[75.15] Bases de Datos

María Inés Parnisari

3 de marzo de 2015

Índice

Parte I

Introducción a las Bases de Datos

1 Modelo de datos

Un fenómeno o idea usualmente refiere a un objeto y a algún aspecto del objeto, el cual captura un determinado valor en un cierto momento.

Dato tupla <nombre de objeto, propiedad de objeto, valor de la propiedad del objeto, instante>.

Dato elemental terna <nombre de objeto, propiedad de objeto, valor de la propiedad del objeto>.

Modelo de datos herramienta intelectual que provee una interpretación del mundo real. Es un dispositivo de abstracción.

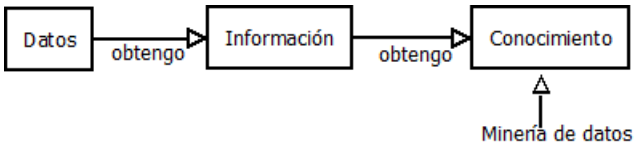


Figura 1: Datos, información y conocimiento

Ejemplos de modelos de representación simbólica de la información: lenguaje natural, fórmulas matemáticas, mapas, partituras.

La pieza más elemental de información es el **dato**. Un dato solo es de utilidad si está vinculado a otros datos. La estructura y el contexto le dan significado a los datos.

2 BDs y SGBDs

Sistema de Base de Datos está formado por cuatro componentes:

1. Hardware
2. Software
3. Datos
4. Personas

   a) **Administradores**: definen el esquema de la base de datos, definen permisos de acceso, ejecutan mantenimientos de rutina (backups, ampliando la capacidad de los discos, monitoreando la performance)

   b) **Usuarios**

Tipo de usuario	Descripción	Interfaz que utiliza
Ingenuos	Consultan la base de datos	Rellenando formularios de consulta, o leyendo reportes generados de la base de datos. Usan aplicaciones
Sofisticados	Consultan la base de datos mediante un lenguaje, sin programar aplicaciones	▪ Herramientas OLAP (*Online Analytical Processing*) ▪ Herramientas de *data mining*
Programador de aplicaciones	Crean programas para acceder a la base de datos	Herramientas RAD *(Rapid Application Development)*
Especializados	Programan aplicaciones especiales (*knowledge bases*, sistemas expertos, etc.)	

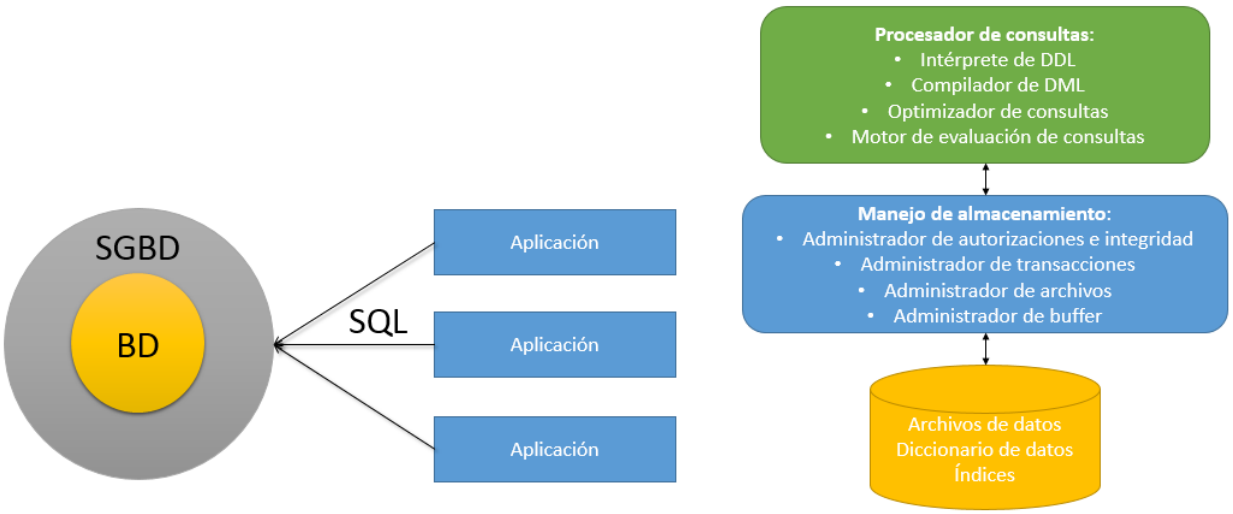
Sistema de Gestión de Base de Datos (SGBD) *Ejemplos: Oracle, MySQL, SQL Sever, DB2, Access, PosgreSQL.* Sistema formado por datos y los programas que acceden a los datos. Su objetivo es almacenar y permitir consultas en formas convenientes y eficientes.

