CAPÍTULO 15

Algoritmos para procesamiento y optimización de consultas

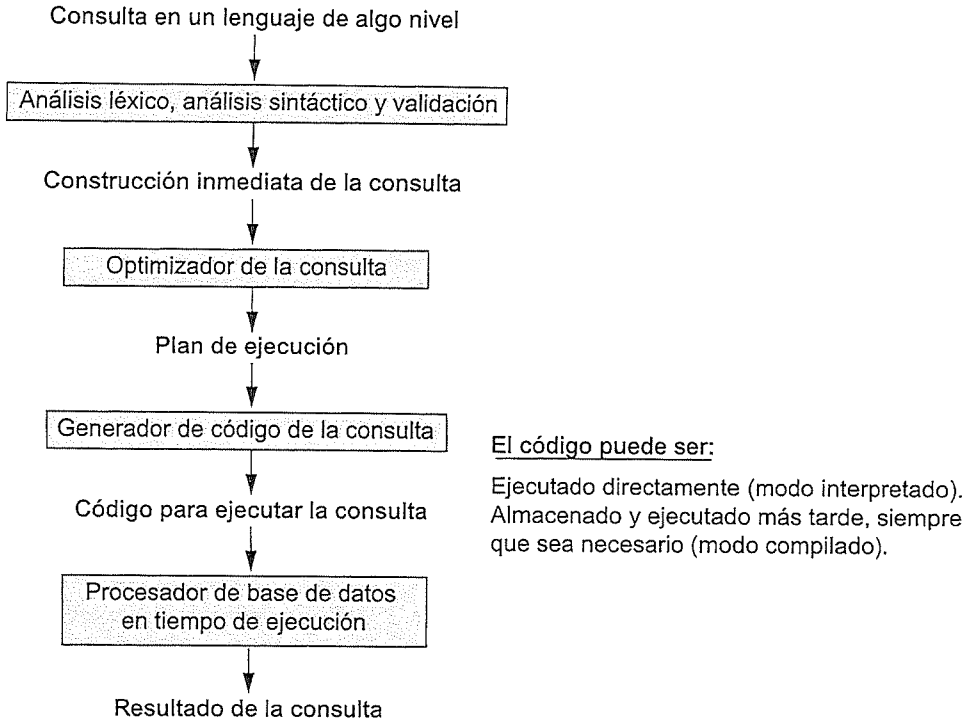
En este capítulo revisaremos las técnicas utilizadas por un DBMS para procesar, optimizar y ejecutar consultas de alto nivel. Una consulta expresada en un lenguaje de alto nivel como SQL debe, en una primera fase, ser explorada, analizada sintácticamente y validada. El **analizador léxico** identifica los elementos del lenguaje como, por ejemplo, las palabras reservadas de SQL, los nombres de atributos y los nombres de las relaciones, en el texto de la consulta, mientras que el **analizador sintáctico** comprueba la sintaxis de la consulta para determinar si ha sido formulada con arreglo a las reglas de sintaxis (las reglas de la gramática) del lenguaje de consultas. La consulta debe ser también **validada**, comprobando que todos los nombres de atributos y de relaciones son válidos y tienen significado semántico dentro del esquema de la base de datos en particular sobre la cual se realiza la consulta.[1] A continuación, se creará una representación interna de la consulta; normalmente mediante una estructura de datos en árbol denominada **árbol de consultas**. También es posible representar la consulta utilizando una estructura de datos en grafo denominada **grafo de consulta**. Seguidamente, el DBMS debe desarrollar una estrategia de ejecución para obtener el resultado de la consulta a partir de los ficheros de la base de datos. Lo habitual es que en una consulta se disponga de muchas estrategias distintas de ejecución; el proceso de elección de la estrategia más adecuada se conoce como **optimización de consultas**.

La Figura 15.1 muestra los distintos pasos a seguir en la ejecución de una consulta de alto nivel. El **optimizador de consultas** se encarga de generar un plan de ejecución y el **generador de código** se encarga de generar el código para ejecutar dicho plan. El **procesador de base de datos en tiempo de ejecución** tiene como cometido la ejecución del código, bien en modo compilado o bien en modo interpretado, para generar el resultado de la consulta. Si se produce un error en tiempo de ejecución el procesador de base de datos en tiempo de ejecución generará un mensaje de error.

