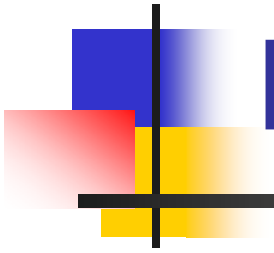
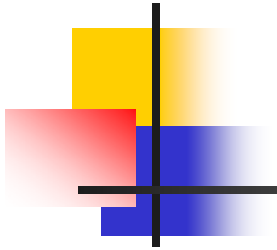


Diseño Físico

Diseño de Bases de Datos Relacionales
Curso 2011/2012

Sergio Ilarri
silarri@unizar.es

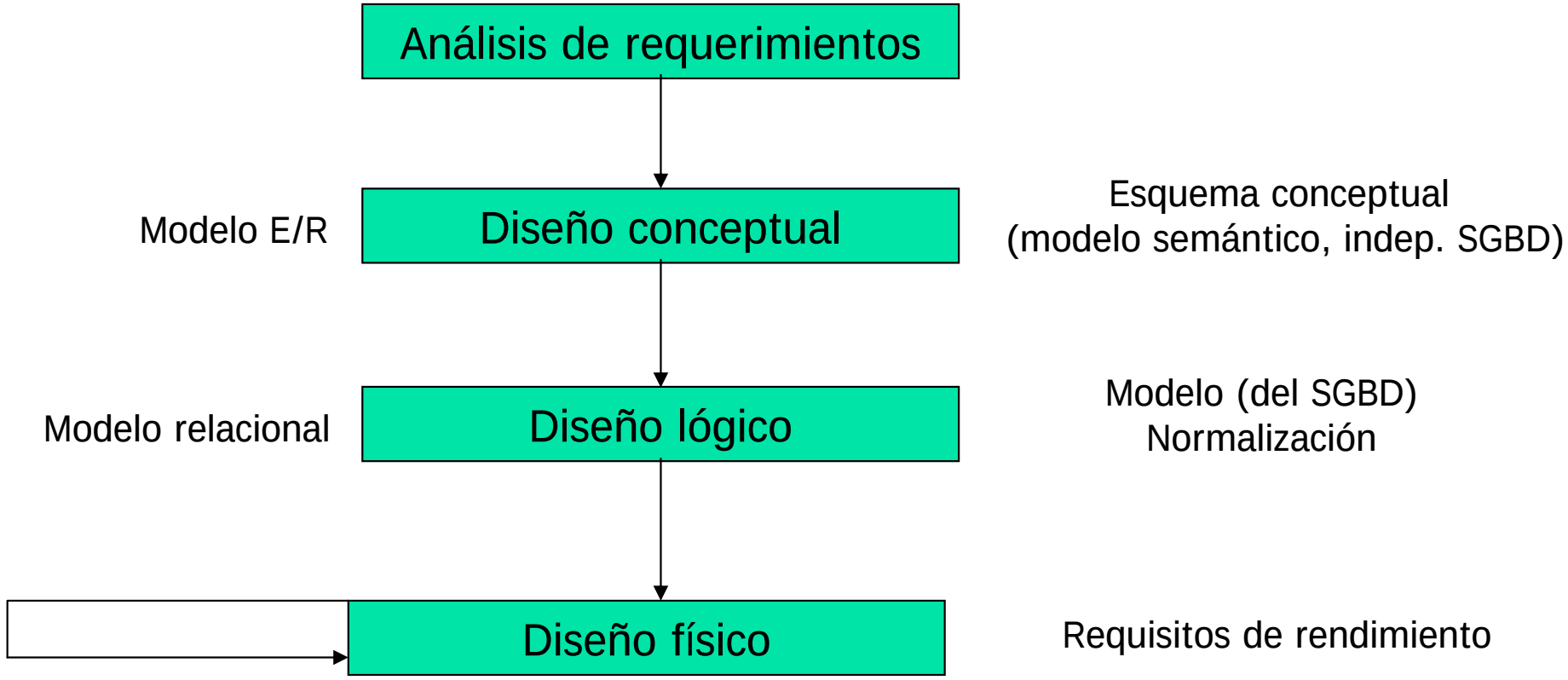


Índice

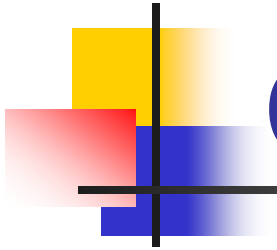
- Introducción, objetivo
- ¿Cómo hacer diseño físico?
- Paso previo: recopilar datos
- Estrategias:
  1. Selección de claves apropiada
  2. Desnormalización
  3. Partición horizontal
  4. Partición vertical
  5. Precálculo de joins
  6. Materialización de vistas
  7. Índices
  8. Agrupamientos dispersos
  9. Agrupamientos de tablas

  ...

Introducción

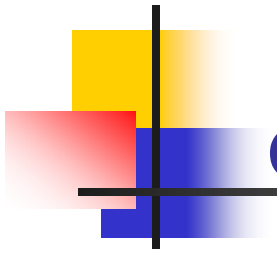


Diseño lógico estándar vs. Diseño lógico específico



Objetivo del Diseño Físico

- Escoger estructuras de almacenamiento y caminos de acceso adecuados y que equilibren:
  - Rendimiento (*throughput*, latencia, tiempo de respuesta, etc.)
  - Coste (espacio usado, reorganizaciones de datos, etc.)
- Su importancia crece con el volumen de datos
- Dificultades:
  - Muchos aspectos dependen del SGBD concreto
  - Muchos factores, efectos interrelacionados
  - No existen metodologías claramente definidas
  - Entender la optimización de consultas es muy útil
- Objetivo alcanzable: un “buen” (pero no necesariamente óptimo) diseño, prueba y error



¿Cómo Hacer Diseño Físico?