Está formado por tres capas, las **abstracciones**:

1. *Nivel exterior*: "vistas" que ven los usuarios. Solo ven una parte de la base de datos, la que les interesa
2. *Nivel lógico*: cómo se agrupan lógicamente los datos, las entidades y sus relaciones. *Ejemplo: tablas, grafos.*
3. *Nivel físico*: como se organizan físicamente los datos. *Ejemplos: bitmaps, árboles.*

Gestión de Base de Datos está formada por las siguientes herramientas:

1. *Compiladores*: proveen el lenguaje de consulta.
2. *Catálogo*: provee información de la base de datos; por ejemplo, para sacar estadísticas de consultas.
3. *Optimizador de consultas*
4. *Manejador de transacciones*: asegura que una transacción se ejecute completamente o permita una vuelta atrás.
5. *Manejador de concurrencia*: asegura la integridad de la base de datos ante usuarios concurrentes.
6. *Manejador de seguridad*:
   a) Contra hechos no intencionales. *Ejemplo: fallas de energía.*
   b) Contra hechos intencionales. *Ejemplo: ataques de terceros para robar información.*



(a) Los datos se almacenan en una base de datos, que es administrada por un sistema de gestión de bases de datos (b) Estructura general de un sistema de base de datos

Figura 2: SGBD

2.1 Base de datos

Base de Datos (BD) conjunto de datos interrelacionados.

Instancia de la base de datos colección de información en la base de datos en un momento particular.

Esquema de la base de datos descripción de la base de datos. La base de datos física varía con el tiempo (porque se producen altas, bajas y modificaciones), pero el esquema es fijo (o se cambia en raras ocasiones).

- Modelo conceptual: representación de alto nivel, que concentra los requerimientos del cliente. Utiliza el modelo Entidad-Relación.
- Modelo lógico: representación de bajo nivel. Utiliza el modelo relacional.

- Modelo físico: utiliza **SQL**.

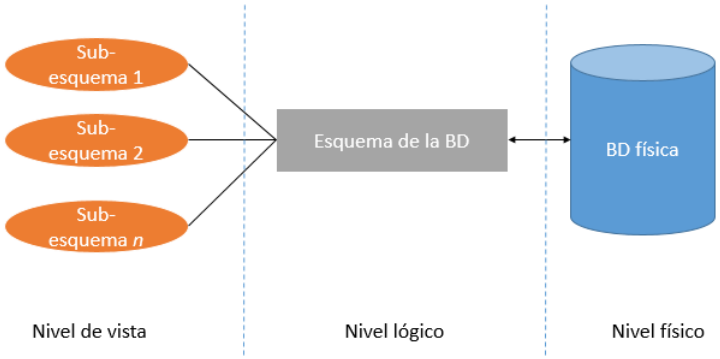


Figura 3: Base de datos

2.2 ¿Por qué una base de datos y no un sistema de archivo?

Desventajas de usar un sistema de archivo:

- Inconsistencia y redundancia, porque se usan muchos archivos, y tal vez tienen distintos formatos
- Dificultad para acceder a los datos, porque se necesitan programas para acceder a los archivos
- Problemas de integridad, porque es difícil cambiar todos los programas para que adopten restricciones de consistencia
- Dificultad para asegurar la atomicidad de las transacciones
- Problemas para garantizar el acceso concurrente
- Problemas de seguridad