El término optimización resulta, en realidad, inapropiado ya que en algunos casos el plan de ejecución elegido no resulta ser la estrategia más optima (se trata solo de una *estrategia razonablemente eficiente* para ejecutar la consulta). Por lo general, la búsqueda de la estrategia más optima consume demasiado tiempo excepto en el caso de las consultas más sencillas y es posible que se necesite información sobre cómo están construidos los ficheros e incluso sobre el contenido de los mismos (información que quizá no esté disponible por

[1] No discutiremos aquí las fases de análisis léxico y comprobación de sintaxis del procesamiento de una consulta, ya que estos temas se tratan en los textos relacionados con los compiladores.

Figura 15.1. Pasos habituales al procesar una consulta de alto nivel.



completo en el catálogo del DBMS). Por tanto, la *planificación de una estrategia de ejecución* puede llegar a requerir una descripción más detallada que en el caso de la *optimización de consultas*.

En el caso de los lenguajes de navegación por bases de datos de nivel más bajo (como, por ejemplo, DML en red o HDML jerárquico [consulte los Apéndices D y E]), el programador debe elegir la estrategia de ejecución de consultas a la hora de escribir un programa de bases de datos. Si un DBMS proporciona únicamente un lenguaje de navegación, las *necesidades u ocasiones de optimización* por parte del DBMS resultan ser bastante limitadas; en lugar de esto, al programador se le da la posibilidad de elegir la estrategia de ejecución *óptima*. Por otra parte, un lenguaje de consultas de alto nivel (como SQL para DBMS relacionales [RDBMS] u OQL [consulte el Capítulo 21] para DBMS orientadas a objetos [ODBMS]), es más expresivo por su propia naturaleza ya que especifica cuáles son los resultados esperados de la consulta en lugar de identificar los detalles sobre *cómo* se debería obtener el resultado. Según lo anterior, la optimización de consultas sí resulta necesaria en el caso de consultas especificadas en un lenguaje de alto nivel.

Nos concentraremos en describir la optimización de consultas en el contexto de un RDBMS, ya que muchas de las técnicas que describiremos han sido adaptadas para ODBMS.[2] Un DBMS relacional debe evaluar de forma sistemática las distintas estrategias de ejecución de consultas y elegir una estrategia razonablemente eficiente u óptima. Todos los DBMS disponen por lo general de varios algoritmos genéricos de acceso a bases de datos que implementan operaciones relacionales como SELECT o JOIN o combinaciones de ambas. Sólo las estrategias de ejecución que pueden ser implementadas por los algoritmos de acceso del DBMS y que se aplican a una consulta en particular y a un diseño de base de datos en particular son las que pueden ser tomadas en consideración por el módulo de optimización de consultas.

[2] Existen algunas técnicas y problemas de optimización de consultas que sólo pertenecen al ámbito de los ODBMS. Sin embargo, no los discutiremos aquí ya que sólo estamos ofreciendo una introducción a la optimización de consultas.

En la Sección 15.1 comenzaremos con una visión general sobre cómo se traducen habitualmente las consultas SQL a consultas de álgebra relacional para, posteriormente, ser optimizadas. A continuación, en las Secciones 15.2 a 15.6 veremos los algoritmos utilizados en la implementación de operaciones relacionales. Posteriormente, ofreceremos una visión general de las estrategias de optimización de consultas. Existen dos técnicas principales para la construcción de las optimizaciones de consultas. La primera técnica se basa en **reglas heurísticas** para la ordenación de las operaciones a realizar en una estrategia de ejecución de una consulta. Una regla heurística es una regla que funciona bien en la mayoría de los casos, aunque no está garantizado que funcione correctamente en todos los casos. Normalmente, las reglas reordenan las operaciones en un árbol de consultas. La segunda de las técnicas implica una **estimación sistemática** del coste de las diferentes estrategias de ejecución y la elección del plan de ejecución con la estimación de coste más baja. Normalmente, estas técnicas se combinan en un optimizador de consultas. Discutiremos la optimización heurística en la Sección 15.7 y la estimación de costes en la Sección 15.8. Posteriormente, en la Sección 15.9 haremos un breve repaso de los factores a tener en cuenta durante la optimización de consultas en el RDBMS comercial de Oracle. La Sección 15.10 presenta la idea de la optimización semántica de consultas, en la cual se utilizan restricciones conocidas para desarrollar estrategias eficaces de ejecución de consultas.

