuestra tabla de clientes. Dichos índices podrán ser únicos o no únicos. En este caso, cada entrada del índice estará representada por una combinación de los valores indexados y apuntará al (los) RID de las filas de la tabla cuyas columnas contienen el valor indicado en la entrada del índice. A la hora de definir el índice, el orden de las columnas puede ser relevante, es decir, un índice definido sobre las columnas nombre y apellidos, puede ser diferente a un índice definido sobre las columnas apellidos y nombre, de tal manera que cada uno de estos índices ayudará a resolver de forma eficiente diferentes tipos de consulta.

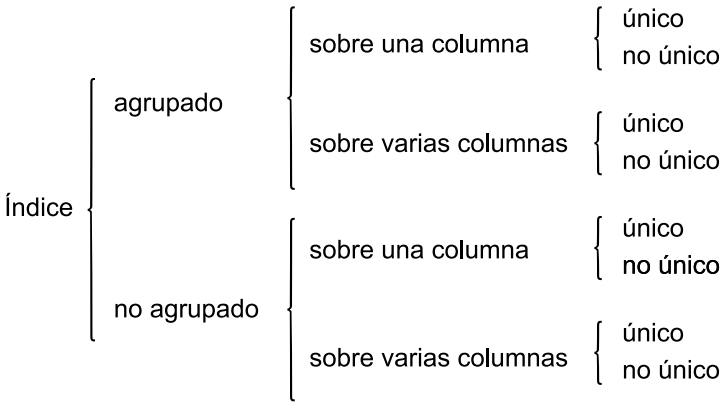
Los índices comentados hasta ahora se denominan índices ordenados porque las columnas indexadas (en general, alfanuméricas) permiten la ordenación de sus valores, hecho que facilita la estructuración del índice. No obstante los datos a los que apuntan, es decir las filas de la tabla, no tienen por qué estar ordenadas. Una alternativa a dichos índices sería almacenar las filas de la tabla de forma ordenada de acuerdo a las columnas indexadas. Esto es lo que se conoce con el nombre de índice agrupado (o *clustered index* en inglés). Ordenar las filas de una tabla puede mejorar significativamente el tiempo de respuesta en la resolución de peticiones que implican un acceso secuencial por valor, ya que pueden minimizar el número de operaciones de E/S. Notad que solo es posible tener un índice agrupado por tabla.

Número de operaciones de E/S

El número de operaciones de E/S es un factor que se debe reducir y permite mejorar significativamente el rendimiento del procesamiento de las consultas.

Tal y como se muestra en la siguiente figura, los índices explicados hasta este momento se pueden clasificar en función de si los datos se guardan de forma ordenada en el fichero que los contienen, del número de columnas que se indexan y de la unicidad de valores de las mismas.

Figura 2. Clasificación de índices



Aparte de los índices previamente presentados, tenemos otros tipos de índices que nos permiten indexar información no textual, como pueden ser los índices geográficos. Estos permiten consultar elementos geográficos dentro de mapas basándose en su posición, y facilitan el uso de funciones, como la distancia entre puntos o la intersección de figuras geométricas.

Existen distintas estructuras de datos que pueden utilizarse para crear índices en bases de datos, como por ejemplo los árboles B+, las funciones de dispersión (en inglés *hash*), los mapas de *bits* (en inglés *bitmap*) los índices parametrizables (GiST y GIN), los árboles R, los Quadtree, etc. En la siguiente sección veremos las estructuras de datos más utilizadas para almacenar y gestionar índices de bases de datos. Luego veremos cómo crear estos índices en Oracle y PostgreSQL. A continuación introduciremos brevemente los conceptos básicos asociados a optimización de consultas, y acabaremos con una serie de recomendaciones que ayuden al lector a saber qué índices se deben crear en cada caso concreto.

3.1. Tipos de índices más habituales

La estructura más utilizada en los índices de bases de datos son los árboles B+, porque se comportan bien en la mayoría de situaciones. No obstante, no hay un tipo de índice universal que funcione bien en todos los casos. Otros tipos de índices, como por ejemplo los basados en funciones de dispersión o en mapas de *bits*, pueden ser más adecuados en ciertas circunstancias. A continuación vamos a explicar qué son y cómo funcionan los índices basados en árboles B+, en funciones de dispersión y en mapas de *bits*. Asimismo, cuando hablemos de PostgreSQL, se introducirán un par de tipos de índices más particulares de dicho SGBD, los índices GiST y GIN.

3.1.1. Árboles B+

La estructura de árbol B+ es la más utilizada para implementar índices en bases de datos relacionales. De hecho, de todos los tipos de índices que hemos presentado, los árboles B+ (tal como los explicaremos o variantes similares) son los únicos que están disponibles en todos los SGBD relacionales.

Los índices basados en árboles B+ son índices ordenados. Por lo tanto, tal y como se indica en la clasificación de la figura 2, permiten indexar una o múltiples columnas, con independencia de si contienen valores únicos o no y de forma ordenada (árbol B+ agrupado) o no ordenada (árbol B+ no agrupado). Estos índices pueden ayudar a resolver eficientemente cualquier acceso por valor, ya sea directo o secuencial. También son especialmente útiles para devolver resultados de forma ordenada, como por ejemplo, en las operaciones SQL que contienen la cláusula `ORDER BY`.

Los árboles B+ son un tipo particular de árbol de búsqueda cuyo objetivo primordial es minimizar las operaciones de E/S en las búsquedas.

Un árbol B+ se compone de nodos que están interrelacionados entre ellos. Las relaciones entre nodos son dirigidas y relacionan un nodo padre (el origen de la relación) con un nodo hijo (el destino de la relación). De hecho, se trata de relaciones de orden entre los valores indexados, que se encuentran distribuidos entre los distintos nodos del árbol. En un árbol no se permiten ciclos, es decir, un nodo no puede estar interrelacionado consigo mismo, ni directa ni indirectamente. Cada nodo del árbol, excepto un nodo especial llamado raíz, tiene un nodo padre y diversos (cero o más) nodos hijo. El nodo raíz no tiene padre, los nodos que no tienen hijos se llaman nodos hoja y los nodos que no son hojas se llaman nodos internos.

Un árbol B+ tiene asociadas las siguientes características:

- **Los nodos hoja y los nodos no hoja** (nodo raíz y nodos internos) **tienen estructuras diferentes.** Los nodos no hoja contienen valores para dirigir la búsqueda hacia los nodos hoja y apuntadores hacia nodos hijo (que pueden ser nodos internos o nodos hoja). Por su parte, los nodos hoja contienen las entradas del índice (es decir, las parejas valor y RID). Además, cada nodo hoja puede contener hasta dos apuntadores más. Un apuntador al siguiente nodo hoja y un apuntador al nodo hoja anterior. Estos apuntadores tienen como objetivo facilitar la resolución de accesos secuenciales (parciales o completos) por valor.

- **Un árbol B+ tiene un número de orden *d* que indica la capacidad de sus nodos y**, en consecuencia, indica también **el número de nodos hijo que puede tener cada nodo interno**. Los nodos tienen como máximo *2d* valores o entradas, dependiendo de si se trata de un nodo interno u hoja, respectivamente. Adicionalmente, los árboles B+ también imponen una

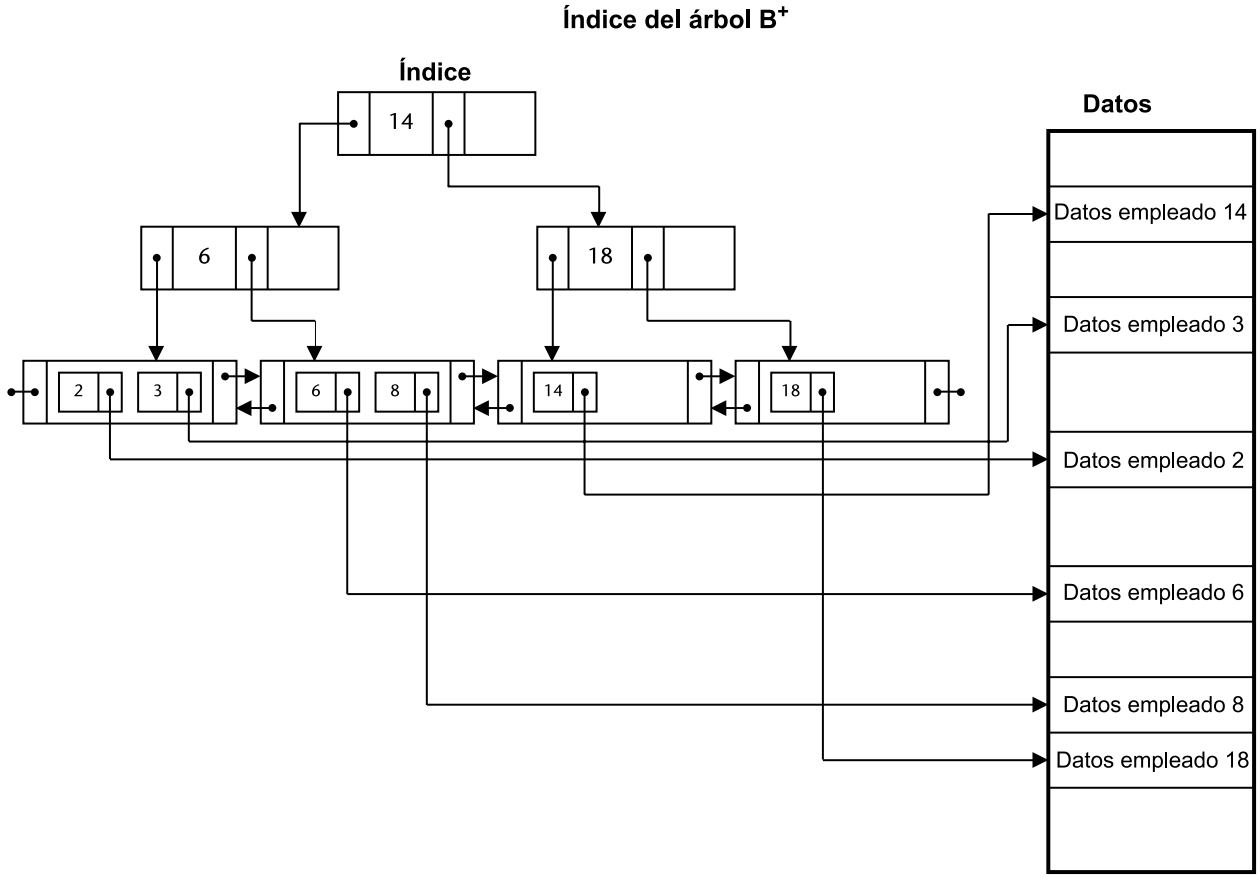
ocupación mínima para los nodos. Esta ocupación mínima es del 50%. Existe un nodo, en concreto el nodo raíz, que está exento de cumplir esta condición. Por lo tanto, en un árbol B+ de orden *d*, todos los nodos internos (a excepción de la raíz) tienen una ocupación mínima de *d* valores (y un mínimo de *d+1* apuntadores a nodos hijo) y una ocupación máxima de *2d* valores (y un número máximo de *2d+1* apuntadores a nodos hijo). En el caso de nodos hoja, tendrán entre *d* y *2d* entradas (y un máximo de 2 apuntadores). La condición de ocupación mínima causa que un árbol B+ sea equilibrado.

- Todos **los valores de un nodo interno deben estar presentes en algún nodo hoja y no pueden estar repetidos**.

- **Los nodos hoja deben contener todas las entradas**. Es decir, todos los valores que en un momento dado existen de la columna (o columnas) sobre la cual se construye el índice, tienen su correspondiente entrada en un nodo hoja.

La siguiente figura muestra un ejemplo de índice de árbol B+ definido sobre una columna de tipo numérico:

Figura 3. Ejemplo de árbol B+ para indexar una columna numérica



Estructura de un árbol B+

La figura muestra un índice árbol B+ de orden *d*=1 que sirve para acceder a datos de empleados según el valor del atributo `IDEmpleado`. Fijaos que los RID que apuntan a los datos solo se encuentran en los nodos hoja, que los nodos internos sirven para buscar los valores de las hojas y que el hecho de que las hojas estén conectadas entre sí facilita el acceso secuencial por valor.

Suponed que la figura anterior se corresponde con un índice único definido sobre la clave primaria (`IDEmpleado`) de una tabla que guarda datos sobre empleados. Suponed también que un usuario plantea la siguiente consulta:

```
SELECT *
   FROM EMPLEADO
   WHERE IDEmpleado > 16;
```

En este caso el SGBD detecta que se está filtrando los valores de la consulta en función de la columna `IDEmpleado` para la cual existe un índice en forma de árbol B+. Supongamos que en este punto el SGBD decide usar el índice. En tal caso, el SGBD cargaría en memoria el nodo raíz del árbol B+. A continuación consultaría el único valor que contiene (14) y lo compararía con el valor a buscar (16). Como 14 es menor que 16, se debe seguir el apuntador de la derecha del valor para acceder al siguiente nodo (recordad que el objetivo de los nodos intermedios es conducir la búsqueda hacia los nodos hoja). Después de cargarse en memoria, se consultaría el valor que contiene dicho nodo, que es 18. Como 18 es mayor que 16, se seguirá el apuntador de la izquierda del valor para acceder al nodo hoja. Este nodo hoja deberá contener valores inferiores a 18. Al llegar al nodo hoja se lee la única entrada que contiene. Dicha entrada tiene el valor de 14, que al ser inferior de 16 no es de interés para la consulta. Como el nodo hoja no contiene más entradas, para devolver los valores con identificador de empleado superior a 16 se sigue el apuntador al siguiente nodo hoja del árbol. En dicho nodo se encuentra una nueva entrada del índice con el valor de 18. Como 18 es mayor que 16 el sistema utiliza el RID de la entrada para acceder a los datos del empleado y devolverlos en la consulta. Al no haber más entradas en el nodo hoja y no haber tampoco otros nodos hoja a continuación, la consulta acaba y se retornan los datos del empleado con identificador de empleado 18.

Los árboles B+ son útiles para:

- **Realizar accesos directos por valor**: primero hay que localizar la hoja que tiene la entrada del valor buscado, y después, utilizar el RID de la entrada para encontrar los datos a los que se quiere acceder.

- **Realizar accesos secuenciales por valor**: primero hay que localizar la primera entrada de la secuencia y utilizar su RID para devolver el primer resultado. Posteriormente se obtendrán los siguientes valores de forma ordenada y sus RID navegando por los nodos hoja del índice.

Las explicaciones previas asumen que el árbol B+ es no agrupado. En el caso que fuese agrupado, podemos ganar eficiencia, especialmente en los accesos secuenciales por valor. En este caso, una vez se ha localizado la primera entrada de interés en un nodo hoja del árbol B+, siguiendo su RID podemos acceder al primer resultado en el fichero que guarda las filas que están siendo indexadas. Como dicho fichero está ordenado físicamente según el valor que toma la columna (o columnas) sobre la que se ha construido el árbol B+, es suficiente realizar la lectura de las páginas de dicho fichero y, por lo tanto, no es necesario consultar los nodos hoja restantes del árbol B+.

De forma similar, en las explicaciones, la tabla sobre la cual se ha construido el árbol B+ se encuentra en un único fichero que solo almacena datos de esa tabla. Algunos SGBD (por ejemplo, este sería el caso de Oracle y PostgreSQL) permiten crear índices particionados sobre tablas que han sido, a su vez, fragmentadas de forma horizontal. En este caso, cada fragmento de la tabla se encuentra almacenado en un fichero diferente que solo almacena datos de esa tabla, y cada uno de estos fragmentos dispone de su propio índice en forma de árbol B+, que únicamente indexa los datos contenidos en el fragmento. Para finalizar, otro tipo de índice que puede ser útil en el diseño físico es el basado en funciones. Este tipo de índice no indexa los valores de una columna (o un conjunto de ellas), sino los valores de una función (o la composición de un conjunto de ellas) que se aplica sobre una o más columnas. Esto puede ser muy útil cuando se hace un uso intenso de una determinada función sobre las mismas columnas de una tabla.

Coste de localización de una entrada

Una práctica habitual al crear árboles B+ es hacer coincidir los nodos del árbol con el tamaño de la página. Así, el número de accesos de E/S necesarios para localizar cualquier entrada del árbol es h, siendo h la altura del árbol.

Como la altura del árbol es directamente proporcional al número de accesos necesarios para encontrar una entrada, es importante crear árboles que tengan una altura tan pequeña como sea posible. Por lo tanto, es necesario que los nodos del árbol sean grandes pero sin sobrepasar el tamaño de una página, porque, de lo contrario, no se podrían consultar con una única operación de E/S. Adicionalmente, nos interesa que los nodos estén tan llenos como sea posible, es decir, nos interesa que el orden d del árbol sea tan grande como sea posible. La idea no es solo que los nodos sean grandes, sino también que estén llenos, es decir, que tengan un índice de ocupación elevado.

Ejemplo: Cálculo del orden de un árbol B+ para ajustar los nodos a las páginas de datos

Suponed que nuestro SGBD trabaja con páginas de 8 Kb (como en el caso de PostgreSQL) y que queremos indexar la columna DNI de una tabla que guarda información de personas. La columna DNI ocupa 8 *bytes* y es la clave primaria de la tabla personas. Consideramos que el tamaño del RID y de los apuntadores a los nodos del árbol es de 4 y 3 *bytes*, respectivamente.

Según estos datos, el cálculo del orden *d* del árbol B+ que permite una ocupación máxima de los nodos, se efectúa de la siguiente manera:

- **Nodo hoja**: en un nodo hoja caben un máximo de *2d* entradas y hasta dos apuntadores a nodos hoja. En nuestro ejemplo, para almacenar cada entrada necesitamos 12 *bytes* (8+4). Y para almacenar los dos apuntadores 6 *bytes* (2*3). Cada nodo hoja tiene un tamaño de 8192 *bytes* (8 Kb). En consecuencia:
  $2d*12 + 6 = 8192 \Rightarrow d= 341{,}0833 \Rightarrow d= 341$, dado que *d* es número entero.

- **Nodo no hoja**: en un nodo no hoja caben un máximo de *2d* valores y *2d+1* apuntadores a nodos hijo. Cada nodo no hoja tiene un tamaño de 8192 *bytes* (8 Kb). En consecuencia:
  $2d*8 + (2d+1)*3 = 8192 \Rightarrow d= 372{,}2272 \Rightarrow d= 372$, dado que *d* es número entero.

El orden *d* de un árbol B+ es único y debe satisfacer las necesidades de almacenamiento de los nodos hoja y no hoja. Por lo tanto, en el caso de nuestro ejemplo, el orden máximo *d* del árbol B+ es 341.

Por otro lado, dado el orden del árbol B+ y las propiedades comentadas anteriormente, es fácil calcular cuántas filas de datos se pueden indexar en un índice en forma de árbol B+ en función de la altura del mismo. A continuación presentamos un ejemplo que indica el número de filas que puede indexar el árbol B+ anterior con alturas 1, 2 y 3, respectivamente.

Ejemplo: cálculo del número de filas que se pueden llegar a indexar en función de la altura del árbol B+

Continuando con el árbol B+ anterior, cuyo orden *d* es 341, la cuestión es cuántas filas diferentes de la tabla de personas podemos llegar a indexar como máximo. Esto dependerá de la altura (*h*) del árbol B+.