2.3 Diseño de una base de datos

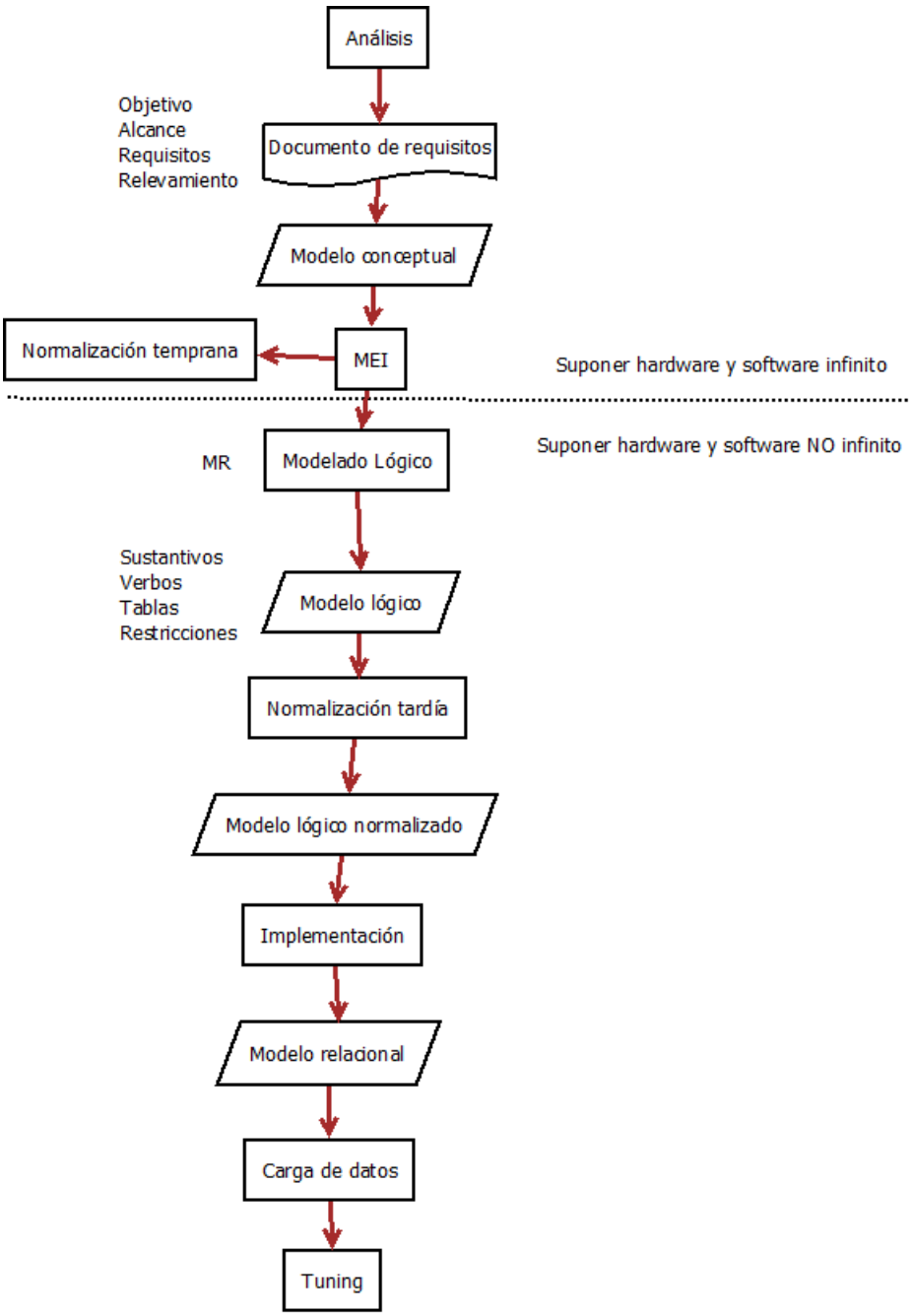


Figura 4: Metodología de diseño de una base de datos

2.4 Modelos de datos

Modelo de datos guía para organizar los datos de una base de datos. Formado por tres componentes:

1. la *estructura* de los datos (tablas, árboles, redes, etc.),
2. las *operaciones* sobre las estructuras, y
3. las *restricciones* sobre las estructuras o sobre las operaciones (para que los datos mantengan su integridad semántica).

La **potencia semántica** de un modelo de datos está determinada por su capacidad de representar, además de la estructura de los datos, el significado de los mismos y sus interrelaciones.