- Una aproximación
  1. Paso previo: recopilar datos de las operaciones esperadas
  2. Optimizar las operaciones con mayores restricciones
  3. Optimizar otras operaciones
     - Tener en cuenta que favorecer consultas normalmente perjudica a las actualizaciones

Paso Previo: Recopilar Datos (I)

- Por cada operación:
  - Tipo: INSERT, SELECT, UPDATE, DELETE
  - Tablas que accede (con su cardinalidad)
  - Condiciones de selección (selectividad)
  - Condiciones de combinación-join (selectividad)
  - Atributos a proyectar/modificar
  - Frecuencia estimada
  - Restricciones temporales de ejecución (si las hay)
- Regla 80-20
- Considerar la prioridad y frecuencia de las operaciones

Paso Previo: Recopilar Datos (II)

Ejemplo

```
SELECT nombre, nombreAsign
FROM    Profesor, Asignatura, Matricula
WHERE  Profesor.dni= Asignatura.codProf and
           Asignatura.codAsig= Matricula.codAsig and
           Profesor.dept='IIS'
```

a) Tipo: SELECT
b) Tablas: Profesor(300), Asignatura(1000), Matricula(10000)
c) Conds. selección: dept='IIS' (30 profesores) (**10%**)
d) Condiciones join: Profesor.dni=Matricula.codProf (10000) (**100%**)
   Asignatura. codAsig=Matricula.codAsig (10000) (**100%**)
e) Atributos a proyectar: nombre, nombreAsign
f) Frecuencia: 3 veces/día
g) Restricciones: que nunca tarde más de unos segundos

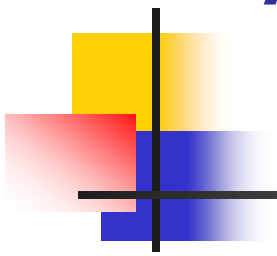
Paso Previo: Recopilar Datos (III)

Ejemplo

UPDATE Profesor
SET dept='IIS' {En general una constante}
WHERE dni=18 {En general una constante}

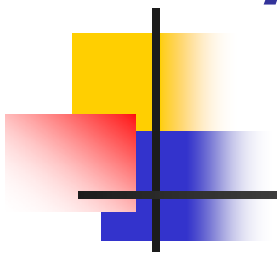
a) Tipo: UPDATE
b) Tablas: Profesor(300)
c) Conds. selec.: dni=18 (0.3%)
d) Conds. Join: no hay
e) Atributos a modificar: dept
f) Frecuencia 10 veces/año
g) Restricciones: no hay

Algunas Consideraciones Iniciales de Diseño Físico (I)

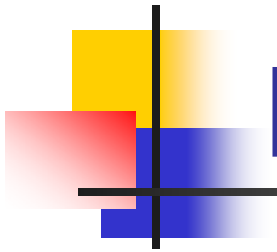


- Selección de tipos de datos adecuados para los atributos
  - Objetivos
    - Minimizar el espacio de almacenamiento
    - Representar todos los posibles valores
    - Mejorar la integridad de los datos
    - Soportar las manipulaciones de datos requeridas
  - Para algunos atributos con dominio de valores disperso → tablas de códigos

Algunas Consideraciones Iniciales de Diseño Físico (II)



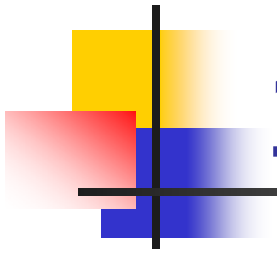
- Diseño de los ficheros subyacentes
  - Organización de ficheros:
    - Organización secuencial
    - Organización indexada:
      - Índice primario vs. Índice secundario
    - Organización dispersa
  - Oracle: *tablespace*
- Uso de varios discos
  - RAID → *striping* (*round-robin*)
  - Balancear la carga de E/S entre varios controladores de disco



Estrategias de Diseño Físico

1. Selección de claves apropiada
2. Desnormalización
3. Partición horizontal
4. Partición vertical
5. Precálculo de joins
6. Materialización de vistas
7. Índices
8. Agrupamientos dispersos
9. Agrupamientos de tablas

...



1) Reconsiderar las Claves (I)

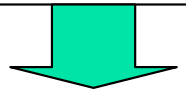
- Clave: conjunto mínimo de atributos que garantizan la unicidad de la tupla
- Idea:
  - Usar claves significativas
  - Evitar los *autonuméricos*
- Objetivo: reducir el número de joins necesarios para recuperar la información de interés

1) Reconsiderar las Claves (II)

Matricula(<u>codAsign, codAlumno</u>)
Asignatura(<u>cod</u>, nombre, creditos, ...)

codAsign = 1, 2, ...

```
SELECT COUNT(codAlumno)
FROM Matricula, Asignatura
WHERE cod= codAsign AND
        nombre=´Diseño de Bases de Datos Relacionales´
```



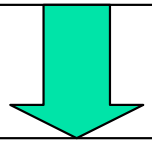
```
SELECT COUNT(codAlumno)
FROM Matricula
WHERE codAsign=´DBDR´
```

2) Desnormalización (I)

- Hacer lo contrario a normalizar:
  - Redundancia controlada
  - En lugar de dividir R en R1 y R2 con R=R1⋈R2
  - Pasamos algunos (o todos) atributos de R2 a R1:
    - ¿Cuándo?: cuando esos atributos se necesitan casi siempre que se consulta R1
    - Objetivo: evitar hacer el join
    - Problema: habrá que controlar las redundancias
      - Ojo: controlar la redundancia podría ser tan costoso como hacer el join
    - Último recurso

2) Desnormalización (II)

Persona(dni, nombre, dir, codP)
CodPostal(codP, ciudad)



Persona(dni, nombre, dir, codP, ciudad)
CodPostal(codP, ciudad)

Si suponemos que siempre que obtenemos el CP de una persona, queremos saber la ciudad...