- **Si h=1**, el árbol B+ tiene un único nodo (que es a la vez nodo raíz y hoja) que contiene 341*2 entradas, es decir, permite indexar 682 personas diferentes.

- **Si h=2**, el árbol B+ tiene un nodo raíz que contiene 341*2 valores y 341*2+1 apuntadores a nodos hijo, que a la vez, son nodos hoja. Por lo tanto, tenemos 683 nodos hoja, cada uno con 682 entradas. En consecuencia, tendremos un total de 683*682 entradas, es decir, el árbol B+ permite indexar 465.806 personas diferentes.

- **Si h=3**, el árbol B+ tiene un nodo raíz que contiene 682 valores y 683 apuntadores a nodos internos. Tenemos, pues, 683 nodos internos. Cada nodo interno, a su vez, tiene 682 valores y 683 apuntadores a nodos hoja. Por lo tanto, tenemos un total de 466.489 (683^2) nodos hoja. El número total de entradas será 466.489*682, es decir, el árbol B+ permite indexar 318.145.498 personas diferentes (notad que este número ya excedería el total de habitantes de España).

La altura de un índice en forma de árbol B+ condiciona el número de operaciones de E/S necesarias para dar respuesta a los accesos por valor. En particular, en una consulta de acceso directo por valor, solo es necesario consultar *h* nodos, siendo *h* la altura del árbol, para identificar si hay alguna fila de una tabla que satisface los criterios de búsqueda. En caso de que sea así, consultar los datos de la fila requerirá de una nueva operación de E/S para cargar los datos de la fila en memoria. A continuación se muestra mediante un ejemplo el número de operaciones de E/S necesarias para resolver una consulta SQL en el índice anteriormente creado.

Ejemplo: Cálculo de operaciones necesarias de E/S para responder a una consulta de acceso directo por valor utilizando un árbol B+

Imaginemos que nuestro árbol B+ (visto en los dos ejemplos anteriores) tiene altura *h* igual a 3, orden *d* de 341 y que los nodos presentan ocupación máxima. Supongamos que queremos resolver la siguiente consulta:

```
SELECT *
   FROM personas
   WHERE DNI='46742377';
```

¿Cuántas operaciones de E/S son necesarias para localizar la persona con DNI 46742377?

Suponiendo que en la tabla de personas existe una persona con el DNI indicado, serían necesarias 4 operaciones de E/S, 3 en el índice, para localizar el nodo hoja que contiene la entrada que nos interesa, más 1 operación de E/S adicional en el fichero de datos para recuperar los datos de la persona deseada. Si no existe ninguna persona con el DNI necesario serían necesarias 3 operaciones de E/S (las del índice). Imaginemos el número de operaciones de E/S para localizar esa persona entre un total de 318.145.498 personas si no tuviésemos el árbol B+. O incluso peor, imaginemos qué pasaría si la persona con el DNI deseado (46742377) no existiese en la base de datos.

Coste de mantenimiento

Los árboles B+ son árboles equilibrados. Este tipo de árboles tienen características muy interesantes que facilitan la búsqueda de elementos, pero mantener los árboles equilibrados tiene un coste asociado.

Como ya hemos visto, los árboles B+ deben satisfacer la restricción de que todos los nodos (excepto el nodo raíz) deben contener como mínimo *d* valores o entradas dependiendo del tipo de nodo, siendo *d* el orden del árbol. Este hecho requiere que la modificación de los valores de la columna indexada y la inserción y eliminación de filas de la tabla indexada implique reestructurar el índice a menudo. Por ello en tablas y/o columnas con una frecuencia de actualización muy alta el uso de este tipo de índice puede tener un coste de mantenimiento muy elevado.

3.1.2. Tablas de dispersión

Otro tipo de estructura de datos que se utiliza en la creación de índices es la de dispersión (o *hash* en inglés). Este tipo de estructuras permite un acceso directo por valor a los datos muy eficiente, incluso más eficiente que usando árboles B+ en muchos casos. No obstante, este tipo de índices no dan soporte al acceso secuencial por valor, es decir, no resulta de utilidad en búsquedas que utilicen operadores distintos a la igualdad (>, >=, <, <=, <>).

La filosofía de estos índices es la misma que hay detrás de las tablas de dispersión utilizadas para almacenar datos en memoria interna. Aunque en nuestro caso el objetivo es utilizar funciones de dispersión para minimizar el número de operaciones de E/S.

En los índices basados en dispersión, existe una función de dispersión (normalmente llamada *h*) que identifica la página de datos donde se ubicarán las filas de la tabla indexada. La función de dispersión *h* es una función que tie-

ne por dominio el conjunto de los posibles valores de la columna indexada y como rang, las referencias de las páginas disponibles en la base de datos para la tabla indexada: $h(v) \rightarrow p$.

Para cada fila de la tabla indexada la función de dispersión calcula, a partir del valor de su columna indexada, la página de datos donde se deberá almacenar la fila.

Como el valor de la función de dispersión establece la ubicación de los datos de la tabla, en principio solo es posible utilizar un índice de dispersión por tabla. No obstante, en algunos casos, se utilizan tablas auxiliares para permitir una indirección y así soportar el uso de más de un índice de dispersión por tabla, a cambio de incrementar en una unidad el número de operaciones de E/S necesarias en la recuperación de los datos.

Debido a sus características, no tiene sentido que los índices de dispersión sean agrupados, ya que la función de dispersión devuelve directamente la página donde almacenar los datos y no garantiza que datos con valores cercanos tengan valores de h próximos. Hay otras características que complicarían también su agrupación, como la gestión de sinónimos que veremos más adelante.

Normalmente, el número de valores posibles que devuelve la función de dispersión debe ser menor que el número de valores posibles de la columna indexada. Eso se debe a dos hechos:

1) El número de posibles valores puede ser muy superior al número de páginas disponibles.

2) La mayoría de los posibles valores no se utilizan.

Suponed por ejemplo un identificador de empleado de 32 *bits*. Los posibles valores de dicho identificador es superior a 4.000 millones (concretamente es igual a 4.294.967.296=2^{32}). En caso de tener una función de dispersión que devuelve el mismo número de valores que la columna que indexa, necesitaríamos 4.294.967.296 páginas para almacenar la tabla indexada. Si el tamaño de página de nuestro SGBD fuera de 8 Kb se requeriría 32 *terabytes* (2^{32}*****8 *kilobytes*) para almacenar la tabla. Además, cada página solo contendría una fila, con lo cual se estaría malgastando espacio. Como puede comprobarse, esta estrategia de almacenamiento no es precisamente eficiente.

Por todo ello, las funciones de dispersión acostumbran a reducir el número de valores posibles y por lo tanto no son inyectivas sino exhaustivas: es decir, dos valores diferentes (v_i y v_j tal que $v_i \neq v_j$) pueden tener el mismo valor de h: $h(v_i)=h(v_j)$. Cuando esto pasa, los valores v_i y v_j se denominan sinónimos.

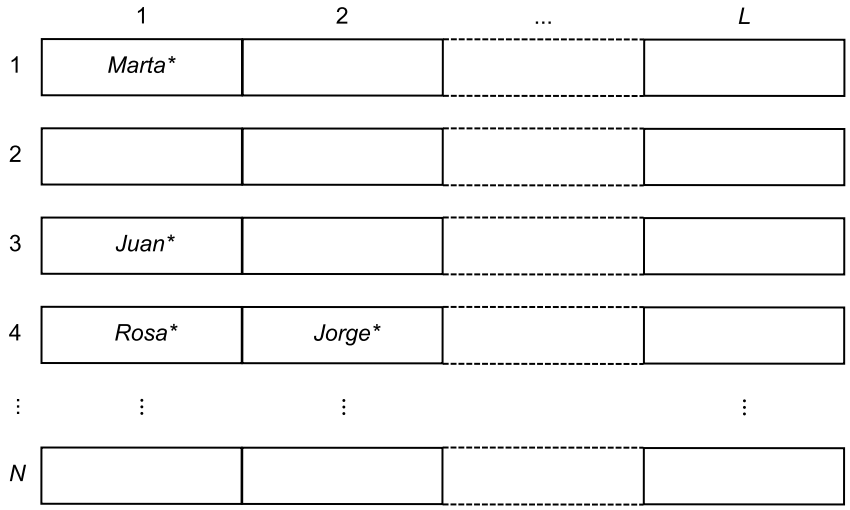
Las filas de los valores sinónimos se almacenan en la misma página. Cuando la página está llena se crea una nueva página, llamada página de excedentes, donde se almacenarán las filas de los nuevos sinónimos y se crea un apuntador en la página original que apunta a la nueva página de excedentes. En caso de que la página de excedentes se llene, se encadenará una nueva página de excedentes, y así sucesivamente. Una situación similar a la descrita también puede acontecer cuando el índice de dispersión se construye sobre una columna que puede tomar valores repetidos. La página inicial que debería contener los datos de interés una vez aplicada la función de dispersión se denomina página primaria, en contraposición a las páginas de excedentes.

Tal y como el lector habrá intuido, la habilidad de la función *h* para evitar valores sinónimos afectará significativamente al número de operaciones de E/S requeridas para la recuperación de los datos de interés.

Ejemplo: Localización de una fila de datos utilizando índices de dispersión

Supongamos que queremos indexar una tabla de empleados con un índice de dispersión según la columna nombre de pila. Supongamos también que *h* tiene los siguientes valores para los siguientes nombres de pila (*Juan* → *3*, *Marta* → *1*, *Rosa* → *4*, *Jorge* → *4*). La siguiente figura nos muestra la distribución de los datos de la tabla en las páginas de datos de la base de datos. En particular, *N* es el número de páginas primarias que tenemos disponibles (que viene determinado por el número de posibles valores de la función de dispersión) y *L* es el número de filas que caben en página. *Nombre** representa los datos completos de la fila de la tabla con valor *Nombre*. Por lo tanto, por ejemplo, en la cuarta página se encuentran los datos de las filas que corresponden a los nombres de *Rosa* y *Jorge*.

Figura 4. Distribución de los datos de una tabla siguiendo un índice de dispersión



Supongamos que el usuario realiza la siguiente consulta:

```
SELECT *
   FROM EMPLEADO
   WHERE nombre = 'Jorge';
```

El SGBD identificaría que la columna *nombre* está indexada según el índice anteriormente mostrado y calcularía *h('Jorge')*. El resultado de dicho cálculo sería el número de página donde se encuentran los datos de la fila (la página 4) o, en el peor de los casos, la página desde donde acceder a sus datos. El sistema cargaría en memoria la cuarta página, consultaría la primera fila de la misma, que equivale a un sinónimo (*Rosa*). Posteriormente consultaría la segunda fila de la página para darse cuenta de que es la que estaba buscan-

do. En ese momento podrían retornar los datos y finalizar con la consulta. El número de operaciones de E/S para resolver la consulta sería 1.

Coste de localización de una entrada

Cuando se utiliza un índice de dispersión, el coste de localizar una fila varía bastante en función de si esta se encuentra en una página de excedentes o no. Si una entrada no se encuentra en una página de excedentes solo habrá que hacer una operación de E/S para obtenerla. En caso contrario habrá que hacer distintas operaciones de E/S hasta encontrar la página de excedentes que contenga la fila buscada.

En consecuencia, para conseguir un buen rendimiento del índice interesa que haya pocas páginas de excedentes. Esto se puede conseguir de dos formas distintas:

- Escoger una función de dispersión que tenga en cuenta la distribución de valores de la columna (o columnas) indexada para minimizar el número de sinónimos.

- Diseñar el índice para reducir el número de páginas de excedentes.

Existe numerosa bibliografía que trata el tema de cómo escoger funciones de dispersión adecuadas para cada caso. Tratar este tema queda fuera de los objetivos de este material, principalmente por la dificultad de definir criterios que sirvan para la mayoría de casos.

Para ver cómo podemos conseguir reducir el número de páginas de excedentes, introduciremos el concepto de factor de carga. Se llama factor de carga al número de entradas que se espera tener, dividido por el número de entradas que caben en las páginas primarias:

$$C = M / (N \times L)$$

M representa el número de entradas que esperamos tener, *N* el número de páginas primarias (o número de posibles valores de la función de dispersión) y *L* es el número de filas que caben en cada página. El valor de *L* es sencillo de calcular ya que se hace dividiendo el tamaño de página por el tamaño de cada fila de la tabla indexada. Idealmente también se debería conocer el valor de *M*, si no exactamente, al menos sí de forma aproximada.

El factor de carga es un elemento de diseño que permite ajustar los valores de *N* (modificando la función de dispersión) o de *L* (cambiando el tamaño de la página de datos) para reducir el número de páginas de excedentes necesarias.

Un factor de carga bajo indica que habrá menos excedentes, minimizando el número de operaciones de E/S necesarias para obtener los datos. En contrapartida, requerirá de más páginas primarias. Por otro lado, un factor de carga

elevado permitirá reducir el número de páginas primarias, pero aumentará el número de páginas de excedentes. Esto puede causar que se requieran más operaciones de E/S para resolver las consultas.

Los índices de dispersión que hemos explicado, que requieren fijar a priori el número *N* de páginas primarias, se conocen también bajo la denominación de índices de dispersión estática. Como alternativa existen los índices de dispersión dinámica. En estos índices no es necesario que el número *N* de páginas primarias esté fijado a priori. En otras palabras, el número de páginas primarias crece en función de las necesidades, a medida que se añaden más filas en la tabla sobre la que se ha construido el índice. Entre los índices de dispersión dinámica más conocidos están el *Extendible hashing* y el *Linear hashing*. Este último está disponible, por ejemplo, en el SGBD Berkeley DB. Asimismo, este tipo de índices, inicialmente diseñados para el ámbito de las bases de datos, también han sido transferidos a lenguajes de programación, como sería el caso de Python que dispone de *Extendible hashing*.

Coste de mantenimiento

En el caso general, las inserciones, modificaciones y las eliminaciones implicará insertar o eliminar filas en las páginas primarias o en las de excedentes. El número de operaciones de E/S dependerán del factor de carga.

Las eliminaciones de filas se pueden hacer de forma lógica (solo marcando la fila afectada como borrada) o de forma física. La eliminación física de una fila puede implicar mover filas de las páginas de excedentes a las páginas primarias, reduciendo la longitud de las cadenas de excedentes.

Las operaciones de modificación sobre una columna indexada implicará mover las filas afectadas a páginas de memoria distintas y, en algunos casos, reestructurar la página primaria con filas de las páginas de excedentes. En este tipo de operaciones el mantenimiento puede llegar a ser más alto que en el caso de los índices basados en árboles B+, debido a que deben moverse las filas de datos de página. En los índices en forma de árbol B+ no agrupados se puede realizar la actualización simplemente modificando la información del índice.

3.1.3. Mapa de *bits*

Otro tipo de índices útiles para indexar columnas con pocos valores posibles que se repiten a menudo son los índices de mapas de *bits* (o índices *bitmap* en inglés). La principal ventaja de este tipo de índice es el poco tamaño que ocupa cuando se usa bajo las condiciones adecuadas, permitiendo incluso tener todo el índice cargado en memoria y evitar operaciones de E/S para consultarlo. Este tipo de índice permite identificar qué filas cumplen una determinada condición utilizando solo operaciones de *bits*. Como las operaciones sobre se-

cuencias de *bits* son mucho más rápidas que sus alternativas sobre otros tipos de datos, este índice permite identificar si una fila cumple las condiciones especificadas más rápidamente.

En un índice de mapa de *bits* se crea una secuencia de *bits* para cada fila de la tabla que indica el valor de la columna indexada. La manera de crear dicha secuencia puede variar, pero en estos materiales explicaremos su forma más simple: tendrá tantos *bits* como posibles valores puede tener la columna indexada. El *bit i*-ésimo de la secuencia de *bits* relativa a una fila tendrá un 1 en caso de que la fila tenga el valor correspondiente a la posición *i* y un 0 en caso contrario. El valor del índice para toda una tabla estará formado por una matriz de *N* filas y *M* columnas, siendo *N* el número de filas y *M* los posibles valores de la columna indexada. Normalmente se denomina mapa de *bits* a la secuencia de *bits* relacionada con el posible valor de un índice (las columnas de la matriz), que indica qué filas de la tabla tienen dicho valor.

Por ejemplo, supongamos que tenemos una tabla de empleados como la que se muestra en la figura 5, con las columnas DNI, nombre, vehículo propio y zona geográfica. Supongamos también que la columna vehículo propio admite valores nulos o bien los valores SÍ/NO, y la columna zona geográfica es obligatoria y admite solo 4 valores (Barcelona, Girona, Lleida, y Tarragona). Como las columnas vehículo propio y zona geográfica tienen pocos valores posibles, podemos considerar adecuado crear dos índices de mapa de *bits*, uno para cada columna.

Figura 5. Ejemplo de representación de dos índices de tipo de mapa de *bits* (columnas vehículo propio y zona)

Empleado			
DNI	**Nombre**	**Vehículo propio**	**Zona**
88775997	José	SÍ	Barcelona
46781222	María	SÍ	Lleida
E-998272	Jordi	NULL	Girona
87271923	Neus	SÍ	Barcelona
88928187	Elena	No	Tarragona

Vehículo propio	
Sí	**No**
1	0
1	0
0	0
1	0
0	1

Zona geográfica			
Barcelona	**Girona**	**Lleida**	**Tarragona**
1	0	0	0
0	0	1	0
0	1	0	0
1	0	0	0
0	0	0	1

En la figura anterior se puede ver gráficamente la creación de dichos índices. Habrá dos mapas de *bits* del índice sobre la columna vehículo, uno para cada posible valor. Cada mapa de *bits* tendrá 5 *bits*, uno por cada fila de la tabla. El lector podría pensar que se podría utilizar un único mapa de *bits*. No obstante, en este caso es obligatorio utilizar dos mapas de *bits*, ya que la columna admite valores nulos y, por lo tanto, que una fila no tenga un valor cierto no implica necesariamente que sea falso (tal y como muestra la tercera fila de nuestro ejemplo). En la figura también podemos ver el índice de mapa de *bits* resul-

tante de indexar la columna zona. Notad que en este caso hay 4 mapas de *bits* que se corresponden con los posibles valores de la columna zona geográfica. Cada mapa de *bits* consta de 5 *bits*.

Una de las ventajas de los índices de mapas de *bits* es el escaso tamaño que ocupan cuando el número de valores diferentes que toma la columna indexada son pocos. En el ejemplo anterior necesitaríamos 10 *bits* (2 valores x 5 filas) para almacenar el primer índice y 20 *bits* (4 valores x 5 filas) para almacenar el segundo. Además, los mapas de *bits* se pueden comprimir y sus ratios de compresión son muy elevadas debido a la distribución homogénea de sus valores.

Este tipo de índice es aplicable solo cuando se realizan operaciones de igualdad y desigualdad (=, <>), inclusión (`IN`, `NOT IN`) o lógicas (`AND`, `OR`, `NOT`). En consecuencia, no es adecuado cuando se realizan operaciones de comparación sobre las columnas indexadas (<, <=, >, >=).