2.4.1 Operaciones en una base de datos

Estado de una BD valor de todos los atributos de todos los objetos de la base de datos.

Operadores los hay de dos tipos:

1. De consulta (no modifican el estado de la base de datos)
2. De actualización (modifican el estado de la base de datos)

Transacción conjunto de operaciones elementales. *Ejemplo: la transferencia a una cuenta.*

```
leer(cuenta_a)
cuenta_a <- cuenta_a + 100
escribir(cuenta_a)
leer(cuenta_b)
cuenta_b <- cuenta_b - 100
escribir(cuenta_b)
```

2.4.2 Restricciones

Tipos de restricciones sobre una base de datos:

- **Inherentes**: determinado por la estructura de la base de datos
- **Explícitas**: provienen del mundo real
- **Implícitas**: se derivan de las explícitas
- **De estado**: restringen los posibles valores de los atributos. *Ejemplo: la edad de una persona no puede ser negativa.*
- **De actualización**: restringen las posibles operaciones sobre los atributos. *Ejemplo: no se puede disminuir la edad de una persona.*
- **De entidad**: *Ejemplo: no puede haber dos objetos iguales en la misma base de datos.*
- **De integridad referencial**: *Ejemplo: no se puede referenciar a un objeto que no existe en la base de datos.*

2.5 Modelos de bases de datos

1. Entidad-Interrelación: para el diseño conceptual
2. Relacional: para el modelo lógico. La relaciones se representan a través de claves foráneas
3. Jerárquico (árbol): el acceso a niveles inferiores sólo es posible a través de sus padres
4. De red: las relaciones entre los datos se representan con referencias físicas
5. De objetos: existe una persistencia de los objetos más allá de la existencia en memoria

3 Modelo Entidad-Interrelación (E-R)

Nota importante: no podemos tener nombres de entidades o interrelaciones repetidos.

3.1 Entidades

El modelo entidad-interrelación se basa en la idea de que el mundo real está compuesto por objetos (las *entidades*) y las *interrelaciones* que hay entre las entidades.

Entidad algo que existe, concreto o abstracto. *Ejemplo: una persona Juan, la cuenta bancaria n° 4500011.*

Conjunto de entidades / tipo de entidad conjunto de entidades del mismo tipo. *Ejemplo: el conjunto de entidades "cliente" abarca todas las personas que tienen cuenta en un banco.*

Un conjunto de entidades E tiene un predicado asociado p para probar si una entidad e pertenece al conjunto.

$$E = \{e/p(e)\}$$

Los conjuntos de entidades no necesitan ser disjuntos. *Ejemplo: "animales" y "mamíferos" no son disjuntos.*

Un tipo de entidad queda definido por:

1. **Nombre**: en singular o frase simple en singular. *Ejemplos: "cliente", "cuenta corriente"*
2. **Significado**: texto preciso, conciso y claro. Debe indicar, si existieran, dependencias con otros tipos de entidades
3. **Atributos**: características de todas las entidades de ese tipo. *Ejemplo: para el conjunto de entidades "cliente", posibles atributos son "nombre", "DNI", "calle", "ciudad", etc.* Cada atributo tiene un conjunto de **valores** permitidos, denominado el **dominio** del atributo. *Ejemplo: el DNI debe ser un número positivo.* Los atributos deben ser **atómicos** (es decir, que no se pueden descomponer), y no pueden estar repetidos. Formalmente, un **atributo** es una función que hace corresponder un conjunto de entidades a un dominio o producto cartesiano de dominios. *Ejemplo: una entidad "empleado" está descrita por el conjunto {(nombre, Juan), (apellido, Cancela), (DNI, 5.678.901)}.*

   Un atributo puede tener el valor **nulo**, por varias razones:

   a) No aplica
   b) Faltante
   c) Desconocido

   Tipos de atributos:

   a) *Simple vs compuesto. Ejemplo: nombre puede descomponerse en primer nombre, segundo nombre, y apellido.*
   b) *Un valor vs Multivaluado*: se repite varias veces en la entidad. *Ejemplo: número de teléfono.*
   c) *Derivable*: su valor puede calcularse a partir de otros atributos. Debe evitarse. *Ejemplo: si tengo el atributo "fecha de nacimiento", "edad" es un atributo derivable.*
4. **Identificador**: atributo o conjunto de atributos que permiten distinguir a cada entidad dentro del conjunto de entidades del mismo tipo.