15.1 Traducción de consultas SQL al álgebra relacional

En la práctica, SQL es el lenguaje de consultas que se utiliza en la mayoría de los RDBMS comerciales. Una consulta SQL se traduce, en primer lugar, a una expresión extendida equivalente de álgebra relacional (representada como una estructura de datos de árbol de consultas) que es optimizada posteriormente. Por regla general, las consultas SQL se descomponen en **bloques de consulta** que forman las unidades básicas que se pueden traducir a los operadores algebraicos y después ser optimizadas. Un bloque de consulta contiene una única expresión SELECT-FROM-WHERE, y también cláusulas GROUP BY y HAVING en el caso de que éstas formen parte del bloque. Por tanto, las consultas anidadas dentro de una consulta se identifican como bloques de consulta independientes. Debido a que SQL incluye operadores de agregación (como MAX, MIN, SUM y COUNT), estos operadores deben ser también incluidos en el álgebra extendida, según vimos en la Sección 6.4.

Observe la siguiente consulta SQL sobre la tabla EMPLEADO de la Figura 5.5:

```
SELECT   Apellido1, Nombre
FROM     EMPLEADO
WHERE    Sueldo > ( SELECT   MAX (Sueldo)
                    FROM     EMPLEADO
                    WHERE    Dno=5 );
```

Esta consulta incluye una subconsulta anidada y, por tanto, debería ser descompuesta en dos bloques. El bloque interno es:

```
(SELECT   MAX (Sueldo)
 FROM     EMPLEADO
 WHERE    Dno=5 )
```

y el bloque externo es:

```
SELECT   Apellido1, Nombre
FROM     EMPLEADO
WHERE    Sueldo > c
```

donde c representa el resultado devuelto por el bloque interno. Podríamos traducir el bloque a la expresión extendida de álgebra relacional:

$\mathcal{T}$MAX Sueldo($\sigma_{Dno=5}$(EMPLEADO))

y el bloque externo a la expresión:

$\pi_{Apellido1,Nombre}(\sigma_{Sueldo>c}(EMPLEADO))$

El *optimizador de consultas* seleccionaría a continuación un plan de ejecución para cada bloque. Deberíamos observar que en el ejemplo anterior el bloque interno necesita ser evaluado sólo una vez para obtener el sueldo máximo que será utilizado posteriormente (como constante c) por el bloque externo. En el Capítulo 8 habíamos llamado a esto una *consulta anidada no correlacionada* . Es mucho más difícil optimizar las *consultas anidadas correlacionadas* ya que son más complejas (consulte la Sección 8.5), debido a que una variable de tupla del bloque externo aparece en la cláusula WHERE del bloque interno.

15.2 Algoritmos para ordenación externa

La ordenación es uno de los algoritmos principales utilizados en el procesamiento de las consultas. Por ejemplo, siempre que en una consulta SQL se especifique una cláusula ORDER BY, el resultado de la consulta debe ser ordenado. La ordenación es también un componente clave en los algoritmos de ordenación y mezclado utilizados en las operaciones de JOIN y en algunas otras (como UNION y INTERSECTION), y en los algoritmos de eliminación de duplicados de la operación PROYECTO (cuando en una consulta SQL se especifica la opción DISTINCT en la cláusula SELECT). En esta sección veremos uno de estos algoritmos. Es posible evitar la ordenación si existe el índice adecuado que permita un acceso ordenado a los registros.

La **ordenación externa** tiene relación con los algoritmos de ordenación que son adecuados para los ficheros de tamaño grande con registros almacenados en disco que no caben en su totalidad en la memoria principal, como sucede con la mayoría de los ficheros de bases de datos.[3]

El **algoritmo de ordenación externa** más habitual es el que utiliza una **estrategia de ordenación-mezcla**, que comienza con la ordenación de pequeños subficheros (denominados **porciones**) del fichero principal y, a continuación, mezcla esos subficheros ordenados para crear subficheros ordenados más grandes que, a su vez, serán mezclados nuevamente. El algoritmo de ordenación-mezcla, al igual que los algoritmos de bases de datos, necesita de un *espacio temporal* en memoria principal donde se realiza la ordenación y la mezcla de las porciones. El algoritmo básico, descrito en la Figura 15.2, consta de dos fases: la fase de ordenación y la fase de mezclado.