No está en 3FN

¿Qué hacemos con la tabla de códigos postales?

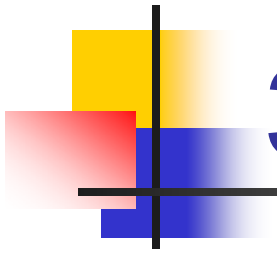
-Se puede mantener, para introducir directamente la ciudad en Persona a partir del codP

-Ayuda para garantizar la consistencia

Redundancias => mantener la consistencia con disparadores

2) Desnormalización (III)

- Tipos comunes de desnormalización:
  - Combinar dos tipos de entidad de un tipo de interrelación 1:1
  - Combinar dos tipos de entidad de un tipo de interrelación 1:N
    - Se puede conservar también alguna de las tablas originales (por ejemplo, para protegerse contra anomalías de borrado)
- Será necesario mantener la consistencia:
  - Disparadores, herramientas de sincronización entre datos base y réplicas



3) Partición Horizontal (I)

- Idea: distribuir las filas de una tabla en varias
- Objetivo: ↓ coste de selección, ↓ tamaño de la tabla
- Puede ser interesante si las distintas categorías de tuplas se usan por separado
- Dada $R = s_{C1}(R) \cup \ldots \cup s_{Cn}(R)$, conviene si:
  - Muchas operaciones con la BD son con $s_{Ci}(R)$
  - Algunos atributos son inaplicables (NULL) según Ci
- Pasar a usar, por eficiencia, las nuevas tablas
- Puede acompañarse de una proyección sobre cada fragmento (para eliminar valores inaplicables al grupo –ej., subtipos-)
- Partición lógica (vistas) vs. partición física vs. partición soportada por el SGBD de forma transparente

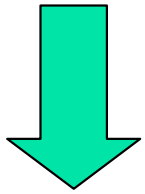
3) Partición Horizontal (II)

- Posibilidades:
  1. Cada $\sigma_{Ci}(R)$ se guarda en una tabla y se define una vista sobre R
  2. Si $\sigma_{Ci}(R)$ es muy frecuente, con Ci muy selectivo y R muy grande, podemos almacenar $\sigma_{Ci}(R)$ en otra tabla y conservar R

-Disparadores (→ mantener la consistencia)
-¿En qué tabla se inserta?
-Si después se suprime la nueva tabla, crear una vista (→ independencia física)

3) Partición Horizontal (III)

Persona(dni, nombre, status, dpto, curso, categoria, trabajo)



Se evitarán atributos nulos

Alumno(dni, nombre, curso)

Profesor(dni, nombre, dpto, categoria)

PAS(dni, nombre, dpto, trabajo)

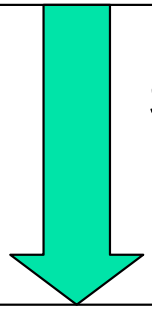
Vista Persona, unión rellenando con nulos

Se puede ver como:

- Diseño conceptual/lógico (clase Persona con subclases Profesor y PAS)
- Diseño físico (separamos por frecuencias de acceso)

3) Partición Horizontal (IV)

Empleados(dni, nombre, direccion, tnfo, cargo, dptoDirigido)



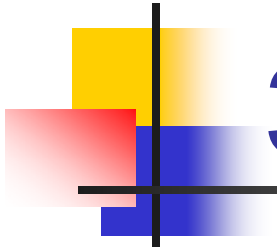
Si se pregunta mucho por directores

Directores(dni, dptoDirigido) + restric. clave ajena

Empleados (dni, nombre, cargo, direccion, tnfo)

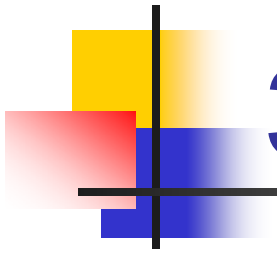
Si luego se destruye la tabla Directores:

-Crear una vista para asegurar la independencia física



3) Partición Horizontal (V)

- Partición soportada por el SGBD:
  - La define el administrador pero luego su uso es transparente
  - Algunas ventajas:
    - Divide y vencerás: podado de particiones en consultas (optimizador de consultas), disminuyen la competición en el acceso a la tabla completa
    - Agrupamiento de datos (ordenación) por particiones
    - Salva posibles limitaciones referentes al tamaño máximo de las tablas (tamaño de *RIDs –Row Identifiers–*)
    - Facilita la administración (+ fáciles de manejar), al poder trabajar con fragmentos/subconjuntos de datos (copias de seguridad sólo de particiones recientes, desfragmentación, ordenación, etc.)
    - Gestión automática de entradas y salidas de particiones (*roll-in*, *roll-out*, por ejemplo para atributos de tipo fecha)



3) Partición Horizontal (VI)

- Tipos de partición horizontal soportados por Oracle:
  - Partición por rango
  - Partición por lista
    - Se especifican los valores de atributos que caracterizan cada partición (no necesariamente forman rangos)
  - Partición por intervalo
    - Las particiones se crean automáticamente conforme llegan datos
  - Partición por dispersión
    - Distribución equilibrada, para disminuir la contención en el acceso
  - Partición compuesta
    - Ejemplo (rango-rango): particionado por fecha de solicitud y luego cada partición particionada por fecha de envío
  - …

3) Partición Horizontal (VII)