3.2 Interrelaciones

Interrelación asociación entre dos o más conjuntos de entidades.

Conjunto de interrelaciones / tipo de interrelación conjunto de interrelaciones del mismo tipo. Si $E_1, E_2, \ldots, E_n$ son conjuntos de entidades, entonces el conjunto de interrelaciones R es un subconjunto del producto cartesiano

$$\{(e_1, e_2, \ldots, e_n)\,/e_1 \in E_1, e_2 \in E_2, \ldots, e_n \in E_n\}$$

donde la tupla $(e_1, e_2, \ldots, e_n)$ es una interrelación.

Figura 5: "Tenencia" es una interrelación entre dos entidades

Un tipo de interrelación puede tener atributos descriptivos. *Ejemplo: la interrelación "tenencia" puede tener un atributo "fecha último movimiento".*

Características:

Grado de una interrelación cantidad de tipos de entidades que participan en la relación.

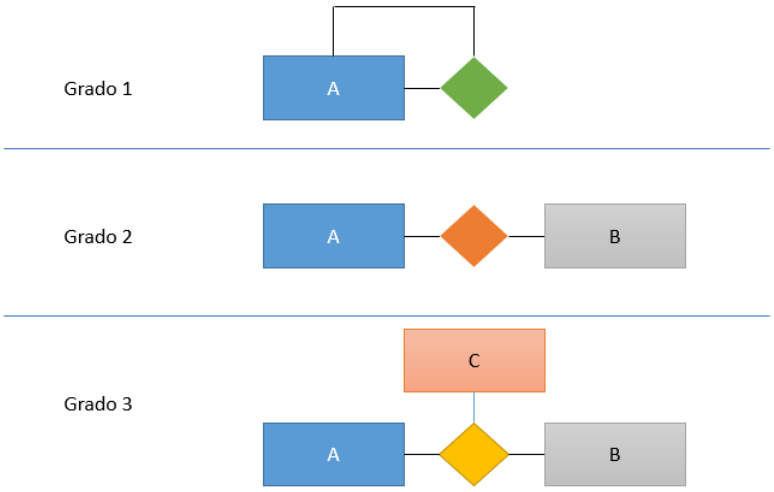


Figura 6: Grado de una interrelación

Cardinalidad de una interrelación cantidad de tipos de entidades con las que puede estar relacionado.

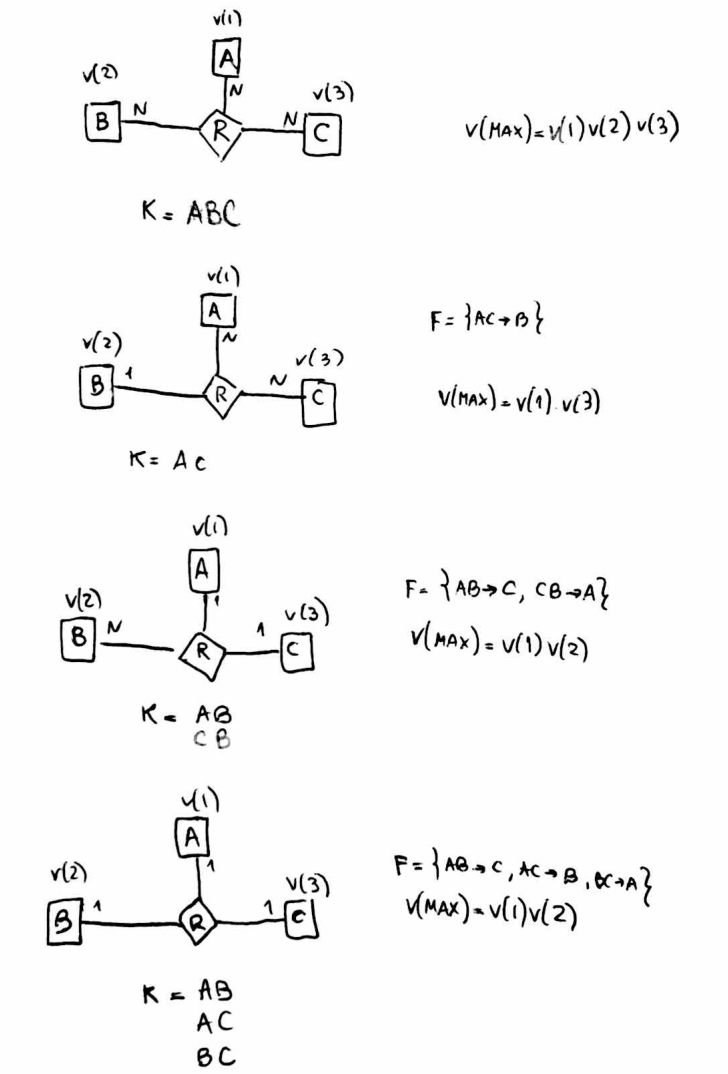
Rol función que desempeña una entidad en una interrelación. Normalmente no se especifica porque está implícito. *Ejemplos: un cliente "tiene" una cuenta, un empleado "es jefe de"* n *empleados.*