Ejemplo: Resolución de una consulta utilizando índices de mapas de *bits*

Supongamos que el usuario desea consultar los empleados que son responsables de la zona de Barcelona o Lleida y que tienen vehículo propio.

```
SELECT *
   FROM EMPLEADO
   WHERE vehiculo = 'SI' AND zona IN ('Barcelona','Lleida');
```

El SGBD al recibir esta consulta comprobaría que las columnas utilizadas en la cláusula `WHERE` están indexadas por índices de mapas de *bits* (vehículo propio y zona). Suponiendo que el SGBD decidiera utilizar estos índices, los pasos que podría realizar para identificar las filas a devolver serían los siguientes:

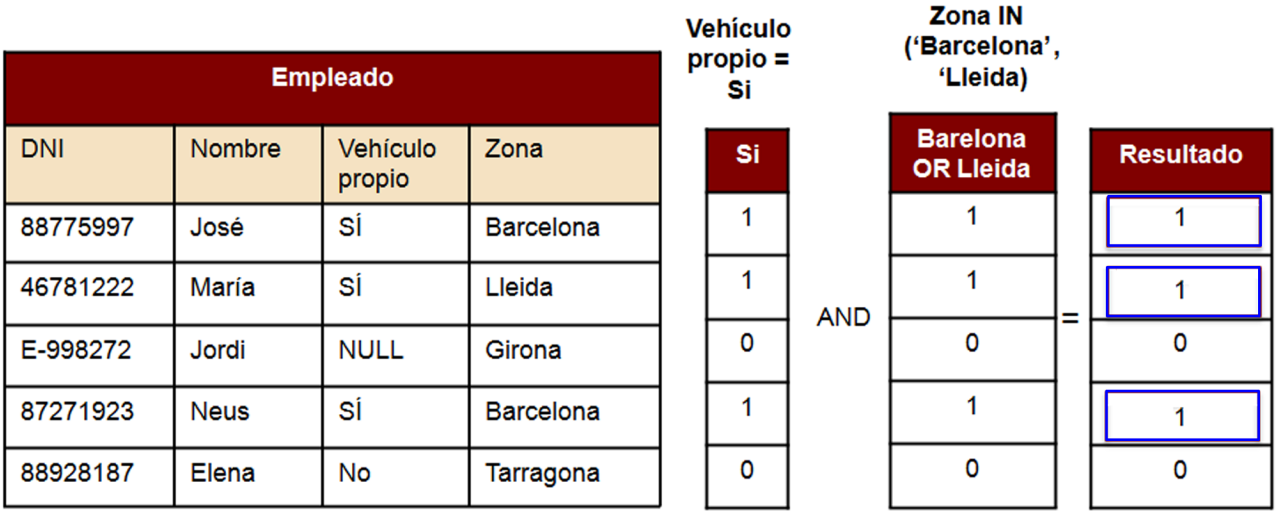
Para identificar los empleados con vehículo propio se debería obtener las filas que tienen un 1 en la primera posición del mapa de *bits*, es decir, las filas con un 1 en el mapa de *bits* del valor *Sí*. El resultado incluye la primera, la segunda y la cuarta fila.

Para identificar a los empleados de las zonas de Barcelona y Lleida, se deben identificar las filas con un 1 en el primer o el tercer mapa de *bits*. Esto se puede hacer usando una operación `OR` lógica entre los mapas de *bits* de los valores Barcelona y Lleida. El resultado incluiría las filas 1, 2 y 4.

Finalmente, se identifican las filas que han satisfecho ambas condiciones: las filas 1, 2 y 4, que se podrían calcular realizando una operación `AND` lógica entre los dos mapas de *bits* resultantes de los pasos 1 y 2.

Podemos ver las operaciones realizadas y el resultado gráficamente en la siguiente figura:

Figura 6. Ejemplo de cómo resolver una consulta utilizando índices de mapas de *bits*



Coste de localización de una entrada

El acceso a los índices de mapa de *bits* es normalmente muy rápido cuando el número de posibles valores de la columna indexada es limitado. A medida que el número de valores posibles de la columna indexada aumenta, el tamaño del mapa de *bits*, y en consecuencia el tiempo de respuesta, se incrementa exponencialmente.

Coste de mantenimiento

Calcular en qué casos es aconsejable un índice de tipo mapa de *bits* puede ser una tarea complicada. Para tomar una decisión con rigor, se debería tener en cuenta, entre otros, el número de filas de la tabla, el número de columnas a indexar, y el número de posibles valores para cada columna. Esta información debería usarse para estimar el tamaño del índice. Normalmente, y para simplificar, se tiende a evitar este tipo de índices cuando el número de valores posibles para la columna a indexar está por encima de los 100, ya que su rendimiento decrece muy rápidamente y su tamaño se incrementa de forma exponencial.

Además, este tipo de índice solo es recomendable para tablas estáticas, es decir, para tablas con pocas actualizaciones. El motivo es que cuando se realiza una modificación sobre una columna indexada, se bloquea todo el índice de mapa de *bits*. Dicho bloqueo impide que se pueda utilizar el índice de nuevo hasta que no se resuelva la transacción que originó el bloqueo. Por lo tanto, si en una tabla de un millón de filas modificáramos el valor de una fila, bloquearíamos el índice entero, provocando que no se pudiera utilizar para acceder a ningún valor de la tabla y ralentizando enormemente otras operaciones que pudieran estar ejecutándose concurrentemente en la base de datos.

Sin embargo, este tipo de índices es muy apropiado para *data warehouses* que se actualizan pocas veces y en horarios donde no hay actividad, como por ejemplo por la noche. Si la tabla sobre la que queremos crear el índice es accedida frecuentemente por múltiples usuarios, entonces debe evitarse su uso.

3.2. Definición de índices en sistemas gestores de bases de datos

Los índices son elementos del diseño físico de la base de datos y como consecuencia no están cubiertos en el estándar SQL. No obstante, la sentencia `CREATE INDEX` está presente en todos los SGBD, aunque con parámetros distintos en función del SGBD con el que trabajemos.

A continuación vamos a ver de forma esquemática cómo crear índices en Oracle y PostgreSQL.

3.2.1. Definición de índices en Oracle

Los índices utilizados por Oracle son los basados en árboles B+ y los de mapas de *bits*. Los índices en forma de árbol B+ son los utilizados por defecto. Oracle también incluye otros tipos de índice en algunas de sus extensiones, por ejemplo, la extensión de Oracle para tratar datos espaciales (*Oracle Spatial*), incluye índices basados en Quadtrees y árboles R.

Lecturas adicionales

Información sobre los *Index-organized Tables*
http://docs.oracle.com/cd/B28359_01/server.111/b28318/schema.htm#i23877

Cómo simular un índice *clustered*
http://www.dba-oracle.com/data_warehouse/clustered_index.htm

Oracle no permite crear índices agrupados (o *clustered*) o índices de dispersión directamente, pero permite simularlos utilizando otras estructuras de datos. Por ejemplo, las tablas organizadas por índices (*Index-Organized Tables* en inglés) se pueden utilizar para simular índices *clustered*.

A continuación mostramos de forma simplificada la sintaxis de la sentencia para crear índices en Oracle:

```
CREATE [{UNIQUE|BITMAP}] INDEX index_name ON
   table (index_expr [{ASC | DESC}] [, index_expr [{ASC|DESC}] ]...)
   [ { TABLESPACE { tablespace | DEFAULT }
      | key_compression
      | ...
      }, ...
   ]
```

Algunas de las opciones a definir en la creación de un índice son:

- `UNIQUE`: se puede indicar si el índice se realiza sobre una columna que no permite repetidos. Si no se utiliza la cláusula `UNIQUE` el índice permitirá valores duplicados.

- `BITMAP`: permite crear índices de tipo mapa de *bits*. Es importante notar que no se permite definir este tipo de índices sobre columnas que toman

valores únicos. En caso de no informar esta cláusula, el índice creado será de tipo árbol B+.

- `Table (index expr [ASC|DESC])...`: indica las columnas indexadas y si el índice guarda los valores de forma ascendente o descendente.

- `TABLESPACE`: indica el espacio de tablas a utilizar para almacenar el índice.

- `Key compression`: permite indicar si se quiere comprimir los valores a indexar y cómo se debe realizar la compresión. Esta opción no es necesaria en los índices de mapas de *bits* porque siempre se comprimen en Oracle. Por ello esta cláusula se orienta a la compresión de las entradas de los índices en forma de árbol B+.

Por ejemplo, supongamos una tabla de ciudades como la que sigue:

```
CREATE TABLE City (
   cityZip       CHAR(5) NOT NULL,
   cityName      VARCHAR(50),
   cityRegion    CHAR(2),
   cityhabitants NUMBER(10)
) TABLESPACE tbs_slow_access;
```

Las sentencias que se muestran a continuación permiten crear tres índices sobre la tabla de ciudades. Uno único sobre la clave primaria (`cityZip`), uno en forma de árbol B+ sobre las columnas ciudad (`cityName`) y provincia (`cityRegion`) y el último de tipo de mapa de *bits* sobre la columna provincia (`cityRegion`).

```
CREATE UNIQUE INDEX indexOnZip
   ON City(cityZip)
   TABLESPACE tbs_indexs;

CREATE INDEX indexOnNameAndRegion
   ON City(cityName, cityRegion)
   TABLESPACE tbs_indexs;

CREATE BITMAP INDEX indexOnRegion
   ON City(cityRegion)
   TABLESPACE tbs_indexs;
```

Notad que si en la tabla de ciudades hubiésemos definido la clave primaria (que es la opción conceptualmente correcta) el índice único sobre la columna `cityZip` se hubiese creado automáticamente, sin necesidad de ejecutar una sentencia separada de creación del índice.

3.2.2. Definición de índices en PostgreSQL

Los índices permitidos en PostgreSQL son en forma de árbol B+, de dispersión, GiST y GIN. Por defecto, si no se indica lo contrario, se utilizan índices basados en árboles B+. Los índices de tipo GiST y GIN se explicarán más adelante.

Aunque PostgreSQL no ofrece índices de tipo mapa de *bits*, utiliza mapas de *bits* internamente para acelerar las consultas. En particular, utiliza un sistema de mapa de *bits* para procesar operaciones lógicas entre distintos índices sobre una misma tabla. Para hacerlo, el sistema recorre cada índice involucrado en la consulta y prepara un mapa de *bits* en memoria indicando las localizaciones de las filas de la tabla que satisfacen las condiciones del índice. Posteriormente, se realizan las operaciones de `AND` y `OR` necesarias sobre los mapas de *bits* para identificar y devolver las filas que forman parte del resultado de la consulta formulada.

Es importante notar que, al igual que en el caso de Oracle, PostgreSQL también incluye otros tipos de índices en sus extensiones. Por ejemplo, la extensión de PostgreSQL para tratar datos espaciales (PostGIS) también incluye índices basados en Quadtrees y árboles R.

A continuación mostramos de forma simplificada la sintaxis de la sentencia para crear índices en PostgreSQL:

```
CREATE [UNIQUE] INDEX [CONCURRENTLY] [name] ON table_name [USING method]
   ({column_name} [opclass] [ASC|DESC] [,...])
   [TABLESPACE tablespace_name]
```

Algunas de las opciones a definir en la creación de un índice son:

- `UNIQUE`: permite indicar que la columna indexada no permite valores duplicados.

- `CONCURRENTLY`: normalmente, en la creación de un índice, PostgreSQL bloquea la tabla indexada para no permitir operaciones de inserción, borrado y modificación mientras el índice está siendo creado. En algunos casos, como por ejemplo en tablas de gran tamaño, la indexación puede tardar horas. La opción `CONCURRENTLY` permite no bloquear la tabla indexada mientras se crea el índice. Eso permite que la tabla continúe siendo accesible mientras se indexa, pero requiere que la tabla sea analizada dos veces (una al crear el índice y otra para ver los cambios que ha habido desde la creación del índice). Por lo tanto, esta opción es un arma de doble filo que hay que saber usar adecuadamente, ya que garantiza la disponibilidad de la tabla indexada pero requiere más tiempo de indexación y más carga de CPU para crear el índice.

- `Method`: es el tipo de índice a usar. Los valores posibles son `btree`, `hash`, `gist`, `spgist` y `gin`.

- `Opclass`: permite definir un tipo de operador distinto para establecer el orden de los valores indexados, como por ejemplo un operador específico para ordenar números reales o coordenadas. Se puede definir un operador para cada columna indexada.

- `TABLESPACE`: indica el espacio de tablas a utilizar para almacenar el índice.

Los índices agrupados no existen explícitamente en PostgreSQL, sino que hay que simularlos vía la sentencia `CLUSTER`. Esta sentencia permite ordenar físicamente las filas de una tabla de acuerdo a un índice anteriormente creado. La sentencia `CLUSTER` ordena la tabla al ejecutarse, pero una vez completada su ejecución, las nuevas filas son añadidas en la tabla de forma secuencial según van llegando. Para volver a reordenar la tabla hay que volver a ejecutar la sentencia `CLUSTER` sin parámetros. A continuación mostramos la sintaxis de la sentencia:

```
CLUSTER [VERBOSE] table_name [USING index_name]
```

La cláusula `VERBOSE` permite mostrar el resultado de la ordenación de la tabla. Por otro lado, el parámetro `index_name` indica el índice (en concreto su nombre) que debe usarse para ordenar la tabla.

A continuación vamos a ver unos ejemplos para mostrar cómo crear índices en PostgreSQL. Supongamos la tabla de ciudades vista anteriormente:

```
CREATE TABLE City (
   cityZip        CHAR(5) NOT NULL,
   cityName       VARCHAR(50),
   cityRegion     CHAR(2),
   cityhabitants  NUMBER(10)
) TABLESPACE tbs_slow_access;
```

Las siguientes sentencias permiten crear tres índices sobre la tabla de ciudades. Uno único sobre la clave primaria (`cityZip`) de tipo dispersión, uno en forma de árbol B+ sobre las columnas ciudad (`cityName`) y provincia (`cityRegion`) y el último también en forma de árbol B+ sobre la provincia (`cityRegion`). La última sentencia permite ordenar la tabla según la columna `cityRegion`, emulando un índice agrupado sobre la provincia.

```
CREATE UNIQUE INDEX indexOnZip
   ON City USING hash (cityZip)
   TABLESPACE tbs_indexs;

CREATE INDEX indexOnNameAndRegion
```

```
    ON City(cityName, cityRegion)
    TABLESPACE tbs_indexs;

CREATE INDEX indexOnRegion
    ON City USING btree (cityRegion)
    TABLESPACE tbs_indexs;

CLUSTER City USING indexOnRegion;
```

A continuación introduciremos los índices GiST, SP-GiST y GIN que proporciona PostgreSQL. Estudiar estos índices a fondo excede las pretensiones de este material.

Lectura complementaria

Más información sobre los índices GiST, SP-GiST y GIN puede encontrarse en los capítulos 55, 56 y 57 del manual de PostgreSQL.

http://www.postgresql.org/docs/9.3/static/index.html

Índices GiST

Los índices GiST no son exactamente un tipo de índice, sino una infraestructura que proporciona diferentes estrategias de indexación y que permite implementar nuevas estrategias. GiST son las siglas de "árbol de búsqueda generalizado" (o *generalized search tree* en inglés). Este tipo de índice proporciona un método de acceso en una estructura de árbol equilibrado que sirve como patrón para implementar esquemas de indexación arbitrarios. Se pueden utilizar este tipo de índices, por ejemplo, para crear índices en forma de árbol B+ o árbol R.

Tradicionalmente, implementar un nuevo método de acceso en un índice era una tarea compleja, ya que requería un conocimiento profundo de los detalles internos de funcionamiento de la base de datos: bloqueos, registros del dietario (en inglés, *log*), transacciones, etc. La interfaz GiST proporciona un nivel de abstracción superior que permite a los desarrolladores implementar un nuevo método de acceso a un índice indicando solo la semántica del tipo de datos a indexar. Por lo tanto, este tipo de índice permite personalizar los índices a nuevos tipos de datos, de forma que expertos en el tipo de datos a indexar puedan hacerlo, aunque no sean expertos en el funcionamiento de la base de datos. PostgreSQL provee de algunos métodos de acceso implementados en GiST, como son los de números reales, hipercubos, pares de *<clave, valor>* y textos con operadores que permiten calcular la similitud entre textos.

Lectura complementaria

Más información sobre cómo extender los índices GiST puede encontrarse en:

http://www.postgresql.org/docs/current/interactive/gist.html

El lector se puede preguntar por qué es necesario un mecanismo de extensión como este habiendo índices de basados en árboles B+ y dispersión en PostgreSQL. La respuesta es que los índices GiST pueden usarse también para dar soporte a nuevos operadores, y no solo a los operadores permitidos por defecto en los índices en forma de árbol B+ (<,<=, =, >, >=) y en los de dispersión (=). Por lo tanto, usando índices de tipo GiST se puede redefinir la semántica de los operadores existentes o crear operadores nuevos. Por ejemplo, se podría definir un nuevo índice para indexar imágenes que proporcionara operadores

para identificar la similitud entre dos imágenes, para identificar cuándo una imagen esta sobreexpuesta (es decir, tiene demasiada luz), o para segmentar una imagen.

Las últimas versiones de PostgreSQL (desde la 9.2) incorporan también los índices de tipo SP-GiST. La filosofía de este tipo de índices es la misma que en el caso de los índices GiSP, pero para árboles de búsqueda no equilibrados. Este tipo de índices permiten implementar índices que no era posible implementar con GiST, como por ejemplo Quadtrees o K-D trees.

Índices GIN

Los índices *generalized inverted index* (GIN) son índices de tipo invertido, que pueden gestionar valores que contienen más de un valor clave, donde el elemento a indexar se concibe o puede ser visto como multivaluado. Eso los hace especialmente interesantes para indexar valores compuestos en búsquedas que requieran identificar las filas que contienen determinados valores componentes. Por ejemplo, el elemento a indexar pueden ser documentos, y las consultas tienen como objetivo recuperar los documentos que contienen un conjunto específico de palabras.

En este tipo de índices se utiliza el término *ítem* para referirse al valor compuesto a indexar (el documento) y la palabra *clave* para referirse a los valores a buscar (las palabras). Los índices GIN almacenan un conjunto de pares (*clave*, $\{RID_0,..., RID_N\}$) que indican que el valor de la clave se encuentra en las filas cuyas direcciones son $RID_0,..., RID_N$.

Al igual que en los índices GiST y SP-GiST, los índices GIN pueden dar soporte a la creación de estrategias de indexación por parte del usuario.

3.3. Optimización de consultas

La optimización de consultas es uno de los aspectos más importantes que hace falta considerar cuando se diseña y construye un SGBD relacional, ya que las técnicas que se utilizan para optimizar consultas condicionan el rendimiento global del sistema. En concreto, afecta al tiempo que necesita el SGBD para responder a las consultas realizadas por los usuarios.

El proceso de optimización de consultas está incluido dentro de un proceso más general que se conoce como procesamiento de consultas. El **procesamiento de consultas** consiste en la transformación de una consulta (expresada con un lenguaje de consultas relacional, como SQL) en un conjunto de instrucciones de bajo nivel. Este conjunto de instrucciones de bajo nivel constituye una estrategia para acceder a los datos con un consumo de recursos mínimo. Precisamente, el objetivo del proceso de optimización de consultas consiste en encontrar esta estrategia.