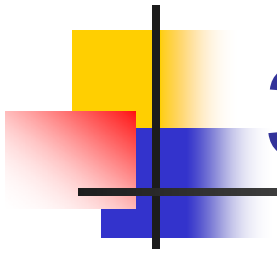
Ejemplo en Oracle

```
CREATE TABLE Venta(idProducto NUMBER(6) PRIMARY KEY, idCliente NUMBER,
   fecha DATE, idCanal CHAR(1), idPromocion NUMBER(6), cantidad NUMBER(3),
   precio NUMBER(10,2))
PARTITION BY RANGE (fecha) (
PARTITION cuat1
   VALUES LESS THAN (TO_DATE('01-APR-2006','dd-MON-yyyy')) TABLESPACE ventas1,
PARTITION cuat2
   VALUES LESS THAN (TO_DATE('01-JUL-2006','dd-MON-yyyy')) TABLESPACE ventas2,
PARTITION cuat3
   VALUES LESS THAN (TO_DATE('01-OCT-2006','dd-MON-yyyy')) TABLESPACE ventas3,
PARTITION cuat4
   VALUES LESS THAN (TO_DATE('01-JAN-2007','dd-MON-yyyy')) TABLESPACE ventas4
);
```

3) Partición Horizontal (VIII)

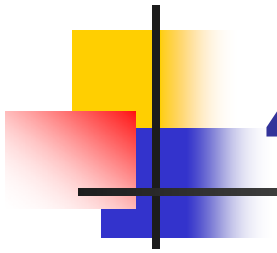
Ejemplo en Oracle

```
CREATE TABLE Empleado(idEmpleado NUMBER(4) PRIMARY KEY,
   nombreEmpleado VARCHAR2(30), tarea VARCHAR2(40), salario NUMBER(7,2),
   nombreDepartamento VARCHAR2(30)
)
PARTITION BY RANGE(fechaContratacion)
INTERVAL (NUMTOYMINTERVAL(1,‘year'))
(PARTITION empleadosHasta2010 values LESS THAN
   (TO_DATE('01-JAN-2011','DD-MON-YYYY'))
) ;
```



3) Partición Horizontal (IX)

- También se pueden dividir los índices
  - Índices globales: sobre la tabla completa
  - Índices locales: definidos sobre cada partición
- Los índices locales son normalmente más eficientes, ya que el correspondiente árbol B+ es más pequeño; operaciones de reorganización sobre fragmentos del índice
- Pero los índices globales son una forma natural de forzar la unicidad de atributos que no coinciden con los atributos de rango
- No todos los SGBD soportan ambos tipos de índices



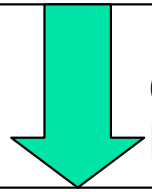
4) Partición Vertical (I)

- Idea: distribuir las columnas de una tabla en varias
- Objetivo: aumentar el factor de bloque
- Dada R(A1, ..., An, B1, ..., Bm), conviene si:
  - Muchas operaciones frecuentes afectan a A1,..., An y pocas a B1, ..., Bm
  - R(A1, ..., An, B1, ..., Bm) es mucho más grande que R(A1, ..., An)
- Disparadores para mantener la consistencia

Lógicamente, la clave primaria hay que repetirla en cada tabla

4) Partición Vertical (II)

Empleados(dni, nombre, cargo, direccion, tfno, fechaNac, ciudad, sexo)



dni, nombre, cargo (50 bytes)
El resto (300 bytes), apenas se usan

Empleados'(dni, nombre, cargo) + disparador
Empleados(...)

-Caben 7 veces más registros en un mismo bloque (50 bytes frente a 350)

-Independencia física al eliminar Empleados': vista

O controlar
desde programa...

5) Precálculo de Joins (I)

- Idea: tener precalculado un join

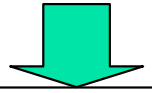
R1 ⋈ R2 ⋈ ... ⋈ Rn si:

  - Se ejecuta con frecuencia
  - Las Ri no se actualizan mucho
- Es un caso de desnormalización
- Para recalcular en caso de cambio de un Ri:
  - Disparadores
  - Scripts periódicos (inconsistencia temporal) si no se requiere siempre la respuesta actualizada

5) Precálculo de Joins (II)

Empleado(dni, nombre)
Asignacion(dni, codEquipo, esJefe), donde esJefe ='S' o 'N'
Equipo(codEquipo, nombreEquipo)

Si queremos obtener cada empleado con su jefe y el del equipo al que pertenece, hay que hacer un join de 5 tablas (Emp1, Emp2, Asig1, Asig2, y Equipo)



CREATE TABLE NombreEquipoJefeEquipo
INSERT INTO NombreEquipoJefeEquipo SELECT...
+ trigger o script que re-cree la tabla periódicamente

Costoso (hay que ejecutar el join con cada actualización)

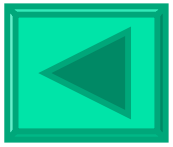
5) Precálculo de Joins (III)

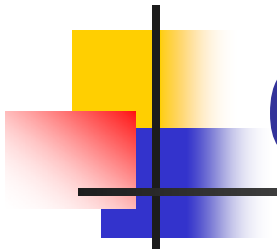
```
SELECT  Emp1.nombre, Emp2.nombre, Equipo.nombreEquipo
FROM    Empleado Emp1, Empleado Emp2,
            Asignacion Asig1, Asignacion Asig2, Equipo
WHERE  Emp1.dni=Asig1. dni and Asig1. esJefe ='N'
and Asig1.codEquipo=Equipo.codEquipo
and Asig1.codEquipo=Asig2.codEquipo
and Asig2. esJefe ='S'
and Emp2. dni=Asig2. dni
```

Emp1 es empleado

Para acceder a nombreEquipo

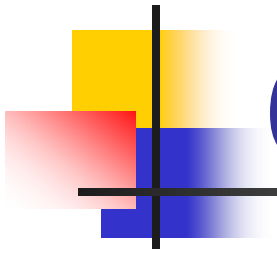
Emp2 es jefe





6) Materialización de Vistas (I)