Instancia de una interrelación asociación entre dos o más entidades. Debe ser unívocamente identificable s*in usar los atributos descriptivos de la interrelación.*

Un tipo de interrelación puede identificarse por:

1. Un atributo propio, o
2. Una clave primaria, formada por los atributos de las claves primarias de los tipos de entidad que asocian. *Ejemplo: "DNI" es la clave primaria de "cliente", y "nro cuenta" es la clave primaria de "cuenta". La clave primaria del tipo de interrelación "tenencia" es (DNI, nro cuenta)*

La estructura de la clave primaria de una interrelación R depende de las restricciones de cardinalidad del conjunto de interrelaciones.



3.2.1 Tipos de interrelaciones

Especialización un conjunto de entidades se divide en grupos de entidades que son distintivas. *Ejemplo: en el contexto de un banco, una persona puede ser "especializada" en un empleado o un cliente, o ambas, o ninguna*

Generalización la inversa de especialización. Existe una superclase y una o más subclases.

Puede haber restricciones de pertenencia:

1. ¿Qué tipos de entidades pueden ser miembros de una subclase?
   a) Definido por condición: existe una condición o predicado explícito que decide la pertenencia o no pertenencia.
   b) Definido por el usuario: el usuario de la BD asigna entidades a una subclase.
2. ¿Las entidades pueden pertenecer a más de una subclase dentro de una superclase?
   a) Sí $\implies$ Generalizaciones **superpuestas**
   b) No $\implies$ Generalizaciones **disjuntas**
3. ¿Una entidad en la superclase debe pertenecer a al menos una entidad en la subclase?
   a) Sí $\implies$ Generalización **total**
   b) No $\implies$ Generalización **parcial**

En la especialización y en la generalización, las subclases *heredan atributos* de la superclase.

Agregación abstracción mediante la cual un conjunto de interrelación es tratado como una superclase.

3.3 Restricciones

3.3.1 Restricciones de participación

Se dice que la participación de un conjunto de entidades E en un conjunto de interrelaciones R es **total** cuando cada entidad en E participa en al menos una interrelación en R. Si solo algunas entidades participan, la participación es **parcial**.

3.3.2 Restricciones de cardinalidad de correspondencia

Cardinalidad de correspondencia cantidad mínima y máxima de entidades a las cuales puede asociarse una entidad a través de un tipo de interrelación.

Para una interrelación *binaria* R entre dos tipos de entidades A y B, la cardinalidad puede ser:

1. Uno a uno $(1:1)$: Una entidad en A está asociada como máximo a una entidad en B, y una entidad en B está asociada como máximo a una entidad en A.