Los usuarios de un SGBD relacional acceden a las bases de datos gestionados por el SGBD mediante consultas expresadas en un lenguaje relacional, generalmente en SQL. Como ya sabéis, SQL se caracteriza principalmente por ser un lenguaje declarativo. La implicación principal de este hecho es que cuando un usuario realiza una consulta indica cuáles son los datos que desea, qué condiciones deben cumplir y en qué tablas se encuentran, pero no dice cómo tienen que encontrarse estos datos ni en qué orden tienen que ejecutarse las operaciones especificadas en la consulta.

Es misión del SGBD determinar cómo tiene que ejecutarse la consulta: tiene que saber cómo están almacenados los datos y qué caminos de acceso (índices) se han definido sobre ellos. Además, tiene que determinar en qué orden se tienen que ejecutar las operaciones que se piden en la consulta. Dada una consulta, tendremos muchas alternativas para resolverla; el SGBD tendrá que evaluar el coste de cada una y escoger aquella cuyo coste sea menor. Esta alternativa recibe el nombre de **plan de acceso de la consulta**.

Para poder optimizar una consulta, el SGBD necesita pasar de un lenguaje declarativo a un lenguaje procedimental. A continuación veremos de forma esquemática cómo se realiza esta traducción dentro del procesamiento de consultas.

3.3.1. El álgebra relacional en el procesamiento de consultas

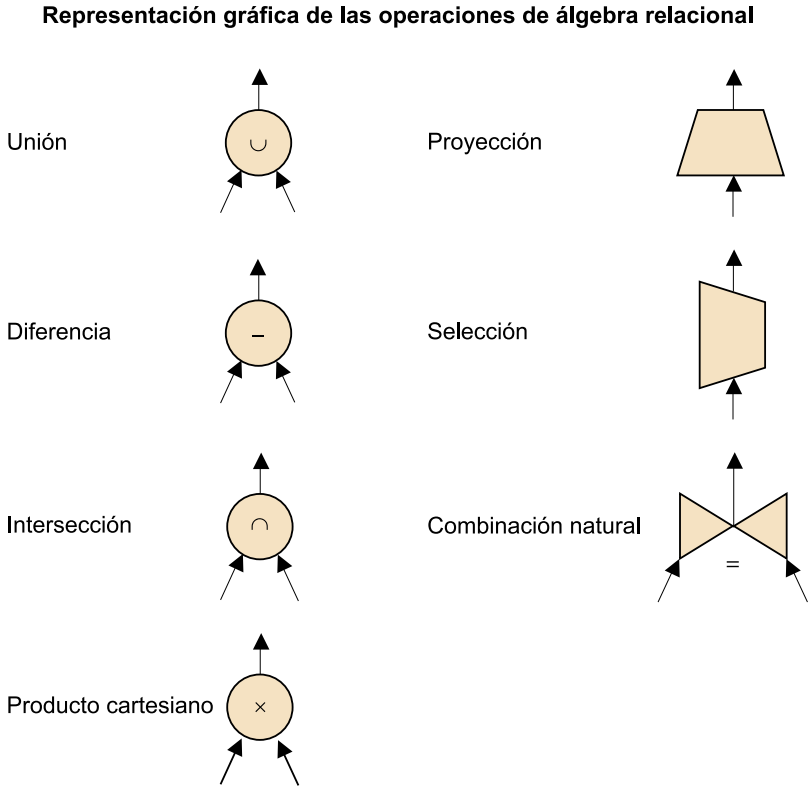
Dada una consulta formulada en lenguaje SQL, es posible encontrar diversas consultas equivalentes expresadas en álgebra relacional. Cada una de estas consultas en álgebra relacional puede representarse gráficamente mediante una estructura en forma de árbol, la cual se conoce con el nombre de árbol sintáctico de la consulta.

El **árbol sintáctico de una consulta** es una estructura en forma de árbol que corresponde a una expresión de álgebra relacional, donde:

- Las **hojas del árbol** representan las tablas que intervienen en la consulta.
- Los **nodos intermedios** son las operaciones de álgebra relacional que intervienen en la consulta original. La aplicación de cada una de estas operaciones da lugar a una tabla intermedia.
- La **raíz del árbol** constituye la respuesta a la consulta formulada.

Todas las operaciones de álgebra relacional pueden representarse gráficamente. En la siguiente figura encontraréis la representación de cada una de las operaciones que ya conocéis:

Figura 7. Representación gráfica de las operaciones de álgebra relacional



Ejemplo: Creación de los posibles árboles sintácticos de una consulta

Imaginemos una base de datos de una empresa que quiere tener constancia de quiénes son sus trabajadores, qué proyectos se desarrollan en un momento determinado y qué empleados están asignados a cada proyecto. A continuación mostramos los esquemas de las tablas que almacenan estos datos:

- EMPLEADOS(num_empl, nombre_empl, categoría_laboral, división, sueldo, jefe)
- PROYECTOS(num_proy, nombre_proy, producto, duración)
- ASIGNACIONES(num_empl, num_proy, dedicación)

Las columnas subrayadas son las claves primarias de cada tabla. También tenemos las siguientes claves foráneas:

- La columna jefe de la tabla EMPLEADOS es una clave foránea que referencia la tabla EMPLEADOS.
- La columna número de empleado (*num_empl*) de la tabla ASIGNACIONES es una clave foránea que referencia la tabla EMPLEADOS.
- La columna número de proyecto (*num_proy*) de la tabla ASIGNACIONES es una clave foránea que referencia la tabla PROYECTOS.

Supongamos que un usuario quiere conocer información sobre los empleados que dedican más de 200 horas a un proyecto. En concreto, el usuario quiere encontrar el nombre y el número de estos empleados y el número de proyectos a los que han dedicado más de 200 horas. Una posibilidad sería formular la siguiente consulta de SQL:

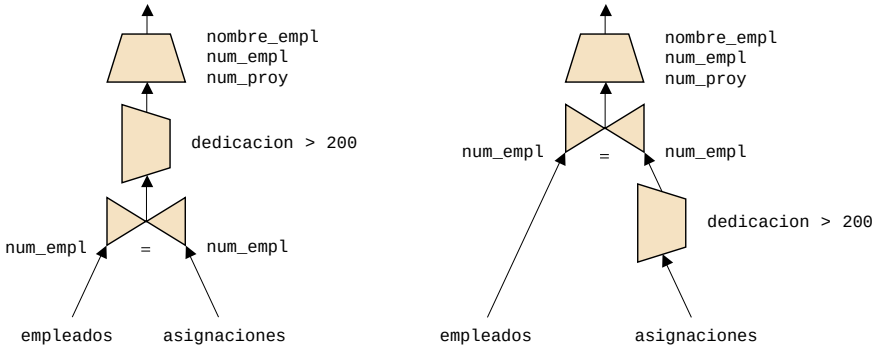
```
SELECT empleados.nombre_empl, empleados.num_empl,
       asignaciones.num_proy
 FROM  empleados, asignaciones
 WHERE empleados.num_empl=asignaciones.num_empl AND
       asignaciones.dedicacion>200;
```

Si nos fijamos, veremos que en esta consulta se incluyen las siguientes operaciones de álgebra relacional:

- Combinación natural: entre las columnas *num_empl* de las tablas EMPLEADOS y ASIGNACIONES.
- Selección: seleccionar las asignaciones con dedicación superior a 200.
- Proyección: interesan las columnas nombre empleado, número empleado y número de proyecto.

Si combinamos estas operaciones, podemos obtener, por ejemplo, los árboles sintácticos que presentamos a continuación:

Figura 8. Ejemplos de dos posibles árboles sintácticos que dan solución a una misma consulta SQL



Podéis observar que las hojas de ambos árboles son las tablas implicadas en la consulta; que los nodos intermedios son las operaciones de álgebra relacional especificadas en la consulta; y que la raíz de ambos árboles representa la respuesta a la consulta.

Como podemos ver, ambos árboles sintácticos son equivalentes a la consulta original formulada en SQL. La diferencia existente entre ambos viene determinada por el orden en el que se ejecutan las operaciones de álgebra relacional. En el caso del árbol de la izquierda, en primer lugar se ejecuta la operación de combinación natural, mientras que en el árbol de la derecha, primeramente se resuelve la operación de selección.

3.3.2. El proceso de optimización sintáctica de consultas

El objetivo de la optimización de consultas es encontrar el plan de acceso de la consulta; es decir, la estrategia de ejecución de la consulta que tiene asociado un **coste mínimo**. Por coste mínimo entendemos que el intervalo de tiempo necesario para encontrar la respuesta a la consulta formulada para el usuario tiene que ser lo más breve posible.

Hay diferentes **factores de coste** que influyen en el tiempo necesario para resolver una consulta. De todos ellos, el más importante es el número de accesos (u operaciones de E/S) que tienen que realizarse en el dispositivo de almacenamiento no volátil, en general a disco, que depende básicamente de las cardinalidades de las tablas, las relaciones existentes entre tablas que intervienen en la consulta y de la longitud de sus filas.

Factores de coste

Otros factores de coste serían el tiempo necesario para acceder a la memoria principal y el tiempo de unidad de proceso (CPU) preciso para ejecutar las operaciones solicitadas en la consulta.

A continuación vamos a ver con un ejemplo la importancia de optimizar consultas. Evaluemos primeramente el coste aproximado (en términos de número de filas a las que se ha accedido) de tres estrategias de ejecución posibles para la consulta presentada en el ejemplo del subapartado anterior:

```
SELECT empleados.nombre_empl, empleados.num_empl, asignaciones.num_proy
   FROM empleados, asignaciones
   WHERE empleados.num_empl = asignaciones.num_empl AND asignaciones.dedicacion > 200;
```

Imaginemos que tenemos los siguientes datos:

- card(EMPLEADOS) = 100,
- card(PROYECTOS) = 20,
- card(ASIGNACIONES) = 400,
- Aproximadamente un 30% de los empleados que participan en proyectos dedican más de 200 horas.

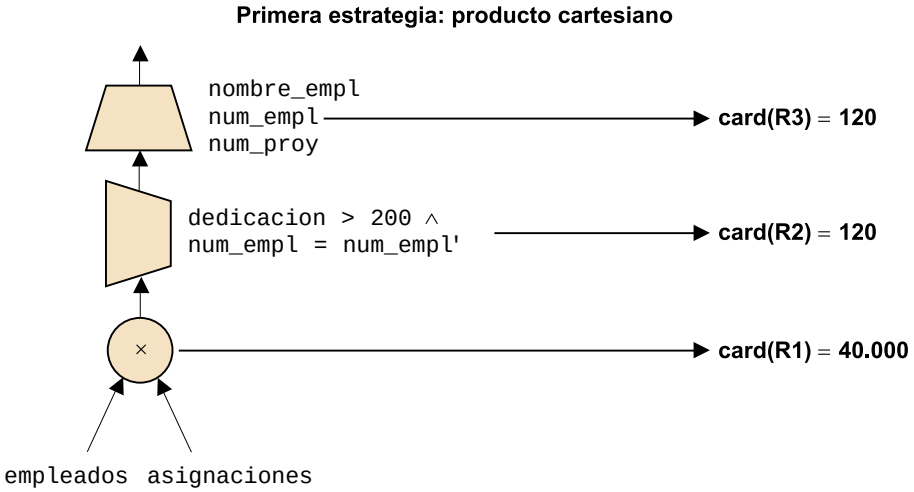
Recordad

La cardinalidad de una tabla T cualquiera se presenta como card(T), y es el número de filas que pertenecen a la extensión de T.

Podemos implementar, entre otras, las tres estrategias siguientes:

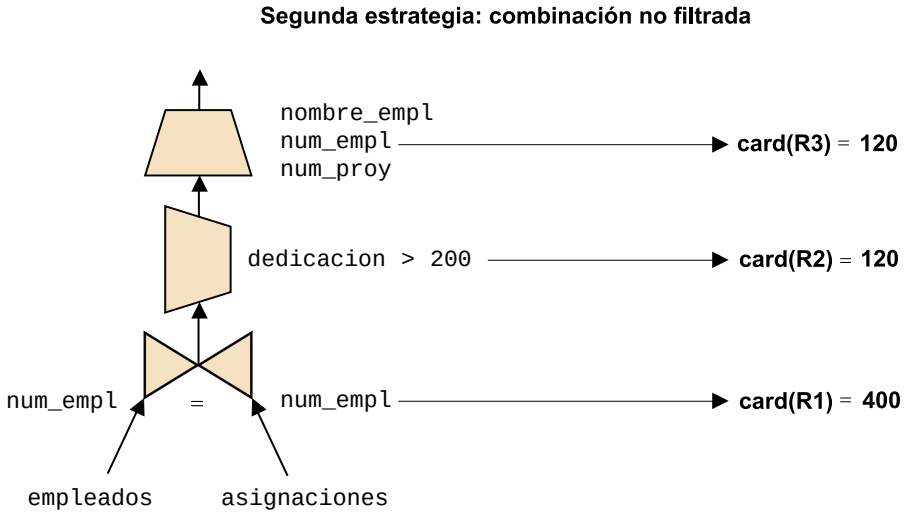
1) Producto cartesiano

Figura 9. Ejemplo de estrategia de consulta utilizando el producto cartesiano



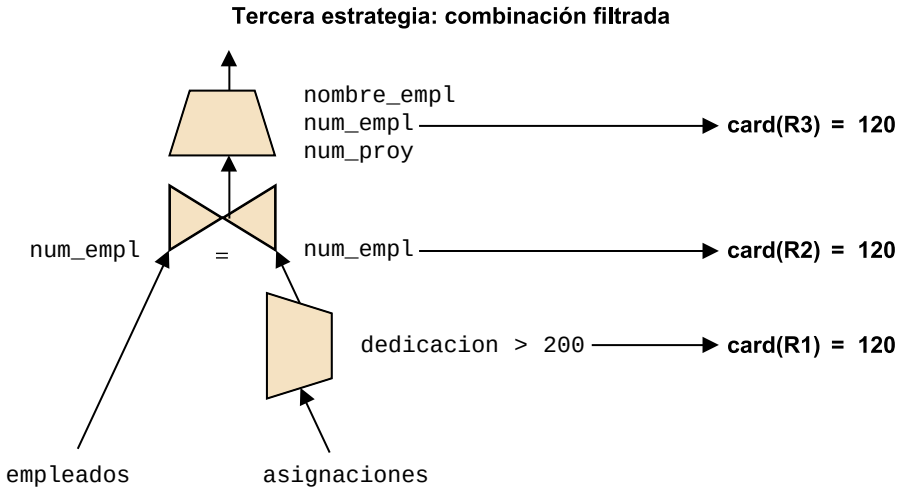
2) Combinación no filtrada

Figura 10. Ejemplo de estrategia de consulta empezando con una operación de combinación



3) Combinación filtrada

Figura 11. Ejemplo de estrategia de consulta utilizando una operación de combinación después de haber filtrado las filas innecesarias



Podemos observar que cada estrategia tiene un coste diferente. La mejor estrategia es la combinación filtrada (la tercera), porque es la que genera tablas intermedias de menor cardinalidad y, por lo tanto, será la estrategia que implicará un número más reducido de operaciones de E/S. Esta estrategia ejecuta las instrucciones tal y como se presenta a continuación:

- Primeramente se ejecuta la operación de selección; esta operación da como resultado una tabla intermedia con 120 filas, ya que sabemos que un 30% de los empleados que participan en proyectos dedican más de 200 horas.

- A continuación, se ejecuta la operación de combinación natural, que genera una tabla intermedia con 120 filas, ya que como la columna número de empleado (*num_empl*) de la tabla ASIGNACIONES es la clave foránea que referencia la tabla EMPLEADOS, toda fila de la tabla intermedia anterior encontrará pareja en la tabla EMPLEADOS.

- Finalmente se ejecuta la operación de proyección, que proporciona la respuesta a la consulta original formulada por el usuario. Como la columna número de empleado (*num_empl*) es la clave primaria de la tabla EMPLEADOS, no tendremos dos filas iguales y, por lo tanto, la respuesta estará compuesta por 120 filas.

El orden de ejecución de las operaciones formuladas dentro de una consulta repercute directamente en el coste de ejecución de la consulta. Por lo tanto, es importante encontrar el mejor orden de ejecución posible para las operaciones de cada consulta. Desgraciadamente, encontrar este orden de ejecución requiere mucho tiempo si las consultas no son simples. Por esa razón, los SGBD relacionales aplican métodos heurísticos a fin de encontrar una estrategia de ejecución razonablemente adecuada para resolver una consulta determinada.

La **optimización sintáctica** de una consulta es el proceso que determina un orden de ejecución razonablemente adecuado de las operaciones que incluye una consulta.

La optimización sintáctica de la consulta comienza con la traducción de la consulta original en un árbol sintáctico equivalente y finaliza cuando se ha encontrado el árbol sintáctico óptimo. Para encontrar el árbol sintáctico óptimo, el SGBD relacional aplica los siguientes métodos heurísticos:

- Ejecutar las operaciones de álgebra relacional que disminuyan la cardinalidad de los resultados intermedios lo antes posible. Las operaciones de álgebra relacional que disminuyen la cardinalidad de la tabla resultante son la selección, la proyección, la diferencia y la intersección.

- Retrasar al máximo la ejecución de las operaciones que incrementan la cardinalidad de los resultados intermedios. Las operaciones de álgebra relacional que incrementan la cardinalidad de la tabla resultante son el producto cartesiano, la unión y la combinación.

Si leemos detenidamente las condiciones anteriores, vemos que la optimización sintáctica reordena simplemente las operaciones de álgebra relacional que figuran en la consulta. En cambio, en nuestro ejemplo hemos considerado la cardinalidad de las tablas intermedias. De hecho, la evaluación del coste del árbol sintáctico óptimo obtenido como resultado de la optimización sintáctica se realiza posteriormente, en la etapa de optimización física de la consulta.

3.3.3. El proceso de optimización física de consultas

El objetivo de la **optimización física** de una consulta es evaluar el coste total de ejecución del árbol sintáctico óptimo asociado a una consulta de un usuario. Este coste de ejecución de un árbol sintáctico es igual a la suma de los costes de todas sus operaciones.

El coste de cada operación incluye tanto el coste de ejecución de la operación en sí como el de escribir el resultado de la operación en una tabla intermedia. Para poder aproximar el coste de las diferentes operaciones, el SGBD relacional necesita conocer la siguiente información:

1) Las estructuras de almacenaje definidas en el esquema interno. De estas estructuras, es importante tener información sobre los siguientes aspectos:

- En qué ficheros se han almacenado las tablas implicadas en la consulta y dónde se encuentran estos ficheros.

- Si se han definido estructuras de agrupación para las filas de una tabla según el valor de una o más columnas.

- Si se han definido vistas materializadas sobre las tablas implicadas en la consulta.

- Los índices (y su tipo) que se han definido en la base de datos.

2) Datos estadísticos sobre las tablas (y sus columnas) e índices definidos en la base de datos y que se almacenan en el catálogo de la base de datos. Algunos ejemplos serían:

- Tabla: cardinalidad, cuántas páginas ocupa, la ocupación media de cada página, número de páginas vacías, tamaño medio de las filas.

- Columna: número de valores diferentes en la columna, número de valores nulos en la columna, valor mínimo y máximo de la columna, histograma de frecuencia de aparición de cada uno de los valores etc. Estos datos son especialmente relevantes en el caso de columnas sobre las que se hayan construido índices.