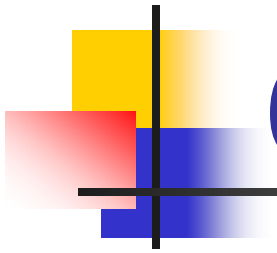
- A diferencia de las vistas normales, las vistas materializadas se almacenan como tablas normales (resultados precalculados almacenados)
- Rendimiento:
  - Útiles en el caso de preguntas frecuentes (no hay que acudir a las tablas base)
  - Pero hay que propagar las actualizaciones de las tablas base a la vista materializada
  - Es interesante indexar las vistas materializadas
- El precálculo de joins se puede considerar un caso particular (si se define como una vista materializada)



6) Materialización de Vistas (II)

- Interesa un enrutamiento automático hacia vistas materializadas
  - El optimizador de consultas determina que ciertas vistas materializadas pueden ser útiles para una consulta
  - El optimizador de consultas determina si esa sustitución sería beneficiosa
  - Si no, es el que define la consulta el que tiene que usarlas explícitamente
    - Aproximación menos flexible, trabajo extra

Parámetro *QUERY_REWRITE_ENABLED* en Oracle



6) Materialización de Vistas (III)

- Dos aproximaciones:
  - Mantenimiento incremental (tiempo real)
  - Reconstrucción periódica
- Limitar el número de vistas materializadas (no más del 10-20% del almacenamiento total)
  - Coste de actualización de las vistas
  - Coste de almacenamiento de las vistas
  - Coste de administración de las vistas
  - Coste de enrutamiento automático

6) Materialización de Vistas (IV)

En Oracle: refresco completo

```
CREATE MATERIALIZED VIEW VentasPorCliente
REFRESH COMPLETE
START WITH TO_DATE('01-01-2011 01:00:00',
   'DD-MM-YYYY HH24:MI:SS')
NEXT SYSDATE + 1
AS SELECT idCliente, sum(venta) AS venta
FROM TablaVentas
GROUP BY idCliente;
```

6) Materialización de Vistas (V)

En Oracle: refresco incremental

```
CREATE MATERIALIZED VIEW LOG ON TablaVentas
WITH ROWID, SEQUENCE(idCliente, venta)
INCLUDING NEW VALUES;

CREATE MATERIALIZED VIEW VentasPorCliente
REFRESH FAST
START WITH TO_DATE('01-01-2011 01:00:00',
   'DD-MM-YYYY HH24:MI:SS')
NEXT SYSDATE + 1
AS SELECT idCliente, sum(venta) AS venta
FROM TablaVentas
GROUP BY idCliente;
```

Actualización incremental → seguimiento de los cambios en las tablas base

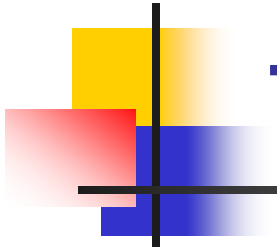
6) Materialización de Vistas (VI)

En Oracle: refresco en tiempo real (al hacerse *commit*)

```
CREATE MATERIALIZED VIEW VentasPorCliente
REFRESH ON COMMIT
AS SELECT idCliente, sum(venta) AS venta
FROM TablaVentas
GROUP BY idCliente;
```

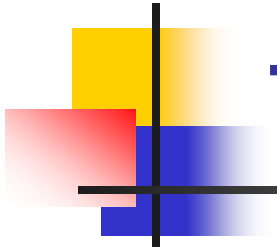
7) Índices (I)

- Conceptos básicos:
  - Índice único (*unique*) vs. secundario (*secondary*):
    - Índice único → se utiliza un atributo (o combinación de atributos) con valor único
    - Índice secundario → índice no único
  - Índice agrupado (*clustered*) no agrupado (*unclustered*):
    - Índice agrupado → los datos se organizan físicamente por orden de los valores del índice, sólo puede haber 1 por tabla
  - Índice denso (*dense*) vs. disperso (*sparse*)
    - Índice denso → una entrada por valor
    - Índice disperso → una entrada por bloque de datos de la tabla
  - Índice compuesto → consta de varios atributos (en cierto orden)
    - C*overing index* → suficientes datos para poder responder a ciertas preguntas sólo con él
  - Implementación más utilizada en SGBD: árboles B+



7) Índices (II)

- Algunas situaciones propicias (reglas heurísticas):
  - Índices únicos para claves primarias / atributos únicos (Oracle lo fuerza)
  - Índices para las claves ajenas (joins, acciones integridad referencial)
  - Atributos usados frecuentemente en condiciones de selección
    - Predicados de igualdad (poco útiles con predicados de desigualdad por la alta selectividad)
    - Predicados de rango (para índices implementados como árboles B+)
    - Más interesantes cuanto menor sea la selectividad de la condición (número de elementos distintos en el índice al menos 30% de la cardinalidad de la tabla)
  - Tablas grandes (pero también pueden ser útiles con las pequeñas)
  - Atributos usados en ORDER BY, GROUP BY (comprobar el SGBD usado para ver si se aprovecharía de dichos índices o no)
  - Favorecer el mayor número posible de consultas, teniendo en cuenta también sus frecuencias



7) Índices (III)

- Algunas precauciones:
  - Considerar el coste de actualización, no definirlos sobre atributos que se actualizan muy frecuentemente (OLTP vs. OLAP)
  - Evitar índices sobre atributos de tipo cadena muy largos
  - Evitar índices sobre atributos con pocos valores (p.ej., sexo)
  - Evitar índices redundantes
  - Cuidado con los valores nulos
  - Seleccionar bien los índices agrupados (sólo 1 máx. por tabla)
  - Si se van a cargar muchos datos, crear el índice después de insertar los datos (vs. construcción incremental)
  - Evitar casos extremos en la cardinalidad del índice (muy pocas entradas o tamaño muy grande comparado con la tabla indexada)
  - No abusar de los índices de cobertura (muy específicos, con claves muy largas → pocas entradas por bloque del índice)