Durante la **fase de ordenación**, las porciones del fichero que caben en el espacio temporal disponible se leen en memoria principal, se ordenan utilizando un algoritmo de ordenación interna y se vuelven a escribir en disco en forma de subficheros temporales ordenados (o porciones). El tamaño de una porción y el **número inicial de porciones (n_R)** viene marcado por el **número de bloques de fichero (b)** y el **espacio temporal disponible (n_B)**. Por ejemplo, si $n_B = 5$ bloques y el tamaño del fichero es $b = 1.024$ bloques, entonces $n_R = \lceil (b/n_B) \rceil$ o 205 porciones iniciales, cada una de ellas con 5 bloques (excepto la última que tendrá 4 bloques). Por tanto, tras la fase de ordenación se almacenarán 205 porciones ordenadas en disco como ficheros temporales.

En la **fase de mezclado**, las porciones ordenadas se mezclan en una o más **pasadas**. El **grado de mezclado (d_M)** es el número de porciones que pueden ser mezcladas en cada pasada. Se necesita un bloque de espacio temporal en cada pasada para almacenar un bloque de cada una de las porciones que están siendo mezcladas y se necesita otro bloque que contenga un bloque del resultado de la mezcla. Según esto, d_M es el valor más pequeño entre $(n_B - 1)$ y n_R, y el número de pasadas es $\lceil (\log_{dM}(n_R)) \rceil$. En nuestro ejemplo, $d_M = 4$ (mezcla en cuatro pasadas); por tanto, las 205 porciones ordenadas iniciales se mezclarían y se quedarían en 52 al final de la primera pasada. Éstas serían mezcladas nuevamente quedando 13, después 4, y después 1 porción, lo

[3] *Los algoritmos de ordenación interna* son adecuados para la ordenación de estructuras de datos que caben en su totalidad en memoria.

Figura 15.2. Descripción del algoritmo de ordenación y mezcla en la ordenación externa.

```
set   i ← 1;
      j ← b;                    {tamaño en bloques del fichero}
      k ← nB;                   {tamaño en bloques del espacio temporal}
      m ← ⌈(j/k)⌉;

{Fase de ordenación}
while (i ≤ m)
do {
      leer los siguientes k bloques del fichero en el búfer o, si quedan menos de k bloques,
          leer los bloques restantes;
      ordenar los registros en el búfer y escribirlos como subfichero temporal;
      i ← i + 1;
}

{Fase de mezcla: mezclar los subficheros hasta que sólo quede 1}
set   i ← 1;
      p ← ⌈log(k-1)m⌉;          {p es el número de pasadas en la fase de mezcla}
      j ← m;
while (i ≤ p)
do {
      n ← 1;
      q ← ⌈(j/(k-1))⌉;          {número de subficheros a escribir en esta pasada}
      while (n ≤ q)
      do {
          leer los siguientes k-1 subficheros o los subficheros restantes (de pasadas previas),
                un bloque cada vez;
          mezclar y escribir como nuevo subfichero un bloque cada vez;
          n ← n + 1;
      }
      j ← q;
      i ← + 1;
}
```

que significa que se necesitan *cuatro pasadas*. El valor mínimo de 2 para d_M corresponde al caso peor de la ejecución del algoritmo, que es:

$$(2 * b) + (2 * (b * (\log_2 b))).$$

El primer término representa el número de accesos a bloques durante la fase de ordenación, ya que a cada bloque del fichero se accede dos veces: una vez para ser leído a memoria y otra vez para escribir nuevamente los registros en disco una vez ordenados. El segundo término representa el número de accesos a bloques durante la fase de mezclado, suponiendo el peor caso en el que d_M vale 2. En general, el logaritmo se toma en base d_M y la expresión que indica el número de bloques accedidos queda de este modo:

$$(2 * b) + (2 * (b * (\log_{dM} n_R))).$$

15.3 Algoritmos para las operaciones SELECT y JOIN

15.3.1 Implementación de la operación SELECT

Existen numerosas opciones a la hora de ejecutar una operación SELECT; algunas dependen de las rutas de acceso específicas al fichero y quizá sólo se apliquen a determinados tipos de condiciones de selección. En esta sección veremos algunos de los algoritmos para la implementación de SELECT. Utilizaremos las siguientes operaciones, que aparecen en la base de datos relacional de la Figura 5.5, para ilustrar nuestra discusión.