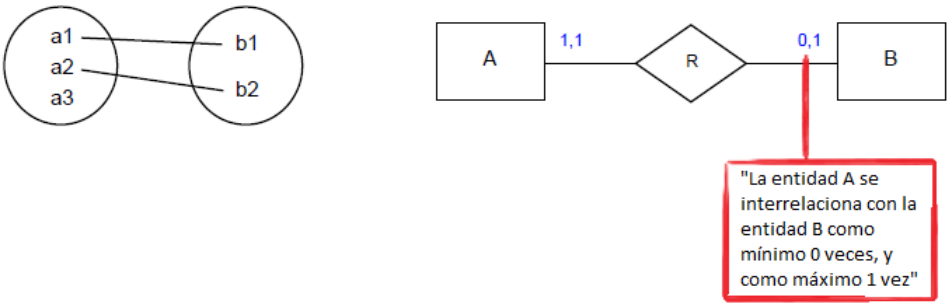


Figura 7: $1:1$

2. Uno a muchos $(1:M)$: Una entidad en A está asociada como máximo con cualquier cantidad de entidades en B, y una entidad en B está asociada como máximo a una entidad en A.
   También existe la restricción "muchos a uno" $(M:1)$.

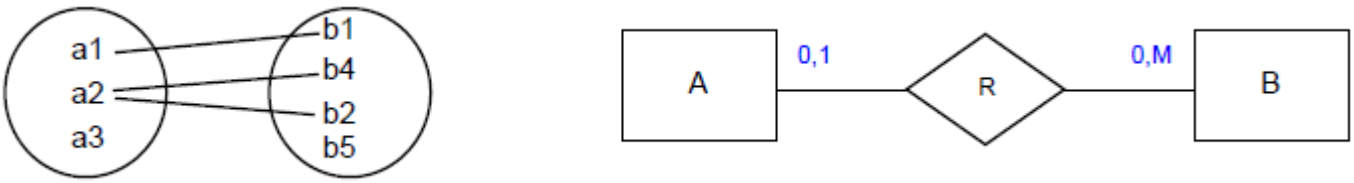


Figura 8: $1:M$

3. Muchos a muchos $(M:M)$: Una entidad en A está asociada como máximo con cualquier cantidad de entidades en B, y una entidad en B está asociada como máximo con cualquier cantidad de entidades en A.

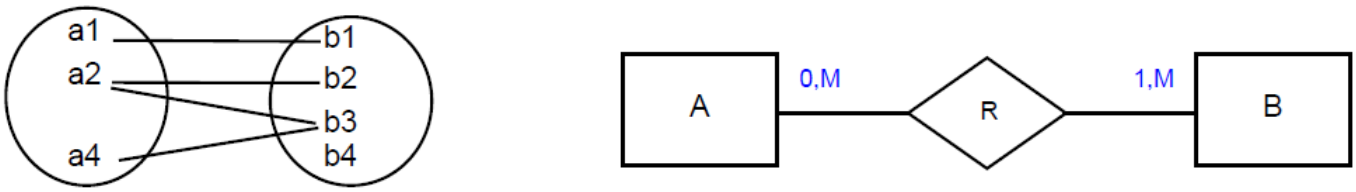


Figura 9: $M:M$

Para una interrelación *ternaria* R entre tres tipos de entidades A, B y C, la cardinalidad puede ser:

- N:N:N

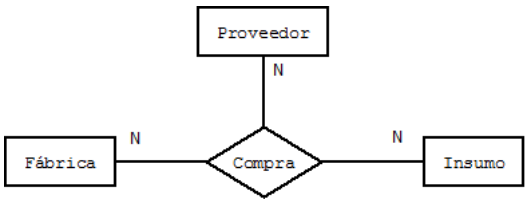


Figura 10: Las fábricas le compran insumos a proveedores.

Claves candidatas de "Compra": $\{id_{fábrica}, id_{insumo}, id_{proveedor}\}$

- N:N:1

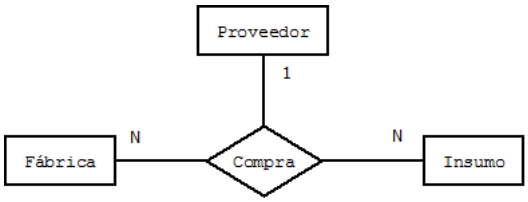


Figura 11: Las fábricas le compran insumos a proveedores. Una fábrica no puede comprar el mismo insumo a más de un proveedor.