7) Índices (IV)

- Ejemplos:

```
CREATE TABLE Persona(dni number primary key, nombre, ...);
```

```
CREATE INDEX nombre_idx ON Persona(nombre);
```

```
CREATE UNIQUE INDEX persona_idx ON Persona(apellidos, nombre);
```

7) Índices (V)

- Índices compuestos:
  - Atributos que aparecen simultáneamente en la cláusula WHERE (con AND), especialmente cuando la condición combinada sea restrictiva
  - Debido a su mayor tamaño, las consultas que sólo involucran a uno de esos atributos se ejecutarán menos eficientemente
  - Sin embargo, para condiciones donde intervienen todos los atributos es más eficiente que tener índices separados para cada atributo
  - Si algunos atributos aparecen más frecuentemente en la cláusula WHERE, colocarlos primero en el índice compuesto
  - Si todos los atributos aparecen con igual frecuencia pero los datos están físicamente ordenados por un atributo, colocar este el primero

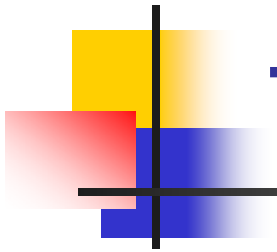
```
CREATE INDEX parteDeReparto ON Reparto(numPedido, fechaReparto);
```

(índice secundario compuesto)

7) Índices (VI)

- Índice de mapa de bits:
  - Por defecto, se usa normalmente un *B-tree index*
  - Alternativa para índices secundarios: *bitmap index*
    - Tabla de bits <tuplas, valores>: bit activado → el registro correspondiente tiene ese valor
    - Para atributos con pocos valores posibles (nunca únicos, ya que en ese caso es mejor el *B-tree index*)
    - Eficiente manipulación de bits
    - Pueden indexar valores nulos

```
CREATE BITMAP INDEX bitmap_ciudad_idx on Persona(ciudad);
vs.
CREATE INDEX ciudad_idx on Persona(ciudad);
```



7) Índices (VII)

- Índice de join (I):
  - Índice sobre atributos de 2 (o más) relaciones que toman valores en el mismo dominio (índice sobre el join)
  - Interesante cuando queremos relacionar tuplas de gran tamaño
  - Pre-calcula los emparejamientos (joins) entre tablas
  - Pueden añadirse atributos de una o ambas relaciones, de modo que se puede evitar el acceso a las tablas base si no se requieren otros atributos
  - En Oracle: *bitmap join indexes*

7) Índices (VIII)

- Índice de join (II):
  - ```
    CREATE BITMAP INDEX cliente_ventas ON Venta(Cliente.sexo)
    FROM Venta, Cliente
    WHERE Venta.idCliente=Cliente.idCliente;
    ```
    ```

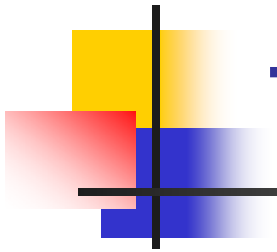
7) Índices (IX)

- Índices basados en funciones:

```
CREATE TABLE Salario(dni NUMBER PRIMARY KEY,
                     salarioBase INTEGER NOT NULL,
                     complementos INTEGER NOT NULL);

CREATE INDEX idx_sal ON Salario(salarioBase + complementos);

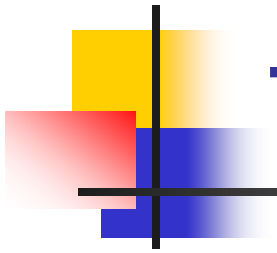
SELECT * FROM Salario
WHERE salarioBase + complementos < 1200
ORDER BY salarioBase;
```



7) Índices (X)

- Índices de dominio:
  - Se refiere a índices diseñados para aplicaciones de propósito especial (p.ej., para indexar datos espaciales)
- Índices particionados:
  - Objetivo: mejorar la disponibilidad, rendimiento, escalabilidad y manejabilidad
  - Por ejemplo, podemos tener un índice particionado de forma que cada partición del índice indexe una partición distinta de una tabla

```
CREATE INDEX idx_alumno ON Alumno(nip)
GLOBAL PARTITION BY RANGE(nip)
(PARTITION p1 VALUES LESS THAN(1000),
 PARTITION p2 VALUES LESS THAN(2000),
 PARTITION p3 VALUES LESS THAN(MAXVALUE));
```



7) Índices (XI)

- Monitorizar el uso de índices (Oracle 9i)

```
ALTER INDEX <nombreÍndice> MONITORING USAGE
```

```
ALTER INDEX <nombreÍndice> NOMONITORING
```

Información en la vista *V$OBJECT_USAGE*

8) Agrupamientos Dispersos

- *Hash clusters*:
  - Si un atributo A se usa con frecuencia en selecciones tipo A=k (k cte.) o en joins y no para recuperar por orden de A (p.ej., selecciones A>=k)

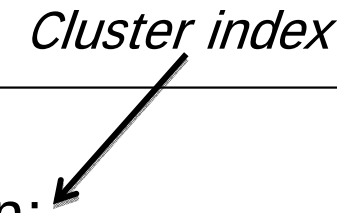
```
CREATE CLUSTER cper(num number) HASHKEYS 100000;
CREATE TABLE Per(dni number primary key,...) CLUSTER cper (dni);
```

hashkeys=número de valores generados por la función hash

  - ¿Indexación vs. dispersión?

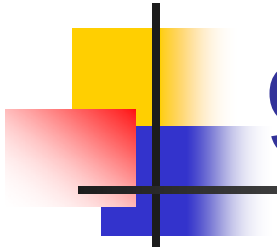
9) Agrupamientos de Tablas (I)

- Algunos SGBD (por ejemplo, Oracle) permiten almacenar varias tablas en mismo agrupamiento (alternativa a desnormalizar)



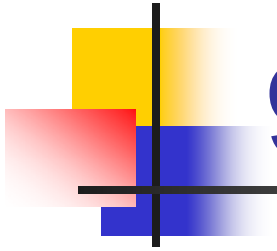