- Índice: los datos dependerán del tipo de índice bajo consideración. Por ejemplo, en el caso de un árbol B+, interesa saber el número de nodos hoja (recordemos que cada nodo equivale a una página) que contiene, la ocupación media de los nodos hoja, la altura y orden del árbol B+.

3) Los algoritmos de implementación de las operaciones de álgebra relacional y de otras extensiones propuestas por SQL (por ejemplo, las cláusulas `ORDER BY`, `DISTINCT`, funciones de agregación etc.) que hay disponibles. Estos algoritmos también se conocen bajo la denominación de métodos de acceso.

La información previamente descrita permite que un SGBD relacional pueda decidir qué algoritmos son aplicables y cuáles no. La aplicabilidad de estos algoritmos depende de los elementos expuestos en el punto 1. Para acabar, el SGBD deberá estimar el coste de cada uno de los algoritmos aplicables y escoger el algoritmo que tenga el coste más bajo, basándose en los datos estadísticos. Esto se debe realizar para todas las operaciones que forman parte de la consulta, y da lugar a lo que se denomina el plan de la consulta.

Datos estadísticos

La gestión de datos estadísticos es crucial para que el proceso de optimización física se realice de forma correcta. Los SGBD proporcionan sentencias SQL que ayudan a la recogida y actualización de los datos estadísticos de los objetos definidos en la base de datos, como por ejemplo la sentencia `ANALYZE` que existe en Oracle y PostgreSQL y que permite obtener estadísticas sobre elementos de la base de datos. En el caso de PostgreSQL, como las filas borradas no se eliminan físicamente de la tabla, se puede añadir el comando `VACUUM` a `ANALYZE` para eliminarlas antes de analizar las estadísticas.

Ejemplo: ¿Qué algoritmo debemos escoger?

Para el árbol sintáctico óptimo encontrado en el ejemplo anterior, el SGBD necesita evaluar el coste de las operaciones de combinación natural y de selección que figuran. Supongamos que sobre la columna dedicación de la tabla ASIGNACIONES se ha definido un índice.

En ese caso, para implementar la operación de selección física, el SGBD podría escoger entre dos algoritmos:

- Un algoritmo que no hiciera uso del índice. En este caso, realizaríamos un recorrido secuencial del fichero que almacena la tabla ASIGNACIONES y comprobaríamos la condición pedida a la consulta (dedicación superior a 200 horas). Por cada fila que cumpla la condición, proyectaríamos las columnas que necesitásemos (*num_empl* y *num_proy*), y las guardaríamos en una tabla intermedia R1.

- Un algoritmo que hiciera uso del índice. Entonces, mediante el índice, accederíamos solamente a las filas que verificaran la condición, proyectaríamos las columnas que nos interesaran y guardaríamos el resultado en una tabla intermedia R1.

De los dos algoritmos, intuitivamente vemos que el segundo seguramente tendrá asociado un coste más bajo, y sería el algoritmo que escogería el SGBD.

Como hemos comentado, cada SGBD tiene un conjunto de algoritmos o métodos de acceso para resolver las operaciones (tanto las de álgebra relacional como las añadidas por SQL) incluidas en las consultas que formulan los usuarios. Los métodos de acceso más comunes son:

- *Table scanning*: cuando no se utiliza un índice y la consulta se resuelve realizando un recorrido secuencial de toda la tabla. Se puede utilizar, por ejemplo, en la resolución de consultas que únicamente incorporan operaciones del álgebra relacional de selección y/o proyección.

- *Sorting algorithms*: permiten ordenar los datos de las tablas de la base de datos o de tablas intermedias que se generan durante el procesamiento de la consulta y que sirven de soporte para algún otro método de acceso (por ejemplo, los algoritmos de *join* que introduciremos posteriormente). También puede resultar de utilidad en la resolución de consultas que incorporan las cláusulas `ORDER BY` y `GROUP BY`, y para la eliminación de duplicados (cláusula `DISTINCT` de SQL o para la resolución de operaciones de unión de álgebra relacional).

- *Index/index-only scanning* y *Block or row index ANDing*: estos métodos ayudan principalmente a resolver consultas que incorporan alguna operación de selección del álgebra relacional sobre columnas para las que se ha definido un índice. El método *index/index-only scanning* se basa en el uso de un único índice, mientras que *block or row index ANDing* se basa en el uso de diversos índices y sirve para fusionar las entradas de dichos índices. El *index/index-only scanning* también puede resultar de utilidad en la resolución de consultas que incorporan las cláusulas `ORDER BY`, `GROUP BY` o funciones de agregación (por ejemplo, `COUNT`, `MIN` o `MAX`).

- *Join*: permiten resolver consultas que incorporan operaciones del álgebra relacional de combinación entre tablas. Existen distintas variantes de este método de acceso. A continuación se describen las más relevantes (el pseudocódigo asociado se muestra en la figura 12):
  - *Nested loop*: consiste en el recorrido secuencial de las tablas implicadas en la combinación, a través de dos bucles imbricados (uno por cada tabla). En última instancia, para cada fila de la tabla que se recorre en el bucle exterior se obtienen todas las filas de la tabla que se recorre en el bucle interior que cumplen la condición expresada en la combinación. Este método siempre se puede aplicar, con independencia de operador de comparación (=, <>, <, <=, >, >=) indicado en la combinación. La tabla de menor tamaño es la que se asocia al bucle exterior.

  - *Index nested loop*: es una variante del método anterior, en donde en el bucle interior no se realiza un recorrido completo de la tabla a la que se asocia, sino que únicamente se accede a las filas que cumplen la condición expresada en la combinación mediante un índice. Si el índice es de dispersión, solo se podrá aplicar cuando el operador de

comparación usado en la combinación sea la igualdad (equicombinación y combinación natural). Si el índice se basa en un árbol B+, el método siempre se podrá aplicar.

– *Sort-Merge*: consiste en la lectura secuencial de las tablas implicadas en la operación de combinación, tomando las filas que verifiquen la condición expresada en la combinación. Requiere que las tablas estén ordenadas físicamente según el valor de las columnas sobre las que se realiza la combinación. Si no están ordenadas, se pueden usar los *sorting algorithms* previamente presentados. Únicamente es aplicable cuando el operador es la igualdad (equicombinación y combinación natural).

– *Hash join*: este método, en su forma más simple, se basa en la construcción de un índice de dispersión sobre la tabla que contiene menos filas (en concreto, se construye sobre la columna implicada en la combinación). A continuación, sobre la otra tabla que interviene en la combinación se realiza un recorrido secuencial y se usa el índice de dispersión construido para saber qué filas forman parte del resultado de la combinación. Solo se puede aplicar en equicombinación y combinación natural. Una vez construido el índice de dispersión, este método se puede considerar un caso particular de *index nested loop*.

Figura 12. Pseudocódigo métodos de *join*

Nested loop: R[AθB]S, θ∈{=,<>,>=,>,<,<=}, card(R)<=card(S)

```
para cada página de R
        transferir página de R de disco a memoria principal
        para cada página de S
          transferir página de S de disco a memoria principal
          para cada fila t de la página de R
            para cada fila s de la página de S
              si t.Aθs.B entonces generar resultado
              fsi
      fpara
    fpara
  fpara
fpara
```

Index nested loop: R[AθB]S, existe índice I_B sobre la columna B de S, θ depende del tipo de índice.

```
para cada página de R
        transferir página de R de disco a memoria principal
        para cada fila t de la página de R
          usar IB para buscar entradas con valor t.A
          si existen entradas en IB que cumplan condición de join
            entonces generar resultado (puede implicar acceso a S)
          fsi
  fpara
fpara
```

Sort-Merge: R[A=B]S

```
ordenar R según A (si R no está ordenada)
ordenar S según B (si S no está ordenada)
recorrer páginas de R y S aplicando algoritmo de merge
 para generar resultado
```

Hash join: R[A=B]S, card(R)<=card(S)

```
crear índice hash HA sobre R.A
para cada página de S
        transferir página de S de disco a memoria principal
        para cada fila s de la página de S
          usar HA para buscar entradas con valor s.B
          si existen entradas en HA
            entonces generar resultado (puede implicar acceso a R)
          fsi
  fpara
fpara
```

Los SGBD pueden soportar todos o una parte de los métodos de acceso que acabamos de explicar. También pueden introducir variantes, por lo que será imprescindible consultar la documentación del fabricante. Las tablas interme-

dias que se generen durante la ejecución de las consultas, si no se pueden almacenar en memoria interna debido a su tamaño, se guardan en memoria externa (por ejemplo, en disco). Para ello, el SGBD utiliza espacios de tabla temporales (en inglés, *temporary tablespaces*).

Finalmente, los SGBD disponen de herramientas que facilitan el análisis de la estrategia (o plan de la consulta) seguida en la resolución de cada consulta. Ejemplos de estas son las sentencias `EXPLAIN PLAN` de Oracle o `EXPLAIN` de PostgreSQL, que permiten analizar el plan de consulta de una sentencia SQL. Debido a la similitud de `EXPLAIN` y `EXPLAIN PLAN`, presentamos únicamente el caso particular de PostgreSQL.

Analizando el plan de consulta en PostgreSQL

A continuación vamos a ver cómo comprobar el plan de una consulta SQL. En PostgreSQL los planes de consulta pueden consultarse mediante la sentencia `EXPLAIN`, que recibe una sentencia SQL como parámetro y devuelve el plan de la consulta en el siguiente formato:

Sentencia `EXPLAIN`

Ver más información de la sentencia `EXPLAIN` en: http://www.postgresql.org/docs/9.3/static/using-explain.html

```
-- Supongamos la siguiente tabla Country
CREATE TABLE Country (
   name     VARCHAR(50) UNIQUE,
   región   CHAR(2),
   ...
);

-- A continuación se consulta la siguiente sentencia mediante la sentencia EXPLAIN
EXPLAIN SELECT * FROM Country;

QUERY PLAN
-----------------------------------------------------
Seq Scan on Country (cost=0.00..458.00 rows=10000 width=244)
```

Tal y como podemos ver, esta sentencia analiza la consulta pasada por parámetro y devuelve el plan de ejecución. Como la consulta de ejemplo no tiene cláusula `WHERE`, el SGBD debe consultar todas las filas de la tabla, por lo tanto, el sistema ha elegido utilizar un método de acceso secuencial para recorrer la tabla (*seq scan*). Los números entre paréntesis tienen el siguiente significado:

- Tiempo estimado de inicialización: es el tiempo necesario antes de que el método de acceso sea capaz de devolver la primera fila del resultado. Por ejemplo, en caso de requerir una ordenación, sería el tiempo necesario para ordenar los datos.

- Tiempo estimado total: tiempo necesario para completar la operación del nodo del plan de consulta, es decir, para devolver todas las filas que conforman el resultado de la consulta.

- Número estimado de filas de salida para este nodo del plan de consulta.

- Tamaño medio estimado de filas producidas en este nodo del plan de consulta (en *bytes*).

Hemos visto en el ejemplo anterior que en PostgreSQL el método de *table scanning* se denomina *seq scan*. Para facilitar al lector la comprensión de los resultados de la sentencia `EXPLAIN`, a continuación presentamos los principales métodos de acceso utilizados en PostgreSQL y su equivalencia con los métodos de acceso presentados anteriormente:

PostgreSQL

En el capítulo titulado *Performance Tips* (http://www.postgresql.org/docs/9.3/static/performance-tips.html) del manual de PostgreSQL se puede encontrar más información al respecto en el caso particular de PostgreSQL.

- *Seq scan*, es el equivalente a *table scanning* y realiza un recorrido secuencial de toda la tabla.

- *Index scan* e *index only scan* son los equivalentes a *index scanning* e *index-only scanning*, respectivamente, y se basan en el uso de árboles B+.

- *Bitmap index scan*, *bitmap heap scan* y *recheck cond* serían los equivalentes al *index scanning* para los índices de mapas de *bits*.

- Operaciones de combinación:
  - *Nested loop* es el equivalente a *nested loop* e *index nested loop*, ya que PostgreSQL utiliza, siempre que sea posible, un índice para consultar las filas de la segunda tabla que están relacionadas con la primera.

  - *Hash join y merge join* son el equivalente a los métodos *hash join* y *sort-merge* previamente presentados.

Por ejemplo, supongamos que queremos analizar el plan de consulta de la siguiente sentencia SQL:

```
-- Supongamos la siguiente tabla Region
CREATE TABLE Region (
   id   CHAR(2) UNIQUE,
   name VARCHAR(50),
   ...
);

-- Supongamos la siguiente tabla Country
CREATE TABLE Country (
   name   VARCHAR(50) UNIQUE,
   región CHAR(2),
   ...
```

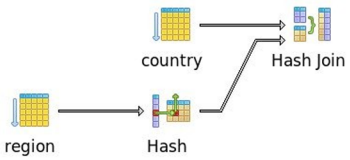
```
);


-- La sentencia SQL a analizar es la siguiente
SELECT c.name as country_name , r.name as region_name
FROM   Country c, Region r
WHERE  r.id=c.region;
```

PostgreSQL nos permite visualizar el plan de consulta textualmente o de forma gráfica, tal y como podemos ver en la siguiente figura.

Figura 13. Visualización de un plan de consulta, de forma textual y gráfica, en PostgreSQL

	QUERY PLAN text
1	Hash Join (cost=1.09..2.19 rows=4 width=24)
2	Hash Cond: (c.region = r.id)
3	-> Seq Scan on country c (cost=0.00..1.04 rows=4 width=15)
4	-> Hash (cost=1.04..1.04 rows=4 width=13)
5	-> Seq Scan on region r (cost=0.00..1.04 rows=4 width=13)



A continuación vamos a interpretar los resultados obtenidos. El método *hash join* que aparece en la raíz del árbol se utiliza para realizar la combinación (en concreto, se trata de una equicombinación) entre las tablas `REGION` y `COUNTRY`. Para realizar esta combinación, se crea una tabla temporal que contiene los datos de `REGION` y un índice de tipo dispersión (*hash*) sobre su clave primaria que facilite realizar la combinación. Para ello es necesario que primero se realice un acceso secuencial (de tipo *seq scan*) sobre la tabla `REGION`, para posteriormente obtener el índice de dispersión de la clave primaria. Una vez hecho esto, se puede realizar una lectura secuencial (de tipo *seq scan*) de las filas de la tabla `COUNTRY`. Para cada fila de país se podrá acceder a su región utilizando el índice de dispersión creado y así hacer la combinación correspondiente.

Como podemos ver, en el ejemplo anterior el SGBD ha creado un índice de dispersión temporal para aumentar la velocidad de la consulta. Nos podemos preguntar, qué hubiese pasado si ya existiera un índice sobre la clave primaria de región. Supongamos que creáramos dicho índice y volviéramos a consultar el plan de la consulta. El resultado sería el siguiente:

Figura 14. Visualización de un plan de consulta. Ejemplo con *nested loop*

```
QUERY PLAN
text
1  Nested Loop  (cost=10000000000.00..10000000008.16 rows=4 width=24)
2    ->  Seq Scan on country c  (cost=10000000000.00..10000000001.04 rows=4 width=15)
3    ->  Index Scan using region_id_index on region r  (cost=0.00..1.77 rows=1 width=13)
4          Index Cond: (id = c.region)
```

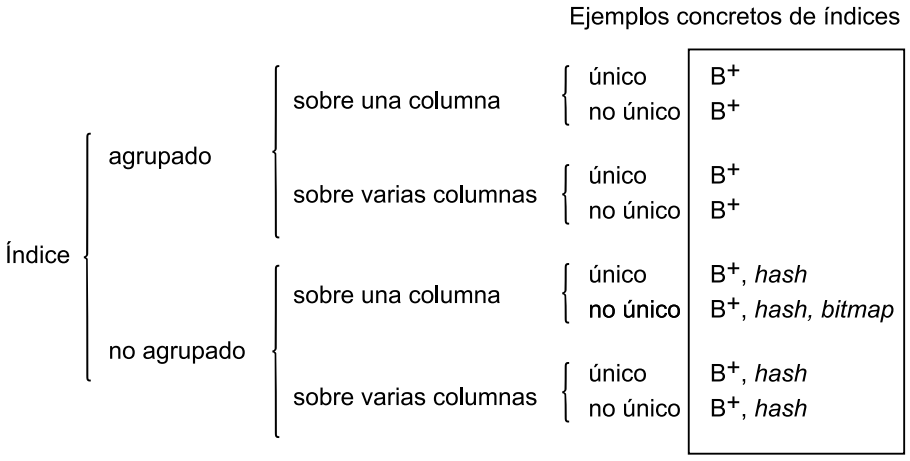
Ahora no hace falta crear un índice temporal porque la columna `id` de la tabla `REGION` ya está indexada. Por lo tanto, en este caso la consulta se realizará aplicando un método de *nested loop*. O sea, se consultarán las filas de la tabla `COUNTRY` de forma secuencial y, para cada fila, se buscará la fila de la tabla `REGION` asociada a través del índice existente.

3.4. Seleccionando qué índices y de qué tipo crear

Las peculiaridades de los índices hacen que sea aconsejable gestionarlos aparte de los datos de las tablas. Por este motivo, hay un tipo de espacio virtual para contener los índices llamado espacio de índices.

Tal y como se resume en la siguiente figura, en estos materiales hemos introducido los índices en forma de árbol B+, de dispersión y de mapas de *bits*, pero existen otros índices (hemos mencionado algunos de ellos) y variaciones de los presentados aquí que pueden ser muy útiles en determinados casos. El índice en forma de árbol B+ es el índice por excelencia en SGBD relacionales, debido a su aplicabilidad en todo tipo de situaciones y su buen rendimiento. No obstante, el uso de otro tipo de índices en circunstancias concretas puede mejorar el rendimiento de la base de datos de forma significativa.

Figura 15. Clasificación de los índices presentados en estos materiales



Hemos aprendido a definir índices y en qué casos es conveniente hacerlo. A continuación estudiaremos una serie de consideraciones que hay que tener en cuenta desde el punto de vista del rendimiento.

Antes de plantearse crear un nuevo índice es importante saber que índices crean los SGBD por defecto. Es importante consultar la documentación para el caso concreto de cada SGBD, pero por norma general podemos asumir que el SGBD creará automáticamente índices para todas las claves primarias y alternativas que se hayan definido en las tablas.

Resulta conveniente definir un índice en los siguientes casos:

- Garantizar la unicidad de algunas columnas. Definir un índice único es quizás la manera más fácil y eficiente de garantizar que no hay duplicidad de valores en la columna.

- Garantizar la ordenación de las filas dentro de la tabla. Algunos procesos pueden requerir acceder a las tablas mediante acceso secuencial por valor. En estos casos, y cuando la tabla tiene un número elevado de filas, la elección de un índice agrupado tiene una importancia crítica.

- Optimizar las comprobaciones de las restricciones de integridad referencial. El SGBD tiene que garantizar que las restricciones de integridad referencial de la base de datos no se violan. Para hacerlo debe comprobar la existencia de la clave relacionada en la otra tabla cuando se inserta o se borra una fila. En estos casos, es recomendable disponer de índice sobre las claves foráneas con el fin de optimizar los accesos de comprobación del propio SGBD.

- Columnas frecuentemente referenciadas en cláusulas `WHERE`. En general, estas columnas son buenas candidatas para un índice.