OP1: $\sigma_{Dni = '123456789'}$ (EMPLEADO)

OP2: $\sigma_{NumeroDpto > 5}$ (DEPARTAMENTO)

OP3: $\sigma_{Dno = 5}$ (EMPLEADO)

OP4: $\sigma_{Dno = 5 \text{ AND } Sueldo > 30000 \text{ AND } Sexo = 'F'}$ (EMPLEADO)

OP5: $\sigma_{DniEmpleado='123456789' \text{ AND } NumProy = 10}$(TRABAJA_EN)n

Métodos de búsqueda en una selección simple. Es posible utilizar varios algoritmos de búsqueda para realizar la selección de registros en un fichero. Estos algoritmos se conocen, a veces, como **exploraciones de fichero**, ya que exploran los registros del fichero en el que se realiza la búsqueda y encuentran los registros que satisfacen una condición de selección.[4] Si el algoritmo de búsqueda implica la utilización de un índice, esta búsqueda mediante un índice se denomina **exploración indexada**. Los siguientes métodos de búsqueda (S1 a S6) son ejemplos de algunos de los algoritmos de búsqueda que se pueden utilizar para implementar una operación de selección.

- **S1—Búsqueda lineal (fuerza bruta).** Extraer todos los registros del fichero y comprobar si sus valores de atributos satisfacen la condición de selección.
- **S2—Búsqueda binaria.** Si la condición de selección implica una comparación de igualdad sobre un atributo clave con el que está ordenado el fichero se puede utilizar la búsqueda binaria, que es más eficaz que la búsqueda lineal. Un ejemplo es OP1 si Dni es el atributo de ordenación para el fichero EMPLEADO.[5]
- **S3—Utilización de un índice primario (o clave *hash* o clave de dispersión).** Si la condición de selección implica una comparación de igualdad sobre un **atributo clave** con un índice primario (o clave *hash*) (por ejemplo, Dni = '123456789' en OP1), se utilizará el índice primario (o clave *hash*) para encontrar el registro. Observe que con esta condición se extrae un único registro (como mucho).
- **S4—Utilización de un índice primario para encontrar varios registros.** Si la condición de comparación es >, >=, <, o <= sobre un campo clave con un índice primario (por ejemplo, NúmeroDpto > 5 en OP2), se utilizará el índice para encontrar el registro que satisfaga la correspondiente condición de igualdad (NúmeroDpto = 5); a continuación, se extraerán los registros posteriores en el fichero ordenado. Si la condición es NúmeroDpto < 5, se extraerán todos los registros precedentes.
- **S5—Utilización de un índice agrupado para encontrar varios registros.** Si la condición de selección implica una comparación de igualdad sobre un **atributo que no es clave** mediante un índice

[4] A veces, a una operación de selección se la denomina **filtro**, ya que filtra los registros del fichero que *no* satisfacen la condición de selección.

[5] Por lo general, no se utiliza la búsqueda binaria en las búsquedas realizadas en bases de datos, ya que no se utilizan los ficheros ordenados a menos que dispongan del correspondiente índice primario.

agrupado (por ejemplo, Dno = 5 en OP3), se utilizará el índice para extraer todos los registros que cumplan la condición.

- **S6—Utilización de un índice secundario (árbol B+) sobre una comparación de igualdad.** Este método de búsqueda se puede utilizar para extraer un único registro si el campo indexado es una **clave** (tiene valores únicos) o para extraer varios registros si el campo indexado **no es clave**. Esto también se puede utilizar en las comparaciones del tipo >, >=, <, o <=.

En la Sección 15.8, veremos cómo desarrollar fórmulas que calculen el coste de acceso de estos métodos de búsqueda en términos de número de accesos a bloques y de tiempo de acceso. El método S1 se aplica a cualquier fichero, pero los demás métodos dependen de si se dispone de la ruta de acceso adecuada al atributo utilizado en la condición de selección. Los métodos S4 y S6 pueden utilizarse para extraer registros en un *rango* determinado (por ejemplo, 30000 <= Sueldo <= 35000). Las consultas que implican condiciones de este tipo se denominan **consultas de rango**.