```
CREATE CLUSTER profAsign(num number);
CREATE INDEX idx_profAsign ON CLUSTER profAsign;
CREATE TABLE Profesor(dni number primary key,
   nom char(10), ...) CLUSTER profAsign(dni);
CREATE TABLE Asignatura(cod number, dniprof number references
    profesores(dni), ...) CLUSTER profAsign(dniprof);
```

Clúster = grupo de una o más tablas que se almacenan físicamente juntas porque comparten atributos comunes (la clave del cúster) y normalmente se utilizan juntas en consultas SQL



9) Agrupamientos de Tablas (II)

- Algunos consejos:
  - Utilizarlos cuando se acceden frecuentemente en joins
  - No utilizarlos cuando los joins son ocasionales o cuando los atributos comunes se modifican frecuentemente (→ movimientos de tuplas entre bloques)
  - De forma similar, si frecuentemente se añaden o borran tuplas puede ser contraproducente
  - No utilizarlos si con frecuencia se hacen procesamientos secuenciales completos de las tablas por separado (más costosos si están en un agrupamiento)
  - No utilizarlos si el número de tuplas para los distintos atributos comunes varía considerablemente (desperdicio de espacio vs. colisiones, cadena de bloques con el mismo valor para la clave del clúster)
  - ...



9) Agrupamientos de Tablas (III)

- También podría tener un clúster con una única tabla:
  - Si siempre se acceden a datos de esa tabla por grupos
  - Por ejemplo, selección de tuplas de una tabla que detallan valores asociados a la misma tupla en otra tabla
    - No se incrementa el coste del acceso secuencial completo a la única tabla incluida en el clúster
- Alternativa a desnormalizar:
  - Estos agrupamientos proporcionan una desnormalización física de las tablas (están físicamente agrupadas), pero no lógica (son tablas independientes)

Otras Estrategias de Diseño Físico (I)

- Compresión de datos
  - ↓ almacenamiento, ↑ procesamiento
- *Striping* de datos
  - Múltiples discos
  - RAID (Redundant Array of Independent Disks)
- Replicación/redundancia
  - P.ej., mirroring, duplicar ciertos datos de una tabla a otra, almacenar atributos derivados
  - ↓ disponibilidad y fiabilidad, ↑ coste de mantenimiento de la consistencia

Otras Estrategias de Diseño Físico (II)

- Parámetros de bajo nivel (dependientes del SGBD)
  - PCTFREE: mínimo porcentaje de espacio de un bloque reservado como espacio libre para posibles actualizaciones sobre tuplas del bloque
  - PCTUSED: después de que un bloque se llena hasta el límite especificado por PCTFREE, el bloque se considera no disponible para nuevas inserciones hasta que se esté por debajo del porcentaje marcado por PCTUSED
  - Para extensiones (*extents*): INITIAL (número de bloques inicial), NEXT (número de bloques en una siguiente ampliación), PCTINCREASE (% de crecimiento de las extensiones), MINEXTENTS, MAXEXTENTS
  - DB_BLOCK_SIZE, DB_FILE_MULTIBLOCK_READ_COUNT, ...

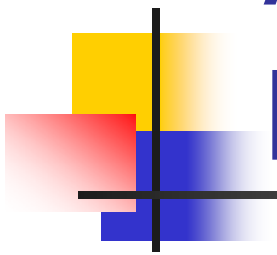
Extensión = unidad lógica de espacio de almacenamiento compuesta por un número de bloques de datos contiguos

Otras Estrategias de Diseño Físico (III)

- Parámetros de bajo nivel. Ejemplo:

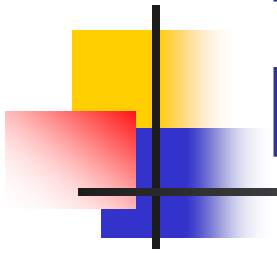
```
CREATE TABLE Profesores (
   codProfesor NUMBER(10) PRIMARY KEY,
   nombre VARCHAR2(30) NOT NULL,
   titulacion VARCHAR2(30) NOT NULL,
   PCTFREE 20
   PCTUSED 15
   TABLESPACE MIS_DATOS
   STORAGE
      (INITIAL 200K
       NEXT 300K
       MAXEXTENTS 10
       PCTINCREASE 30)
);
```

Algunos Consejos Sencillos Para Mejorar el Rendimiento

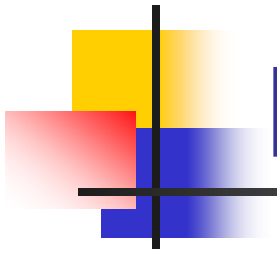


- Monitorizar el uso de índices, crear nuevos si es necesario, eliminar los que no se utilicen, reconstruir índices
- Mantener las estadísticas de la BD actualizadas
- Escribir consultas sencillas, re-escribir consultas + eficientes
- Crear tablas temporales para agrupar resultados intermedios comunes (problema: actualización)
- Combinar varios UPDATE en uno solo
- Proyectar sólo los atributos necesarios
- En caso de problemas (o al menos la primera vez), analizar los planes de ejecución (*EXPLAIN PLAN* en Oracle)
- ...

Herramientas de Apoyo al Diseño Físico



- DB2 Design Advisor (IBM, para DB2)
- SQL Access Advisor (Oracle)
- SQL Tuning Advisor (Oracle)
- Database Engine Tuning Advisor (Microsoft, para SQL Server)
- …



Monitorización del Rendimiento

- IBM (DB2): DB2 Instrumentation Facility
- Oracle: Automatic Workload Repository (AWR), utlbstat.sql / utlestat.sql, Oracle SQL Analyze
- Microsoft (SQL Server): Performance Monitor, PSSDIAG, sqldiag
- …

Contar y Muestrear (I)

- Interesante como ayuda al diseño físico:
  - Para determinar el interés de un índice
  - Para determinar el interés de una vista materializada
  - ...