- Uso de las mismas columnas en cláusulas `ORDER BY` o `GROUP BY`. En el caso de que diversas columnas aparezcan repetidamente, o en consultas muy frecuentes, referenciadas en este tipo de cláusulas, es conveniente crear un índice. Así eliminamos la necesidad de que el SGBD ordene las filas, ya que el índice proporciona automáticamente el orden deseado o facilita la agrupación requerida.

- Combinaciones entre dos o más tablas. En un proceso de combinación el punto más crítico es establecer el enlace entre las dos tablas a través de los predicados de la cláusula `WHERE`. Si definimos un índice en cada tabla con las columnas que participan en la operación de combinación, conseguiremos, en la mayoría de los casos, que el optimizador escoja los índices para resolver la operación, mejorando así el tiempo de ejecución.

- Grandes volúmenes de datos. Cuando tenemos tablas que almacenan grandes volúmenes de datos, como puede pasar en un *data warehouse*, el uso de índices en forma de árbol B+ puede generar estructuras de datos muy pesadas de almacenar y procesar. En estos casos puede resultar más eficiente utilizar índices de tipo mapa de *bits*, que permiten definir estruc-

turas más compactas, menos pesadas y que facilitan una ejecución rápida de las operaciones.

El número y tamaño de los índices varía en las distintas bases de datos, pero normalmente se aplica una regla general que dice que los índices deben ocupar entre el 10% y el 20% del espacio de almacenamiento de la base de datos. En los casos en que ocupan más de un 25% los expertos aconsejan estudiar la estructura de la base de datos a fondo para asegurar que no hay índices superfluos. Además, teniendo en cuenta que los índices tienen un coste de mantenimiento no despreciable, se deben seguir un conjunto de reglas básicas para asegurarse de que el número de índices es adecuado. A continuación enumeramos las más importantes:

- Evitar y eliminar índices redundantes: en algunos casos puede haber índices difíciles de utilizar por el planificador de consultas. Un caso común es cuando tenemos diferentes índices que comparten columnas. En esos casos debemos asegurarnos que el planificador de consultas de nuestro SGBD es capaz de utilizar los índices creados en los casos requeridos. Para hacerlo se puede utilizar la sentencia `EXPLAIN` que hemos visto antes o alguna sentencia equivalente que ofrezca el SGBD con el que trabajemos.

- Añadir índices solo cuando sea necesario: la ventaja de añadir un nuevo índice debe ser clara y estar justificada.

- Añadir o eliminar columnas de índices compuestos para mejorar el rendimiento. Los índices compuestos, es decir aquellos que se definen sobren múltiples columnas, pueden ser muy útiles y minimizar el número de ordenaciones y combinaciones necesarias en búsquedas por múltiples columnas. No obstante, cuantas más columnas tiene un índice más difícil es su aplicación. Por lo tanto, a veces es conveniente minimizar el número de columnas indexadas en un índice para maximizar las consultas donde se puede aplicar. Evidentemente, esta regla no se aplica en caso de que el índice sea sobre una clave primaria, alternativa o foránea compuesta.

- Sopesar si es viable indexar columnas que se actualizan frecuentemente. Modificar el valor de una columna indexada implica reestructurar el índice. Por lo tanto, cuando un índice se crea sobre una columna que se actualiza a menudo, nos debemos preguntar si el tiempo que necesitará el SGBD en reestructurar el índice es menor que el tiempo que ganaremos al utilizarlo en las consultas.

- Realizar el mantenimiento de los índices regularmente, ya que las actualizaciones de la base de datos puede haber provocado de algunos índices dejen de ser rentables. Dicho mantenimiento requerirá borrar índices cuando se detecte que ya no contribuyen a mejorar el rendimiento o cuando claramente castiguen el rendimiento de la base de datos.

4. Vistas materializadas

Es probable que en nuestra base de datos tengamos consultas que se ejecuten muy frecuentemente o consultas que, aunque se ejecuten menos frecuentemente, requieran de cálculos muy complejos. En ambas situaciones surge la misma pregunta, ¿por qué no almacenamos el resultado de la consulta para el futuro? Así, cuando los usuarios vuelvan a realizar la misma consulta, el SGBD no tendrá que volver a calcular el resultado, podrá devolver el resultado guardado previamente. Con esta filosofía se crearon las vistas materializadas.

Una **vista materializada** es un objeto de la base de datos almacenado en el dispositivo de almacenamiento no volátil que contiene el resultado de una consulta precalculada. Las vistas materializadas permiten un acceso más eficiente a los datos a cambio de incrementar el tamaño de la base de datos.

Una vista materializada es parecida a una vista convencional. La diferencia estriba en que en una vista convencional su contenido es virtual y por lo tanto, su contenido se calcula cada vez que se solicita su valor, aplicando para ello un proceso de traducción, mientras que una vista materializada sí que se almacena en el dispositivo de almacenamiento no volátil y sus valores están, en consecuencia, precalculados.

Por lo tanto, podríamos ver una vista materializada como una tabla, pero cuyos valores son redundantes y precalculados a partir de otros datos (disponibles en tablas o vistas materializadas) de la base de datos. Las tablas (o vistas materializadas) sobre las que se define la vista materializada reciben la denominación genérica de tablas de base. El hecho de que las vistas materializadas se comporten como tablas permite que se puedan crear índices sobre las columnas que incorpora. En algunos casos será posible modificar los datos directamente en las vistas materializadas, propagando los cambios a las tablas de base usadas en la definición de las vistas materializadas. En estos casos, se dice que la vista materializada es actualizable. Al igual que en el caso de vistas convencionales, la definición de vistas materializadas actualizables no es siempre posible.

Se puede acceder a las vistas materializadas de forma explícita o de forma implícita. Diremos que una vista materializada se accede de forma explícita cuando esta aparece explícitamente en las sentencias SQL del usuario. Las sentencias SQL a ejecutar indican explícitamente que se debe acceder a la vista materializada como si fuera una tabla más. Esta opción obliga al usuario a conocer

las vistas materializadas existentes y los datos que contienen y puede limitar las opciones del optimizador de consultas para escoger los planes de consulta más adecuados.

Una vista materializada se accede de forma implícita cuando su nombre no aparece de forma explícita en las sentencias SQL a ejecutar. En este caso, el usuario escribe las sentencias SQL como si la vista materializada no existiera. El optimizador de consultas es el encargado de descubrir en qué casos la vista materializada debe usarse. Este método libera al usuario de saber qué vistas materializadas existen y cuándo es necesario utilizarlas. No obstante, su eficacia depende totalmente de que el optimizador de consultas sea capaz de detectar cuándo se puede usar una vista materializada para resolver una consulta y en qué casos usarla es beneficioso. Desafortunadamente, no todos los SGBD tienen ese nivel de sofisticación.

Los datos de una vista materializada se deberán actualizar de forma periódica para no quedar obsoletos. Esto es así porque los cambios que se realizan en las tablas de base no se ven por defecto automáticamente reflejados en la vista materializada. Por ello, frecuentemente se dice que las vistas materializadas constituyen fotografías (*snapshots* en inglés) de la base de datos.

Determinar cuándo y cómo se calcula el contenido de una vista materializada es crítico y puede marcar la diferencia entre una vista materializada eficiente, o una que tiene unos costes de mantenimiento más elevados que los beneficios que se obtienen por su uso. Desde un punto de vista teórico, las vistas materializadas pueden seguir diferentes estrategias de mantenimiento:

- **Actualización automática**: las vistas materializadas se pueden actualizar automáticamente sin la intervención del usuario. Existen distintas opciones, pero las más comunes son la actualización según calendario (es decir, marcar la periodicidad de la actualización con independencia de las modificaciones producidas en las tablas de base), actualización según datos (el SGBD detecta cuándo se han modificado los datos en las tablas de base y propaga los cambios a las vistas materializadas afectadas). La frecuencia y tipo de transacciones sobre las tablas base y el tipo de funciones de agregación utilizadas en la vista serán determinantes para escoger una opción u otra.

- **Actualización bajo demanda**: el usuario debe especificar cuándo se debe actualizar la vista materializada de forma explícita. Es una opción poco recomendable a utilizar en términos generales, pero puede ser útil utilizarla en casos excepcionales. Por ejemplo, este sería el caso de actualización según calendario donde haya habido muchos cambios antes de la fecha de actualización.

- **Actualización completa o incremental**: en una actualización completa se destruye la vista materializada para volverla a crear desde cero. En con-

traposición, la actualización incremental solo introduce cambios en aquellos datos que se hayan visto afectados por los cambios en las tablas de base. La actualización completa es menos eficiente que la incremental, pero tiene la ventaja que se puede realizar siempre. Es más, en algunos casos puede ser la única opción. Por ejemplo, si la vista materializada incluye funciones agregadas, es posible que estas no puedan calcularse de forma incremental, de tal manera que sea obligatorio realizar el cálculo desde cero al modificar los datos de las tablas de base. Este sería el caso de la media aritmética.

En general, las vistas materializadas son muy utilizadas en *data warehouses*. Esto es debido a que permiten precalcular operaciones de agregación muy costosas y sobre gran cantidad de datos. También ayudan a precalcular agregaciones por distintos niveles de granularidad del *data warehouse*, facilitando las operaciones de *drill up* y *drill down* sobre el mismo. Las decisiones sobre la frecuencia y modo de actualización de las vistas materializadas, tal y como se ha argumentado anteriormente, serán de importancia primordial. Por ejemplo, si una vista materializada se usa en un *data warehouse* cuyo proceso de extracción, transformación y carga (*Extraction Tranformation and Load* o ETL en inglés) se realiza cada noche, será suficiente con actualizar la vista materializada después del proceso ETL.

4.1. Definición y uso de vistas materializadas en Oracle

A continuación mostramos de forma esquemática la sintaxis de creación de vistas materializadas en Oracle:

```
CREATE CREATE MATERIALIZED VIEW materialized_view
  [column_alias [, column_alias]...]
  [[{physical_attributes_clause|TABLESPACE tablespace}]...]
  {REFRESH create_mv_refresh}
  [ENABLE QUERY REWRITE]
   ...
  AS subquery;
```

Oracle fue el precursor de las vistas materializadas, y como tal permite un alto grado de personalización. Los elementos principales en la creación de una vista materializada son:

a) `Materialized_view`: nombre de la vista materializada.

b) `Column_alias...`: permite definir el nombre de las columnas de la vista materializada.

c) `Physical_atributes_clause`: permite indicar los parámetros físicos de la vista materializada y el *tablespace* donde asignarla.

Lectura recomendada

Más información en el manual de Oracle
http://docs.oracle.com/cd/B19306_01/server.102/b14200/statements_6002.htm#i2064161

d) `REFRESH create_mv_refresh`: permite indicar cuándo y cómo actualizar la vista materializada. Algunas de las opciones permitidas son:

- Actualización completa o incremental:
  - `FAST`: indica que los datos de la vista materializada se actualizarán de forma incremental.
  - `COMPLETE`: permite indicar que los datos de la vista materializada se actualizarán de forma completa.
  - Si no se especifica ninguna opción el SGBD intentará realizar la actualización incremental. Si el SGBD no puede realizar la actualización de forma incremental entonces realizará una actualización completa.
- Actualización automática según datos, según calendario o actualización bajo demanda:
  - `ON COMMIT`: permite indicar que los datos de la vista materializada se modificarán al modificarse los datos de las tablas de base.
  - `START WITH...NEXT`: permite indicar la fecha de carga inicial de datos (`START WITH`) y configurar la actualización periódica según calendario. En concreto, la sentencia `NEXT` permite indicar el intervalo de tiempo que especifica cada cuanto se deben realizar las cargas periódicas.
  - `ON DEMAND`: indica que los datos de la vista materializada se modificarán de forma manual. Es la opción por defecto si no se usa la opción `ON COMMIT`.
  - `ENABLE QUERY REWRITE`: permite indicar si queremos que el SGBD tenga en cuenta la vista materializada en el proceso de reescritura de consultas (hablaremos de ello en las siguientes líneas). Si no se indica esta cláusula, la vista materializada no se tendrá en cuenta en el proceso de reescritura de consultas.

e) `Subquery`: indica la sentencia SQL que permite calcular la vista materializada.

Por ejemplo, la siguiente sentencia crearía una vista materializada para almacenar las ventas de una empresa agrupadas por año y producto. La vista se crearía por primera vez el día posterior a la ejecución de la sentencia SQL, a las 11 de la mañana, y se refrescaría cada lunes a las 15h de forma incremental.

```
-- Dadas las siguientes tablas de times, products y sales:

CREATE TABLE times (
   time_id          DATE        PRIMARY KEY,
```

```
   calendar_year    VARCHAR(50),
   calendar_month   VARCHAR(50),
   calendar_day     VARCHAR(50),
   ...
);

CREATE TABLE products (
   prod_id          NUMBER(6)   PRIMARY KEY,
   prod_name        VARCHAR(50),
   prod_description VARCHAR(2000),
   ...
);

CREATE TABLE sales (
   prod_id          NUMBER(6),
   cust_id          NUMBER,
   time_id          DATE,
   quantity_sold    NUMBER(3),
   amount_sold      NUMBER(10,2),
   ...
);

-- Creamos una vista materializada que permite precalcular el total de ventas agrupado
por producto y año
CREATE MATERIALIZED VIEW sales_mv
   TABLESPACE tbs_materialized_views
   REFRESH FAST START WITH ROUND(SYSDATE + 1) + 11/24
   NEXT NEXT_DAY(TRUNC(SYSDATE), 'MONDAY') + 15/24
   AS SELECT t.calendar_year, p.prod_id,
      SUM(s.amount_sold) AS sum_sales
      FROM times t, products p, sales s
      WHERE t.time_id = s.time_id AND p.prod_id = s.prod_id
      GROUP BY t.calendar_year, p.prod_id;
```

Oracle permite acceder a las vistas materializadas de forma implícita y explícita. No obstante, aconseja evitar el acceso explícito (usar el nombre de la vista materializada en las sentencias SQL) ya que eso limita la posibilidad de introducir cambios en las vistas materializadas en un futuro. Para utilizar vistas materializadas de forma explícita, solo se debe añadir la vista materializada en las sentencias SQL. Para utilizar vistas materializadas de forma implícita, Oracle utiliza la funcionalidad de reescritura de consultas (*query rewrite* en inglés).

La reescritura de consultas es una técnica de optimización que transforma una sentencia SQL que utiliza tablas de la base de datos en otra sentencia SQL que es equivalente semánticamente y que incluye una o más vistas materializadas. Conceptualmente, podemos ver este proceso como un proceso donde se re-

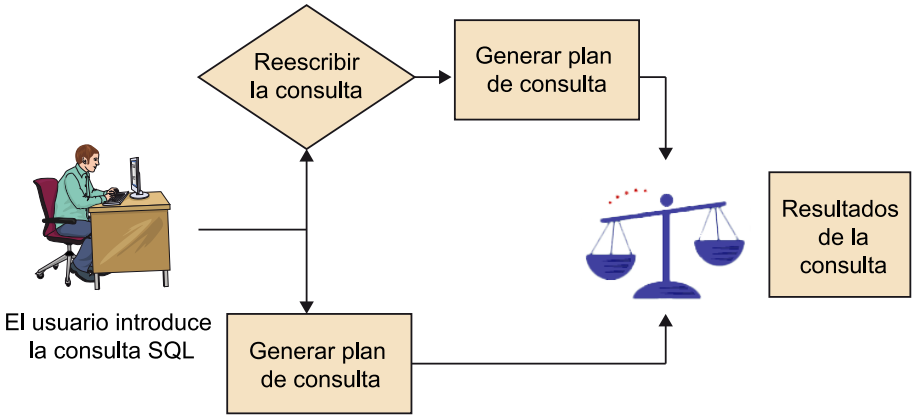
formula la sentencia SQL para que acceda a los datos a través de las vistas materializadas, en vez de hacerlo a través de las tablas originalmente indicadas en la consulta.

La siguiente figura muestra el proceso que sigue Oracle para identificar cuándo es necesario utilizar una vista materializada de forma implícita. Cuando llega una consulta SQL, Oracle comprueba si esta consulta puede rescribirse utilizando vistas materializadas. De ser así, Oracle rescribe la consulta y calcula el plan de consulta óptimo para la nueva consulta SQL. Por otro lado, también calcula el plan de consulta óptimo para la consulta original. Finalmente, compara ambos planes de consulta y escoge el plan de consulta más eficiente.

Lectura complementaria

Más información sobre las vistas materializadas en Oracle y cómo funciona el proceso de reescritura de consultas en los capítulos 9 (vistas materializadas), 18 (reescritura de consultas simple) y 19 (reescritura de consultas avanzada) del libro *Oracle Database Warehousing Guide*: (http://docs.oracle.com/cd/E11882_01/server.112/e25554/toc.htm).

Figura 16. Método para escoger si se debe o no utilizar una vista materializada en la resolución de una consulta



Si queremos acceder a una vista materializada de forma implícita es necesario indicarlo en su definición añadiendo la cláusula `ENABLE QUERY REWRITE`. Por otro lado, el uso de la reescritura impone ciertas restricciones sobre la vista materializada que deberán ser comprobadas y tenidas en cuenta en cada caso.

4.2. Definición y uso de vistas materializadas en PostgreSQL

El uso de vistas materializadas en PostgreSQL no es tan completo como en Oracle. El motivo es que las vistas materializadas no fueron incluidas en PostgreSQL hasta el septiembre del 2013, en la versión 9.3.

La sintaxis de creación de vistas materializadas en PostgreSQL es:

```
CREATE MATERIALIZED VIEW name
    [(column_name [, ...])]
    [WITH (storage_parameter [= value] [, ... ])]
    [TABLESPACE tablespace_name]
    AS query
    [WITH [NO] DATA]
```

Los elementos principales en la creación de una vista materializada son los siguientes:

- `name`: nombre de la vista materializada.
- `(column_name [,...])`: permite definir el nombre de las columnas de la vista materializada.
- `Storage_parameter...`: permite indicar los parámetros físicos de la vista materializada.
- `TABLESPACE`: permite indicar el *tablespace* a utilizar.
- `Query`: indica la sentencia SQL que permite calcular la vista materializada.
- `WITH DATA` o `WITH NO DATA`: permite especificar si la vista materializada debe ser poblada (`WITH DATA`) o no (`WITH NO DATA`) en tiempo de creación.

Versión de PostgreSQL

La versión de PostgreSQL utilizada para realizar estos materiales es la 9.3. Es posible que elementos de diseño físico presentados en este documento se actualicen y añadan nuevas funcionalidades en versiones posteriores.

En PostgreSQL, las vistas materializadas se deben actualizar de forma manual. Por lo tanto, será necesario indicar mediante una sentencia SQL cuándo se deben actualizar los datos de la vista materializada. La sentencia SQL utilizada es la que se presenta a continuación.

```
REFRESH MATERIALIZED VIEW name
    [WITH [NO] DATA]
```

Por defecto, la sentencia actualiza los datos de la vista materializada a partir de la sentencia SQL indicada en su creación. La adición de nuevos datos no se hace de forma incremental, sino que se vacía la vista materializada y se vuelve a poblar de nuevo. Es posible añadir la cláusula `WITH NO DATA` al ejecutar la sentencia `refresh`. En este caso, los datos de la vista actualizada se eliminarán, pero esta no se poblará con nuevos datos.