Métodos de búsqueda en selecciones complejas. Cuando la condición de una operación SELECT es una **condición conjuntiva** (es decir, si está formada por varias condiciones simples conectadas mediante la conjunción lógica AND como, por ejemplo, el caso OP4 anterior) el DBMS puede utilizar los siguientes métodos adicionales para implementar la operación:

- **S7—Selección conjuntiva utilizando un índice individual.** Si un atributo implicado en cualquier **condición simple y única** de la condición conjuntiva tiene una ruta de acceso que permite el uso de uno de los métodos S2 a S6, se utilizará esa condición para extraer los registros y, posteriormente, comprobar si cada registro extraído *satisface las condiciones simples restantes* de la condición conjuntiva.
- **S8—Selección conjuntiva utilizando un índice compuesto.** Si dos o más atributos están implicados en condiciones de igualdad de la condición conjuntiva y existe un índice compuesto (o estructura *hash*) sobre los campos combinados (por ejemplo, si se ha creado un índice sobre la clave compuesta [DniEmpleado, NumProy] del fichero TRABAJA_EN en OP5), podemos utilizar el índice directamente.
- **S9—Selección conjuntiva mediante intersección de punteros a registros.**[6] Si se encuentran disponibles índices secundarios (u otras rutas de acceso) sobre más de uno de los campos implicados en condiciones simples de la condición conjuntiva, y si los índices incluyen punteros a registros (en lugar de punteros a bloques) entonces se puede utilizar cada uno de los índices para extraer el **conjunto de punteros a registros** que satisfacen la condición individual. La **intersección** de estos conjuntos de punteros a registros da como resultado los punteros a registros que satisfacen la condición conjuntiva y que serán utilizados posteriormente para extraer esos registros de forma directa. Si sólo tienen índices secundarios algunas de las condiciones, cada uno de los registros extraídos será revisado posteriormente para determinar si cumple las condiciones restantes.[7]

Siempre que una condición única especifique la selección (como en el caso de OP1, OP2 u OP3), podemos limitarnos a comprobar si existe una ruta de acceso sobre el atributo implicado en esa condición. Si existe una ruta de acceso, se utilizará el método que corresponde a dicha ruta; en caso contrario, se puede utilizar el modelo de búsqueda de fuerza bruta lineal del método S1. Es necesaria la optimización de consultas en una operación SELECT en la mayoría de los casos de condiciones de selección conjuntivas en las que *más de uno* de los atributos implicados en la condición dispone de una ruta de acceso. El optimizador deberá elegir la ruta

[6] Un puntero a registro identifica de manera única a un registro y proporciona la dirección del registro en el disco; por tanto, también se le denomina **identificador de registro**.

[7] La técnica puede ofrecer muchas variantes (por ejemplo, si los índices son *índices lógicos* que almacenan valores de clave primaria en lugar de punteros a registros).

de acceso que *extraiga el menor número de registros* del modo más eficaz mediante la estimación de los diferentes costes (consulte la Sección 15.8) y la elección del método con el menor coste estimado.

Cuando el optimizador tiene que elegir entre las distintas condiciones simples dentro de una condición de selección conjuntiva, lo que hace normalmente es tener en cuenta la selectividad de cada una de las condiciones. La **selectividad** se define como la relación entre el número de registros (tuplas) que satisfacen la condición y el número total de registros (tuplas) del fichero y, de acuerdo con esto, es un número entre cero y 1 (selectividad cero significa que ningún registro satisface la condición y 1 significa que todos los registros satisfacen la condición). Aunque es posible que no estén disponibles selectividades exactas para todas las condiciones, a menudo se guardan en el catálogo del DBMS unas **estimaciones de selectividades** para que sean utilizadas por el optimizador. Por ejemplo, para una condición de igualdad sobre un atributo clave de la relación $r(R)$, $s = 1/|r(R)|$, donde $|r(R)|$ es el número de tuplas de la relación $r(R)$. Para una condición de igualdad sobre un atributo con i *valores distintos*, s se puede estimar a partir de $(|r(R)|/i)/|r(R)|$ o $1/i$, suponiendo que los registros se encuentran distribuidos de manera uniforme entre los distintos valores.[8] Con esta suposición, un total de $|r(R)|/i$ registros cumplirán una condición de igualdad sobre este atributo. Por lo general, el número estimado de registros que satisfacen una condición de selección con selectividad s es $|r(R)| * s$. Cuanto menor sea este valor estimado, más recomendable será tener en cuenta esa condición para extraer los registros.