```
SELECT COUNT(*) AS numDistintosValoresAtributo1
FROM (SELECT DISTINCT atributo1, FROM MiTabla);
```

```
SELECT COUNT(*) AS numDistintosValoresCombinAtributos1y2
FROM (SELECT DISTINCT atributo1, atributo2 FROM MiTabla);
```

```
SELECT COUNT(*) AS cardinalidadDeMiTabla
FROM MiTabla;
```

Contar y Muestrear (II)

- Muestrear
  - Para tener una estimación aproximada con menor coste
  - A veces es suficiente con saber si un valor (por ejemplo, una cuenta) es grande o pequeño, pero no el valor exacto
- En Oracle:

```
SELECT AVG(salario)
FROM Empleados SAMPLE(5);
```

Muestrea un 5% de las tuplas

```
SELECT AVG(salario)
FROM Empleados SAMPLE BLOCK(15);
```

Muestrea un 15% de las bloques de tuplas

```
SELECT COUNT(*) AS cuenta
FROM (SELECT DISTINCT edad
   FROM Empleados SAMPLE(5)
);
```

Estimación del número de valores distintos = cuenta * (1/tasa de muestreo)
(como poco, muestrear un 5%)

Contar y Muestrear (III)

- En Oracle:
  - *OPTIMIZER_DYNAMIC_SAMPLING*

```
ALTER SESION SET OPTIMIZER_DYNAMIC_SAMPLING=9;
```

Niveles de 0 a 10
(0 → no muestreado dinámico)

Diseño Físico y Análisis de Planes de Ejecución (I)

- Para determinar el impacto de los cambios en el diseño físico:
  - Evaluación experimental con volúmenes de datos y carga de trabajo reales → costoso
  - Análisis del impacto en los planes de ejecución
- Herramientas:
  - Oracle:
    - *Explain Plan*
    - *Graphical Explain Plan* (Oracle Enterprise Manager)
  - *Microsoft SQL Server Management Studio*
  - DB2:
    - *IBM Data Studio*
    - *Optim Query Tuner for DB2 for z/OS*
    - *Optim Query Workload Tuner for DB2 for z/OS*

Diseño Físico y Análisis de Planes de Ejecución (II)

- Oracle:

```
EXPLAIN PLAN FOR
<sentencia SQL>
```

```
EXPLAIN PLAN
SET STATEMENT_ID = <identificador> FOR
<sentencia SQL>
```

El resultado se almacena en la tabla *PLAN_TABLE*

```
SET AUTOTRACE ON
```

Opciones: OFF, ON, TRACEONLY, EXPLAIN, STATISTICS

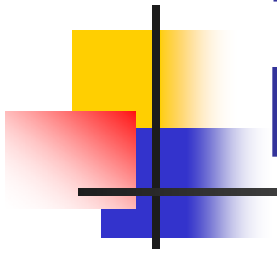
Diseño Físico y Análisis de Planes de Ejecución (III)

- Algunos elementos a vigilar en el plan de ejecución:
  - Procesamientos secuenciales de tablas (*table scans*), salvo si la consulta necesita procesar la tabla completa o si la tabla es pequeña
  - Ordenaciones (*sorts*)
  - Consumo de recursos elevado (CPU, I/O, red), tiempo elevado
  - Elementos de diseño físico no utilizados (índices, vistas materializadas, etc.)
  - ...

Diseño Físico y Análisis de Planes de Ejecución (IV)

- ¿Y si la cosa no va bien? (con precaución...)
  - Cambiar la profundidad de búsqueda del optimizador de consultas (incrementándola o decrementándola)
  - Modificar/actualizar las estadísticas referentes a los objetos accedidos
    - Muchos SGBD ofrecen estas estadísticas a través de vistas modificables por el usuario
    - P.ej., decrementando el tamaño de un índice lo haremos "más apetecible"
  - Proporcionar pistas (*query hints*) al optimizador de consultas

Diseño Físico y Análisis de Planes de Ejecución (V)



- Probar distintas decisiones de diseño físico en una BD con grandes volúmenes de datos es costoso:
  - Una posibilidad es trabajar sin datos e inicializar estadísticas reales en el catálogo
    - El optimizador utilizará esas estadísticas
  - Otra posibilidad es crear “objetos virtuales”, creados únicamente como definiciones para un análisis “qué pasa si...”

Hints para el Optimizador de Consultas (Oracle)

- El que define una consulta puede ofrecer información adicional
- *Hints* en comentarios (entre /* y */ o después de --), precedidos de +, y después de DELETE, INSERT, SELECT, o UPDATE
- Permiten especificar:
  - La aproximación de optimización a aplicar
  - El objetivo de la optimización basada en costes
  - El camino de acceso a utilizar para una relación
  - El orden de joins a considerar
  - El algoritmo de join a aplicar
  - Aspectos de paralelización
- *Stored outlines* (para asegurar la estabilidad de los planes)

Complejo, para más detalles consultar el manual de Oracle (anexo)