El `refresh` de vistas materializadas en PostgreSQL realiza un bloqueo exclusivo y total de la vista materializada. Esto causa que esta no esté accesible mientras se actualiza. Esto es un inconveniente importante dado que limita el uso de las vistas materializadas en PostgreSQL, ya que la actualización se realiza de forma total (no es incremental) y que el tiempo de actualización puede ser considerable.

Por ejemplo, la siguiente sentencia crearía una vista materializada para almacenar las ventas de una empresa agrupadas por año y producto. La vista materializada se crearía y se poblaría en el momento de su creación.

```
-- Dadas las siguientes tablas de times, products y sales:
CREATE TABLE times (
```

```
   time_id          DATE     PRIMARY KEY,
   calendar_year    VARCHAR(50),
   calendar_month   VARCHAR(50),
   calendar_day     VARCHAR(50),
   ...
);

CREATE TABLE products (
   prod_id          NUMBER   PRIMARY KEY,
   prod_name        VARCHAR(50),
   prod_description VARCHAR(2000),
   ...
);

CREATE TABLE sales (
   prod_id          NUMBER,
   cust_id          NUMBER,
   time_id          DATE,
   quantity_sold    NUMBER(3),
   amount_sold      NUMBER(10,2),
   ...
);

-- Creamos una vista materializada que permite precalcular el total de ventas agrupado
por producto y año
CREATE MATERIALIZED VIEW sales_mv
   TABLESPACE tbs_materialized_views
   AS SELECT t.calendar_year, p.prod_id,
      SUM(s.amount_sold) AS sum_sales
      FROM times t, products p, sales s
      WHERE t.time_id = s.time_id AND p.prod_id = s.prod_id
      GROUP BY t.calendar_year, p.prod_id
   WITH DATA;
```

Respecto a cómo acceder a las vistas materializadas, PostgreSQL, de momento, solo permite acceder de forma explícita. Por lo tanto, para utilizar vistas materializadas en una consulta, es necesario incluir sus referencias explícitas en la consulta SQL.

Resumen

Los contenidos de este módulo han sido creados con el objetivo de introducir al lector en el mundo del diseño físico de las bases de datos del que es responsable el administrador de la base de datos.

El módulo empieza definiendo qué es el diseño físico, comentando qué efectos tiene un buen diseño físico en la base de datos y presentando los componentes de almacenaje de una base de datos.

A continuación se describen en detalle algunos de los elementos de diseño físico de bases de datos más significativos. Se empieza con los *tablespaces*, indicando cuál es su objetivo y cómo deben usarse. Se continua con los índices, describiendo los índices basados en árboles B+, en dispersión y en mapas de *bits*, introduciendo los conceptos de optimización y planes de consultas con el objetivo de mostrar al lector cómo evaluar el impacto de los índices en una base de datos, y presentando algunas reglas a tener en cuenta en el diseño de índices. Finalmente, se presenta el concepto de vista materializada, indicando cuál es su función y qué se debe tener en cuenta al crearlas. Todos los elementos de diseño físico presentados se describen primero de forma teórica y posteriormente se indica cómo usarlos en las últimas versiones de Oracle y PostgreSQL.

En caso de que algún lector quiera profundizar en el diseño físico de bases de datos, puede consultar las referencias que hemos ido indicando a lo largo del módulo o los libros indicados en el apartado de bibliografía.

Ejercicios de autoevaluación

1. Dada la siguiente tabla (clave primaria subrayada) que guarda información sobre diferentes centros de *fitness*:

Centro (nombre, dirección, población, m2)

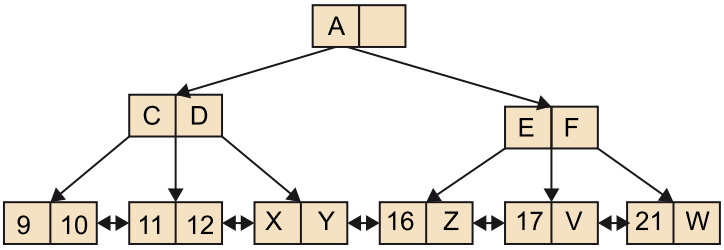
Decid qué tipo de acceso implica las consultas que se muestran a continuación:

1) `SELECT * FROM centro ORDER BY poblacion`

2) `SELECT * FROM centro WHERE m2>100 AND m2<150`

3) `SELECT FROM centro WHERE poblacion='Barcelona' AND m2>100`

2. Dado el siguiente árbol B+, de orden *d* igual a 1 (d=1), que corresponde a un índice definido sobre una tabla *T* para una columna de tipo entero que es clave primaria de la tabla *T*:



Se pide:

a) Indicad cuáles son los valores posibles para X, Y, Z, V y W.

b) Indicad cuáles son los valores posibles para C y D.

c) Indicad cuáles son los valores posibles para E y F.

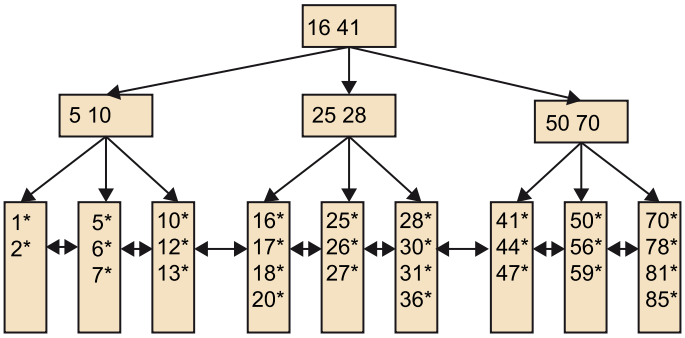
d) Indicad cuáles son los valores posibles para A.

e) Sabiendo que los valores a indexar sobre el árbol B+ ocupan 32 *bytes*, los apuntadores a registros de datos (RID) 4 *bytes*, los apuntadores a nodos 2 *bytes* y las páginas son de 4 Kb, indicad cuál sería el orden *d* más óptimo (el valor más grande posible para *d*) del árbol B+.

3. Tenemos una tabla que guarda datos sobre palacios de congresos con el siguiente esquema (clave primaria subrayada):

palacios_congresos(codigo_palacio, ciudad, importancia, capacidad)

Los datos de la tabla previa están almacenados en un solo fichero, que únicamente almacena datos de esa tabla. Este fichero consta de 10 páginas. Sobre la columna *codigo_palacio* (que es de tipo entero) se ha construido el siguiente árbol B+ de orden d=2 que se muestra en la siguiente figura. Cada nodo del índice constituye una página del fichero que guarda los datos del índice.



a) Suponiendo que el árbol B+ es no agrupado, decid si el árbol B+ ayuda o no a resolver eficientemente las siguientes consultas. Indicad también cuál será el coste mínimo y máximo (en número de operaciones de E/S) de resolución de las consultas.

1) `SELECT * FROM palacios_congresos WHERE codigo_palacio= 80`

2) `SELECT * FROM palacios_congresos WHERE codigo_palacio BETWEEN 20 AND 26`

3) `SELECT * FROM palacios_congresos WHERE ciudad='Barcelona' AND importancia>50`

b) Suponiendo ahora que el árbol B+ es agrupado, ¿cambiaríais alguna de las respuestas formuladas en el apartado anterior?

4. Supongamos que tenemos un índice de dispersión con la siguiente función de dispersión: h(X)=(X mod 3) + 1, donde X representa el valor a insertar. En cada página solo caben 2 valores, y disponemos de 3 páginas primarias.

a) Indicad cómo quedaría el índice si insertamos los siguientes valores: 5, 3, 47, 94, 123 y 214. Calculad el factor de carga tras la inserción del conjunto de valores.

b) ¿Qué solución propondríais si a continuación de los 6 valores anteriores tenemos que añadir 10 millones más de valores? Calculad el factor de carga asociado a la solución dada.

c) ¿Y si esta inserción de 10 millones de valores fuese solo temporal, para la realización de unos cálculos puntuales, y después se borraran el 75% de estos valores porque ya no son necesarios?

5. Supongamos que debemos gestionar una base de datos de venta de productos por Internet. La tabla de ventas de la base de datos tiene un número de filas importante y las consultas que se realizan sobre ella empiezan a demorarse en demasía. La definición de la tabla de ventas es la siguiente:

Sale(saleID, saleCostumer, salePrice, saleDate, saleCountry, saleDayOfWeek, salePaymentMethod, saleHour)

Donde *saleID* es la clave primaria, *saleCostumer* es el cliente que realizó la venta, *salePrice* indica el precio de la venta, *saleDate* indica la fecha en que se realizó la venta, *saleCountry* indica el país dónde se realizó la venta, *saleDayOfWeek* indica el día de la semana (Domingo, Lunes, Martes, etc.) en que se realizó la venta, *salePaymentMethod* indica el método de pago con que se realizó la venta y *saleHour* es un entero entre 1 y 24 que indica la hora en que se realizó la venta.

Nos piden lo siguiente:

a) Cread los índices de mapa de *bits* necesarios para que los accesos a la tabla de ventas sean lo más rápidos posibles. Hay que tener en cuenta los siguientes datos:

1) La tabla de ventas tiene 5 millones de filas.

2) Las consultas que se quieren optimizar son las que realiza el departamento de *marketing*, que son del siguiente tipo:

- Las 10 ventas más cuantiosas por país, por forma de pago o por día de la semana en un periodo de tiempo determinado.
- El número de ventas y cuantía de las ventas semanales en función del país de origen, de la forma de pago y del día de la semana.

3) La clave primaria y la columna *saleDate* ya están indexadas con un índice basado en árbol B+.

4) Hay una media de 20 ventas por minuto.

b) Suponed que la empresa cambia el SGBD por uno que es tan avanzado que permite realizar modificaciones en las columnas indexadas con índices de mapa de *bits* sin bloquear el índice.

1) ¿Qué índices crearíais bajo este supuesto?

2) ¿Cuánto ocuparía cada índice?

3) Indicad los valores de los índices creados para las siguientes filas de la tabla:

saleID	saleCostumer	salePrice	saleDate	saleCountry	saleDayOfWeek	salePaymentMethod	saleHour
54478	112	630	9/12/2013	España	Domingo	Credit card	02
54479	114	6	9/12/2013	Andorra	Domingo	PayPal	04
54480	115	3	9/12/2013	Portugal	Domingo	PayPal	05
54481	212	1200	9/12/2013	Alemania	Domingo	Credit card	05
54482	154	50	9/12/2013	China	Domingo	Check	05
54483	25	680	9/12/2013	Vietnam	Domingo	PayPal	07

e) Indicad cómo se utilizarían los índices creados en la ejecución de la siguiente consulta tomando solo como referencia los datos de la tabla anterior.

```
SELECT salePaymentMethod, count(*)
FROM sale
WHERE saleCountry IN ('China', 'Vietnam')
GROUP BY salePaymentMethod
```

6. Dadas las siguientes tablas (claves primarias subrayadas) de una base de datos del departamento postventa de una cadena de tiendas:

Clientes(codigo_cliente, nombre, edad, sexo)

Compras(num_compra, codigo_cliente, codigo_producto, codigo_tienda, fecha, cantidad)

codigo_producto es clave foránea a *Producto* y *codigo_tienda* es clave foránea a *Tienda*.

Incidencias_postventa(num_incidencia, codigo_producto, tipo_incidencia, fecha, resultado)

codigo_producto es clave foránea a *Producto* y *tipo_incidencia* es clave foránea a *Tipo_incidencia*. No se mantiene información del cliente que ha generado la incidencia. Las incidencias están tipificadas (*tipo_incidencia*), hay solo 10 incidencias diferentes y se encuentran descritas en la tabla *Tipo_incidencia*.

Las tablas *Tipo_incidencia*, *Producto* y *Tienda* no son relevantes para la resolución del ejercicio, por eso no se incluye su esquema. La tabla *Clientes* ocupa 100.000 páginas, la tabla *Compras* 2.000.000 de páginas y la tabla *Incidencias_postventa* ocupa 300.000 páginas. Los datos de cada tabla se almacenan en un solo fichero de datos que únicamente guarda datos de la tabla con la que se relaciona.

Después de observar las consultas de acceso por valor que se hacen, vemos que los criterios de selección más frecuentes son los siguientes:

a) 10% de consultas seleccionan respecto el valor de la columna *Clientes.codigo_cliente*.

b) 35% de consultas seleccionan respecto el valor de la columna *Clientes.nombre*.

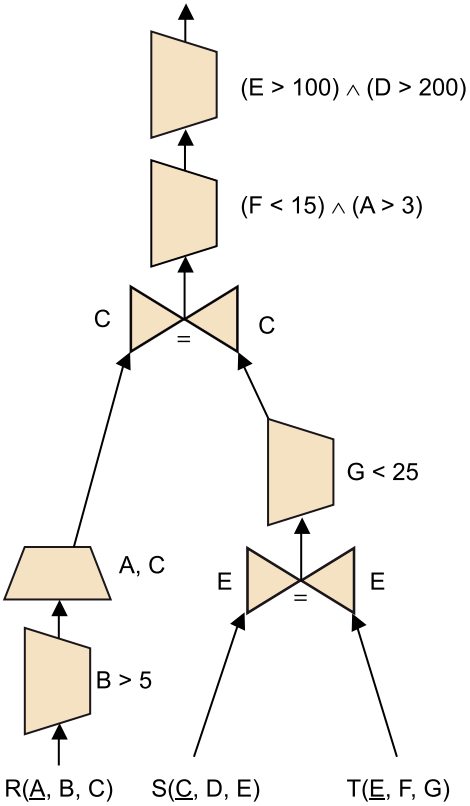
c) 40% de consultas seleccionan respecto el valor de la columna *Compras.codigo_producto*.

d) 5% de consultas seleccionan respecto el valor de la columna *Incidencias_postventa.tipo_incidencia*.

e) 10% de consultas seleccionan respecto el valor de la columna *Incidencias_postventa.fecha*.

Sabemos que sobre estas tablas no existe ningún índice definido, pero se quieren crear con el fin de acelerar las consultas más frecuentes que se hacen. Suponemos que disponemos de los recursos necesarios para construir dos índices en forma de árbol B+ no agrupado, y que con estos podremos alcanzar tiempos de respuesta inferiores, y significativamente mejores que los obtenidos mediante un recorrido secuencial. ¿Sobre qué columnas construiríais los dos índices?

7. Dado el siguiente árbol sintáctico, en el que las columnas subrayadas son las claves primarias de cada tabla, bajad las operaciones de selección tan abajo como sea posible en el árbol para obtener el árbol sintáctico óptimo:



8. Suponed que acabamos de crear la siguiente vista materializada en un SGBD PostgreSQL tal y como se muestra a continuación:

```
-- Dadas las siguientes tablas de times, products y sales:
CREATE TABLE times (
   time_id          DATE     PRIMARY KEY,
   calendar_year    VARCHAR(50),
   calendar_month   VARCHAR(50),
   calendar_day     VARCHAR(50),
   ...
);

CREATE TABLE products (
   prod_id          NUMBER   PRIMARY KEY,
   prod_name        VARCHAR(50),
   prod_description VARCHAR(2000),
   ...
);

CREATE TABLE sales (
   prod_id          NUMBER,
   cust_id          NUMBER,
   time_id          DATE,
   quantity_sold    NUMBER(3),
   amount_sold      NUMBER(10,2),
   ...
);

-- Creamos una vista materializada que permite precalcular el total de ventas agrupado
por producto y año
CREATE MATERIALIZED VIEW sales_mv
   TABLESPACE tbs_materialized_views
   AS SELECT t.calendar_year, p.prod_id,
      SUM(s.amount_sold) AS sum_sales
      FROM times t, products p, sales s
      WHERE t.time_id = s.time_id AND p.prod_id = s.prod_id
      GROUP BY t.calendar_year, p.prod_id
   WITH DATA;
```

Se pide:

a) Usad disparadores para hacer que la vista materializada se actualice cada vez que se añade/borra/modifica una fila en las tablas `times`, `products` y `sales`.

b) Indicad qué problemas podría conllevar la actualización de la vista que acabáis de implementar.

Solucionario

Ejercicios de autoevaluación

1. La primera consulta se trata de un acceso secuencial por valor dado que hay que recorrer todas las filas de la tabla centro considerando el valor de la columna población que nos permitirá su ordenación.

La segunda consulta se trata de un acceso secuencial por valor sobre la tabla centro, al tener que recorrer todas las filas contenidas en esta tabla tomando en consideración únicamente las que cumplen la condición de que sus m2 están entre 100 y 150.

La última consulta se trata de un acceso mixto sobre la tabla centro, donde se efectúa un acceso directo por valor sobre la columna población y un acceso secuencial por valor sobre la columna m2.

2. Se propone la siguiente solución para cada uno de los apartados planteados:

Según la definición de árbol B+, los valores en las hojas están ordenados y no hay repetidos, por lo tanto, $12 < X < Y < 16$. Partiendo de esta premisa, los elementos pueden tomar los siguientes valores:

- X=13, entonces Y = 14 o 15
- X=14, entonces Y = 15
- X=15, entonces Y podría no tener ningún valor asociado.
- Z no puede contener ningún valor puesto que tendría que ser $16 < Z < 17$.
- V puede ser 18, 19 o 20.
- W>21

Según la definición de nodo interno de un árbol B+, C>10 y C <=11, por lo tanto solo puede tomar el valor 11. De manera similar, D > 12 y D <=X, pero teniendo en cuenta que todos los valores de los nodos internos tienen que estar en las hojas D también tiene que estar en las hojas, tiene que ser igual a X. En resumen, C=11 y D=X (hay que hacer notar que en realidad D no puede ser menor que X).

Los valores de E y F, están determinados, solo pueden ser 17 y 21, por el mismo argumento expuesto para el elemento C.

El valor de A tiene que cumplir que tiene que ser menor que todos los valores de la derecha del árbol y además tiene que estar en las hojas. Por lo tanto, necesariamente tiene que ser 16.

Para saber el orden óptimo del árbol, tenemos que calcular los valores que cabrían en los nodos internos y nodos hojas:

- Nodos internos: $2d*32 + (2d+1)*2 = 4096 \Rightarrow d = 60$
- Nodos hoja: $2d*(32+4) +2*2 = 4096 \Rightarrow d = 56$

Por lo tanto el orden óptimo del árbol B+ es 56.

3. Se propone la siguiente solución para cada uno de los apartados planteados:

a) Se resolverán de forma eficiente la primera y la segunda de las consultas, que implican, respectivamente, un acceso directo por valor y un acceso secuencial por valor. En el caso de la última consulta el índice no resulta de utilidad ninguna, dado que el árbol B+ no indexa valores de ciudad ni de importancia. Con relación a los costes, estos serían:

1) El coste (mínimo y máximo) sería 3 operaciones de E/S, que corresponde a la lectura de 3 nodos del árbol B+ (el último sería el nodo hoja que debería contener la entrada buscada). Como no existe el palacio con código 80, no se desencadena ninguna operación de E/S en el fichero que contiene los datos de los palacios de congresos.