Una **condición disyuntiva** (en la cual las condiciones simples están conectadas mediante la operación lógica OR en lugar de estarlo mediante AND) es mucho más difícil de procesar y optimizar que una condición de selección conjuntiva. Tomemos como ejemplo OP4':

$$\text{OP4'}: \sigma_{\text{Dno=5 OR Sueldo>30000 OR Sexo='M'}}(\text{EMPLEADO})$$

Con una condición de este tipo no se puede realizar demasiada optimización, ya que los registros que satisfacen la condición disyuntiva son la *unión* de los registros que satisfacen las condiciones individuales. Por tanto, si *una* cualquiera de las condiciones no dispone de ruta de acceso, nos veremos obligados a utilizar el modelo de búsqueda de fuerza bruta lineal. Sólo si existe una ruta de acceso para *todas* las condiciones podremos optimizar la selección obteniendo los registros (o los identificadores de registro) que satisfacen cada una de las condiciones para después utilizar la operación de unión para eliminar duplicados.

Un DBMS suele disponer de muchos de los métodos discutidos anteriormente y, por lo general, muchos otros métodos adicionales. El optimizador de consultas elegirá el más adecuado para la ejecución de cada operación SELECT en una consulta. Esta optimización utiliza fórmulas para estimar los costes de cada uno de los métodos de acceso disponibles, como veremos en la Sección 15.8. El optimizador elegirá el método de acceso que tenga el menor coste estimado.

15.3.2 Implementación de la operación JOIN

La operación JOIN es una de las operaciones que más tiempo consumen durante el procesamiento de una consulta. Muchas de las operaciones de concatenación que se pueden encontrar en las consultas son de los tipos EQUIJOIN y NATURAL JOIN, así que aquí sólo tendremos en cuenta estos dos. Durante el resto de este capítulo, el término **concatenación** se refiere a una EQUIJOIN (o NATURAL JOIN). Existen muchas maneras de implementar una operación de concatenación que involucre a dos ficheros (**concatenación de dos vías**). Las concatenaciones que involucran a más de dos ficheros se denominan **concatenaciones multivía**. El número de vías posibles para ejecutar concatenaciones multivía crece con mucha rapidez. En esta sección sólo revisaremos técnicas para la implementación de concatenaciones del tipo dos-vías. Para ilustrar nuestra discusión utilizaremos, una vez más, el esquema relacional de la Figura 5.5, en concreto, utilizaremos las relaciones

[8] En los optimizadores más sofisticados es posible almacenar en el catálogo los histogramas que representan la distribución de los registros entre los diferentes valores de los atributos.

EMPLEADO, DEPARTAMENTO y PROYECTO. Los algoritmos que estamos tratando se utilizan en operaciones de concatenación de la forma

$$R \bowtie_{A=B} S$$

donde A y B son atributos compatibles en dominio de R y S, respectivamente. Los métodos que estamos discutiendo pueden ser extendidos a formas más generales de concatenación. Mostraremos cuatro de las técnicas más habituales de ejecutar una concatenación de este tipo utilizando las siguientes operaciones de ejemplo:

OP6: EMPLEADO $\bowtie_{\text{Dno=NúmeroDpto}}$ DEPARTAMENTO

OP7: DEPARTAMENTO $\bowtie_{\text{DniDirector=Dni}}$ EMPLEADO

Métodos para la implementación de concatenaciones.

- **J1—Concatenación de bucle anidado (fuerza bruta).** Para cada registro t de R (bucle externo), obtiene todos los registros s de S (bucle interno) y comprueba si los dos registros satisfacen la condición de concatenación $t[A] = s[B]$.[9]

- **J2—Concatenación de bucle simple (utilizando una estructura de acceso para obtener los registros que cumplen la condición).** Si existe un índice (o clave *hash*) para uno de los atributos de la concatenación (por ejemplo, B de S), obtiene todos los registros t de R, uno a la vez (bucle simple), y a continuación utiliza la estructura de acceso para obtener directamente todos los registros s de S que cumplen $s[B] = t[A]$.

- **J3—Concatenación de ordenación-mezcla.** Si los registros de R y S se encuentran *físicamente clasificados* (ordenados) por el valor de los atributos de concatenación A y B respectivamente, podemos implementar la concatenación del modo más eficiente posible. Ambos ficheros serán explorados concurrentemente siguiendo el orden de los atributos de concatenación y emparejando los registros que tienen los mismos valores para A y B. Si los ficheros no se encuentran ordenados, podrían ser ordenados previamente mediante una ordenación externa (consulte la Sección 15.2). Según este método, las parejas de los bloques de los ficheros se copian en los búferes de memoria en orden y los registros de cada fichero se exploran sólo una vez cada uno para comprobar su emparejamiento con el otro fichero (a menos que A y B no sean atributos clave, en cuyo caso el método necesita ser modificado ligeramente). En la Figura 15.3(a) se muestra un esquema del algoritmo de concatenación de clasificación-mezcla. Utilizamos $R(i)$ para referirnos al *i-ésimo* registro de R. Se puede utilizar una variación de la concatenación de clasificación-mezcla cuando existen los índices secundarios en ambos atributos de concatenación. Los índices proporcionan la posibilidad de acceder (explorar) a los registros siguiendo el orden de los atributos de concatenación, pero ya que los registros se encuentran dispersos por todos los bloques del fichero este método puede resultar bastante ineficaz, ya que todos los accesos a registros pueden implicar el acceso a distintos bloques en disco.

- **J4—Concatenación de dispersión (*hash*).** Los registros de los ficheros R y S se encuentran clasificados en el mismo fichero de dispersión, utilizando la misma función de dispersión sobre los atributos A de R y B de S como claves de dispersión. En primer lugar, una única pasada sobre el fichero con menor número de registros (por ejemplo, R) reparte sus registros en los bloques del fichero de dispersión; esto se denomina **fase de particionamiento**, ya que los registros de R se particionan sobre los bloques de dispersión. En la segunda fase, denominada **fase de prueba**, se hace una única pasada por el otro fichero (S) tomando cada uno de los registros, *probando* en el bloque adecuado y emparejándolos con los registros correspondientes de R en ese bloque. En esta descripción simplificada de la concatenación

[9] En los ficheros en disco, es obvio que los bucles se ejecutarán sobre bloques en disco y, por tanto, esta técnica también recibe el nombre de *concatenación anidada en bloque.*

de dispersión se supone que el fichero de menor tamaño *cabe en su totalidad en los bloques de memoria* tras la primera fase. Veremos más adelante algunas variaciones de la concatenación de dispersión en las que no se presupone esta afirmación.

Figura 15.3. Implementación de JOIN, PROJECT, UNION, INTERSECTION y SET DIFFERENCE utilizando clasificación-mezcla, donde R tiene n tuplas y S tiene m tuplas. (a) Implementación de la operación $T \leftarrow R \bowtie_{A=B} S$. (b) Implementación de la operación $T \leftarrow \pi_{<\text{lista de atributos}>}(R)$.

(a)

```
clasificación de las tuplas de R sobre el atributo A;   (* suponiendo que R tiene n tuplas (registros) *)
clasificación de las tuplas de S sobre el atributo B;   (* suponiendo que S tiene m tuplas (registros) *)
set i ← 1, j ← 1;
while (i ≤ n) and (j ≤ m)
do {  if R(i)[A] > S(j)[B]
          then set j ← j + 1
      elseif R(i)[A] < S(j)[B]
          then     set i ← i + 1
      else { (* R(i)[A] = S(j)[B], así que generamos una tupla emparejada *)
             generar la tupla combinada <R(i), S(j)> en T;

             (* generar otras tuplas que cuadren con R(i), si existen *)
             set l ← j + 1;
             while (l ≤ m) and (R(i)[A] = S(l)[B])
             do {  generar la tupla combinada <R(i), S(l)> en T;
             set l ← l + 1
             }

             (* generar otras tuplas que cuadren con S(j), si existen *)
             set k ¬ i + 1;
             while (k ≤ n) and (R(k)[A] = S(j)[B])
             do {  generar la tupla combinada <R(k), S(j)> en T;
                   set k ← k + 1
             }
             set i ← k, j ← l
   }
}
```

(b)

```
crear una tupla t[<lista de atributos>] en T' para cada tupla t de R;
        (* T' contiene los resultados de la proyección antes de la eliminación de duplicados *)
if <lista de atributos> incluye una clave de R
        then T ← T'
else {  ordenar las tuplas de T';
        set i ← 1, j ← 2;
        while i ≤ n
        do {   generar la tupla T'[i] en T;
               while T'[i] = T'[j] and j ≤ n do j ← j + 1; (* eliminar duplicados *)
               i ← j; j ← i + 1
        }
}
(* T contiene el resultado de la proyección tras la eliminación de duplicados *)
```