2) El coste mínimo será de 5 operaciones de E/S: 4 en el índice en forma de árbol B+ para recuperar las entradas de interés más 1 en el fichero que contiene los datos de la tabla de palacios de congresos (asumimos que todas las filas buscadas, 3 en total, se encuentran en la misma página). Por su parte el coste máximo será de 7 operaciones de E/S: 4 en el árbol B+ más 3 en el fichero que contiene los datos de la tabla (asumimos que cada una de las filas que conforma el resultado de la consulta está en una página diferente del fichero de datos).

3) El coste (mínimo y máximo) de la última consulta será de 10 operaciones de E/S, dado que para resolver la consulta es necesario recorrer todas las páginas del fichero que contiene los datos de los palacios de congresos.

b) Cambiaría la respuesta a la consulta número 2 que se vería beneficiada. Dado que el árbol B+ es agrupado, los datos de los palacios de congresos están almacenados físicamente en el fichero de datos según el valor de la columna sobre la cual se ha construido el índice. El coste (mínimo y máximo) sería de 4 o 5 operaciones de E/S, 3 en el árbol B+ (debido a la ordenación física del fichero de datos solo es necesario acceder al primer nodo hoja con entradas de interés) más 1 o 2 en el fichero de datos (1 si las 3 filas a recuperar están en una única página o 2 si están en páginas consecutivas, dado que podemos afirmar que si el fichero de datos tiene 10 páginas, en promedio, hay entre 2 y 3 filas en cada página).

4. A continuación se proporcionan las respuestas a cada uno de los apartados planteados:

a) Obtendríamos el siguiente resultado:

		L	
	1	3	123
N	2	94	214
	3	5	47

El factor de carga C sería:

C = M/(N*L)
C = 6 valores/(3 páginas*2 valores en cada pág.) = 1

Por lo tanto, el factor de carga es 1.

b) Una primera aproximación a la solución nos podría llevar a pensar que todos estos valores podrían gestionarse mediante páginas de excedentes, pero eso no tendría sentido, tendríamos que gestionar millones de filas en páginas de excedentes, lo que comportaría un significativo número de accesos (operaciones de E/S) a las páginas de excedentes que haría ineficiente este índice basado en la dispersión.

Por lo tanto, la solución es reformular la función de dispersión, que equivale a cambiar el número de páginas primarias a considerar, una vez fijada una C objetivo (de lo contrario no podríamos aislar la N):

C = 10.000.006 /(N*2)

Donde N es el número de páginas primarias, y el nuevo intervalo [1,...,N] que resulta de la nueva función de dispersión.

c) En este caso donde el índice crece y disminuye, está claro que un modelo de dispersión estática no nos sirve y tendríamos que ir a un modelo de índice de dispersión dinámica (*linear hashing* o *extendible hashing*).

5. A continuación se presenta la solución al cada uno de los supuestos planteados:

a) No se crearía ningún índice de tipo mapa de *bits* ya que la tabla tiene un gran número de inserciones. En caso de crear algún índice de mapa de *bits* sobre la tabla, cada inserción bloquearía el índice, evitando que otras operaciones concurrentes pudieran acceder al mismo hasta que la inserción acabara. Eso podría dar problemas en un sistema que requiere procesar operaciones en tiempo real.

b) Si no existe la restricción comentada en el apartado anterior entonces se crearían índices de mapa de *bits* sobre las siguientes columnas:

saleCountry: ya que tiene un número de valores limitado y es utilizado en las operaciones de búsqueda.

saleDayOfWeek: ya que solo tiene siete posibles valores que se repetirán en muchas filas y es utilizado en las operaciones de búsqueda.

salePaymentMethod: ya que tiene pocos valores y se utiliza en las operaciones de búsqueda comentadas.

c) Los índices creados ocuparían:

saleCountry: la tienda realiza ventas por Internet, por lo tanto pueden comprar desde cualquier país del mundo. Según la ONU existen 193 países en el mundo. Por lo tanto, el tamaño del índice será: número de posibles valores (193) x número de filas (5.000.000) = 965.000.000 *bits* = 115 *megabytes* aproximadamente.

saleDayOfWeek: 7 posibles valores*5.000.000 = 35.000.000 *bits*, unos 4 *megabytes*.

salePaymentMethod: suponiendo que solo hay 3 posibles valores (*PayPal*, *credit card*, *check*) sería: 3 * 5.000.000 = unos 2 *megabytes*.

d) Los índices creados tendrían los siguientes valores en la tabla de referencia:

Índice sobre *saleCountry*

España	Andorra	Alemania	China	Vietnam
1	0	0	0	0
0	1	0	0	0
1	0	0	0	0
0	0	1	0	0
0	0	0	1	0
0	0	0	0	1

Índice sobre *saleDayOfWeek*

Domingo	Lunes	Martes	Miércoles	Jueves	Viernes	Sábado
1	0	0	0	0	0	0
1	0	0	0	0	0	0
1	0	0	0	0	0	0
1	0	0	0	0	0	0
1	0	0	0	0	0	0
1	0	0	0	0	0	0

Índice sobre *salePaymentMethod*

Credit card	PayPal	Check
1	0	0
0	1	0
0	1	0
1	0	0
0	0	1
0	1	0

e) Para resolver la consulta de referencia primero se realizaría el filtro de la cláusula `WHERE`. Para identificar las filas que representan ventas de China y Vietnam se realizaría una opera-

ción OR lógica entre los dos últimos mapas de *bits* del índice sobre *saleCountry*. Podemos ver el resultado en las siguientes tablas:

China	Vietnam
0	0
0	0
0	0
0	0
1	0
0	1

China OR Vietnam
0
0
0
0
1
1

De la operación resultante podemos comprobar que las únicas filas que nos interesan son las dos últimas. Para realizar la agrupación se pueden utilizar los mapas de *bits* del índice sobre *salePaymentMethod*. La manera de hacerlo sería realizando una operación AND lógica entre el resultado anterior (*China OR Vietnam*) y el mapa de *bits* de cada grupo de la agrupación, donde hay un grupo por cada valor de la columna *salePaymentMethod*. Podemos ver el resultado de ejecutar la operación sobre cada agrupación a continuación:

Credit card	PayPal	Check	China OR Vietnam
1	0	0	0
0	1	0	0
0	1	0	0
1	0	0	0
0	0	1	1
0	1	0	1

(China OR Vietnam) AND Credit Card	(China OR Vietnam) AND PayPal	(China OR Vietnam) AND Check
0	0	0
0	0	0
0	0	0
0	0	0
0	0	1
0	1	0

De los resultados obtenidos podemos ver que no hay ninguna fila que satisfaga las condiciones en el primer grupo (*Credit card*), que en el segundo grupo (*PayPal*) hay solo una fila (la última), y que en el tercer grupo (*Check*) hay una sola fila (la penúltima).

6. Las ventajas que se obtienen por el hecho de tener implementado un índice sobre una columna son más significativas cuanto mayor es el número de páginas de la tabla a consultar. Este factor, sin embargo, también se tiene que combinar con la frecuencia de ejecución de cada consulta.

Teniendo en cuenta eso, y el hecho de que solo podemos crear dos índices, rápidamente podemos ver que tenemos un claro candidato: la consulta sobre la tabla *Compras* es la que más veces se ejecuta y es, además, la tabla más voluminosa (2.000.000 de páginas) con mucha diferencia respecto del resto de tablas.

Si miramos el resto de consultas, veremos que 2 actúan sobre *Clientes* y 2 sobre *Incidencias_postventa*. Si decidimos crear un índice sobre clientes lo haremos sobre la columna *nombre*, dado que es la consulta que, sobre la tabla *Clientes*, se ejecuta más veces. Si creamos el índice sobre *Incidencias_postventa*, por motivos similares a los anteriores, lo crearemos sobre la columna *fecha*.

Para acabar de decidirnos entre el índice sobre *Clientes.nombre* o sobre *Incidencias_postventa.fecha*, tenemos que hacer cálculos. Si no existe ningún índice, las consultas solo se podrán resolver haciendo recorridos secuenciales de los ficheros que almacenan las tablas. En el caso peor, para resolver las consultas, habrá que recorrer todas las páginas de estos ficheros. De cada 100 consultas, 35 actúan sobre *Clientes.nombre* y 10 actúan sobre *Incidencias_postventa.fecha*. Por lo tanto, el número de páginas accedidas en cada caso será:

Caso (b) consultas que seleccionan *Clientes.nombre*

100.000 páginas * 35 = 3.500.000 páginas

Caso (e) consultas que seleccionan *Incidencias_postventa.fecha*

300.000 páginas * 10 = 3.000.000 páginas

Por lo tanto los dos índices a crear serán: un primer índice sobre *Compras.codigo_producto* y un segundo índice sobre *Clientes.nombre*

Si hacemos los cálculos para todas las consultas, la elección es, obviamente, la misma:

Caso (a) consultas que seleccionan *Clientes.codigo_cliente*

100.000 páginas * 10 = 1.000.000 páginas

Caso (b) consultas que seleccionan *Clientes.nombre*

100.000 páginas * 35 = 3.500.000 páginas

Caso (c) consultas que seleccionan *Compras.codigo_producto*

2.000.000 páginas * 40 = 80.000.000 páginas

Caso (d) consultas que seleccionan *Incidencias_postventa.tipo_incidencia*

300.000 páginas * 5 = 1.500.000 páginas

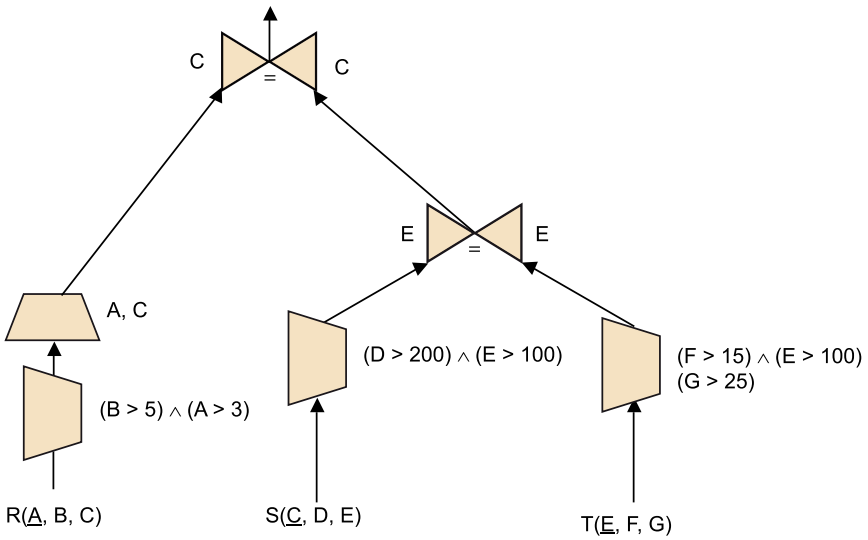
Caso (e) consultas que seleccionan *Incidencias_postventa.fecha*

300.000 páginas *10 = 3.000.000 páginas

7. El árbol resultante es el óptimo después de haber bajado las operaciones de selección $(A > 3) \wedge (D > 200) \wedge (E > 100) \wedge (F > 15) \wedge (G > 25)$. Analizando cada una de las restricciones de manera independiente podemos afirmar que:

- La restricción A>3 puede bajar por la rama de la tabla R, ya que es una columna que pertenece únicamente a dicha tabla.
- La restricción D>200 puede bajar por la rama de la tabla S, ya que es una columna que pertenece únicamente a dicha tabla.
- La restricción F>15 puede bajar por la rama de la tabla T, ya que es una columna que pertenece únicamente a dicha tabla.
- La restricción G>25 puede bajar por la rama de la tabla T, ya que es una columna que pertenece únicamente a dicha tabla.
- La restricción E>100 tiene que bajar por las ramas de las tablas S y T, ya que afecta a ambas tablas. Dichas restricciones se pueden agrupar con las bajadas en los pasos anteriores.

El árbol resultante es el que se muestra a continuación:



8. Las respuestas al ejercicio planteado son las siguientes:

a) Primero se crearía una función que forzara la actualización de la vista materializada y posteriormente se crearía un disparador para cada tabla, indicando que al añadir/modificar/eliminar una fila sobre las tablas afectadas se debe ejecutar la función que actualiza la vista materializada:

```
CREATE OR REPLACE FUNCTION refreshMateralizationViewSales()
```

```
   Returns trigger As '
Begin
   Refresh Materialized View materializedEmployeeView;
   Return NULL;
End

LANGUAGE plpgsql;

Create Trigger refreshMaterializedViewOnTimes
After INSERT Or UPDATE Or DELETE
On times For Each Row
   Execute Procedure refreshMateralizationViewSales();

Create Trigger refreshMaterializedViewOnProducts
After INSERT Or UPDATE Or DELETE
On products For Each Row
   Execute Procedure refreshMateralizationViewSales();

Create Trigger refreshMaterializedViewOnSales
After INSERT Or UPDATE Or DELETE
On sales For Each Row
   Execute Procedure refreshMateralizationViewSales();
```

b) Tal y como se ha comentado en los materiales, actualmente PostgreSQL realiza cargas completas de las vistas materializadas, realizando bloqueos de la vista entera en cuanto se actualiza. Si hubiera una frecuencia muy alta de actualización de las tablas afectadas (`times`, `sales`, `products`) podría resultar en un bloqueo permanente de la vista. Eso haría imposible su consulta, mientras que supondría un gran derroche de recursos (CPU y memoria RAM).

Glosario

árbol sintáctico de una consulta *m* Representación interna en forma de árbol de una consulta, donde las hojas del árbol representan las tablas implicadas en la consulta, los nodos intermedios corresponden a las operaciones pedidas en la consulta y la raíz del árbol es la respuesta a la consulta formulada.

árbol B+ *m* Estructura de datos que se utiliza para organizar índices que permiten implementar el acceso directo y el acceso secuencial por valor.

arquitectura de componentes de almacenaje *f* Esquema de tres niveles, lógico, físico y virtual, con el cual se clasifica y se describe cada componente de los SGBD, especialmente los que están relacionados con el almacenaje de los datos.

dispersión *f* Manera de organizar valores que se puede utilizar en índices que implementan el acceso directo por valor.

entrada *f* Elemento de un índice que consiste, en el caso más simple, en una pareja formada por valor e identificador (o dirección física, RID) de la fila que contiene dicho valor.

espacio virtual *m* Secuencia de páginas virtuales. Proporciona al SGBD una visión simplificada del nivel físico.

extensión *f* Unidad de asignación de espacio en un dispositivo de almacenamiento no volátil. Cada extensión es un número entero de páginas consecutivas y está contenida dentro de un fichero.

factores de coste *m pl* Aspectos que influyen en el coste total de ejecución de una consulta.

fichero *m* Unidad de gestión del espacio en los dispositivos de almacenaje no volátil. Está conformado por una o más extensiones. Normalmente el SO gestiona los ficheros en lugar del SGBD.

índice *m* Estructura de datos que los SGBD utilizan para facilitar las búsquedas necesarias a fin de implementar los accesos por valor a uno o diversos valores.

índice agrupado *m* Índice que garantiza que los datos indexados también están ordenados físicamente según el orden del acceso por valor (directo o secuencial) que proporcionan.

mapa de *bits* *m* Secuencia de *bits* que indica qué filas de una tabla satisfacen una condición. Habrá un *bit* por cada fila de la tabla, y su valor (0,1) indicará si la fila cumple la condición especificada o no.

memoria interna *f* Espacio de la memoria principal del ordenador, normalmente bastante grande, dedicado a contener las páginas que gestiona el SGBD.

método de acceso *m* Algoritmo que permite consultar índices y tablas en búsqueda de los datos requeridos.

nivel físico *m* Nivel que engloba los componentes físicos (fichero, extensión y página) dentro de la arquitectura de componentes de almacenaje.

nivel lógico *m* Nivel que engloba los componentes lógicos (tablas, vistas, restricciones, etc.) dentro de la arquitectura de componentes de almacenaje.

nivel virtual *m* Nivel que engloba el componente virtual dentro de la arquitectura de componentes de almacenaje.

operación de entrada/salida *f* Proceso que permite el transporte de páginas entre la memoria interna y la memoria externa del ordenador. En cada operación de entrada/salida (E/S) se transfiere un cierto número de páginas. En general, para simplificar los cálculos de coste, se asume que en cada operación de E/S se transfiere una página.

optimización de consultas *f* Proceso que lleva a cabo el SGBD destinado a determinar cuál es la mejor estrategia de ejecución de una consulta.

optimización física de una consulta *f* Proceso que evalúa el coste total de ejecución de una consulta y determina qué algoritmos o métodos de acceso tienen que utilizarse para resolver las diferentes operaciones de la consulta, teniendo en cuenta las características de las estructuras de almacenaje y de los caminos de acceso (índices) que se han definido para acceder a los datos.

optimización sintáctica de una consulta *f* Proceso que determina cuál es el orden óptimo de ejecución de las operaciones de álgebra relacional incluidas en una consulta.

página *f* Unidad mínima de acceso y de transferencia de datos entre la memoria interna (o principal o intermedia) del ordenador y la memoria externa (no volátil) que contiene los ficheros de una base de datos. El SO de la máquina lleva a cabo esta transferencia y pasa la página al SGBD para que gestione su contenido. La página también es la estructura que permite al SGBD organizar los datos de una base de datos. En el área de SO la página recibe el nombre de bloque.

partición *f* Sistema de almacenamiento que permite distribuir los datos de una tabla en distintos espacios físicos para aumentar la eficiencia del sistema.

plan de acceso de una consulta *f* Estrategia de ejecución de una consulta que tiene asociado un coste mínimo.

vista materializada *f* Vista de la base de datos que contiene el resultado de una consulta precalculada y se almacena en el dispositivo de almacenamiento no volátil (en general, disco). Se utiliza para aumentar el tiempo de respuesta en operaciones comunes y costosas a cambio de aumentar el espacio de almacenamiento.

Bibliografía

Abelló, A.; Rollón, E.; Rodríguez, M. E. (2008). *Database Design and Administration. Computer Science Engineering. Aula Politècnica* (núm. 131). Edicions UPC.

Cabré, B. (2004). *Diseño físico de bases de datos*. Material docente UOC asignatura Sistemas de gestión de bases de datos. Eureca Media, S. L.

Costal, D. (2002). *Implementación de métodos de acceso*. Material docente UOC asignatura Bases de datos II. Eureca Media, S. L.

Lightstone, S.; Teorey, T.; Nadeau, T. (2007). *Physical Database Design* (3ª ed.). Morgan Kaufmann.

Liu, L.; Özsu, M. T. (Eds.) (2009). *Encyclopedia of Database Systems*. Springer.

PostgreSQL. Manuales accesibles en línea desde: http://www.PostgreSQL.org/docs/. Último acceso diciembre de 2013.

Oracle. Manuales accesibles en línea desde: http://www.oracle.com/technetwork/indexes/documentation/index.html. Último acceso diciembre de 2013.

Ortego, S. (2004). *El componente de procesamiento de consultas y peticiones SQL*. Material docente UOC asignatura Sistemas de gestión de bases de datos. Eureca Media, S. L.

Ramakrishan, R.; Gehrke, J. (2003). *Database Management Systems* (3ª ed.). Mc Graw Hill.

Rodríguez, M. E. (2002). *Otros temas de bases de datos*. Material docente UOC asignatura Bases de datos II. Eureca Media, S. L.

Segret, R. (2002). *Componentes de almacenaje de una base de datos*. Material docente UOC asignatura Bases de datos II. Eureca Media, S. L.