Claves candidatas de "Compra": $\{id_{fábrica}, id_{insumo}\}$

- N:1:1

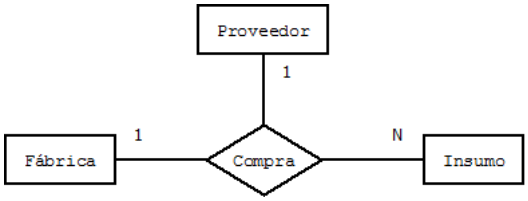


Figura 12: Las fábricas le compran insumos a proveedores. Una fábrica no puede comprar el mismo insumo a más de un proveedor. Un proveedor no puede venderle el mismo insumo a más de una fabrica.

Claves candidatas de "Compra": $\{id_{fábrica}, id_{insumo}\}; \{id_{proveedor}, id_{insumo}\}$

- 1:1:1

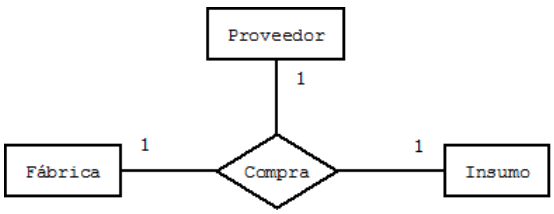


Figura 13: Las fábricas le compran insumos a proveedores. Una fábrica no puede comprar el mismo insumo a más de un proveedor. Un proveedor no puede venderle el mismo insumo a más de una fabrica. Una fábrica no puede comprarle más de un insumo a cada proveedor.

Claves candidatas de "Compra": $\{id_{fábrica}, id_{insumo}\}; \{id_{proveedor}, id_{insumo}\}; \{id_{fábrica}, id_{proveedor}\}$

Teorema: en las relaciones ternarias, las restricciones de cardinalidad no imponen restricciones en las relaciones binarias. *En el ejemplo anterior, una fábrica puede comprar N insumos. Una fábrica puede comprarle a N proveedores. Un proveedor puede proveer N insumos.*

3.3.3 Restricciones de dependencia existencial

Si la existencia de la entidad S depende de la existencia de la entidad D, entonces se dice que S tiene **dependencia existencial** de D. Operacionalmente, esto significa que si D es borrado, también S es borrado. La entidad D es la entidad **dominante** y S es la entidad **subordinada**.

3.3.4 Restricciones de identificación

Superclave conjunto de uno o más atributos que, tomados en conjunto, permiten identificar unívocamente una entidad o una interrelación en el conjunto de entidades o interrelaciones del mismo tipo.

Si K es una superclave, cualquier superconjunto de K también es una superclave.

Clave candidata superclaves minimales; aquellas para las cuales ningún subconjunto propio es una superclave.

Sea R un esquema de relación con atributos $A_1, \ldots, A_n$. El conjunto de atributos $K = (A_1, \ldots, A_k)$ es una clave candidata de R sí y sólo sí satisface:

1. *Unicidad*: en todo momento, no existen dos tuplas distintas de R que tengan el mismo valor para A_i, donde $i \in [1, k]$.
2. *Minimalidad*: ningún atributo de K puede ser eliminado sin que se pierda la propiedad de unicidad.

Clave primaria aquella clave candidata que es elegida por el diseñador del modelo de datos para identificar entidades dentro del conjunto de entidades. Esta clave no debería cambiar en el tiempo. *Ejemplo: la dirección de un cliente no es una buena clave primaria, porque se puede mudar*

Clave foránea atributo o conjunto de atributos que es clave primaria en otra relación. *Ejemplo: en una relación "autos", puede haber una clave foránea "DNI" que es la clave primaria en otra relación "personas", y que indica el dueño de ese auto*

Entidad débil entidad que tiene **dependencia de identificación**, porque no es identificable por sus propios atributos. Está subordinada a otro tipo de entidad, la entidad fuerte.

Entidad fuerte entidad que tiene una clave primaria propia.

El concepto de entidades fuertes y débiles está relacionado con el concepto de dependencia de existencia. Una entidad débil es, por definición, una entidad subordinada a la entidad dominante de la cual depende su identidad. Toda dependencia de identidad es una dependencia de existencia, pero una dependencia de existencia no implica necesariamente una dependencia de identidad.

Discriminador atributo o conjunto de atributos de una entidad débil que lo distingue de otras entidades débiles que dependen de la misma entidad fuerte.

Para una entidad débil, su clave primaria está formada por su identificador más la clave primaria de la entidad fuerte de la cual depende.

3.4 Diagramas E-R

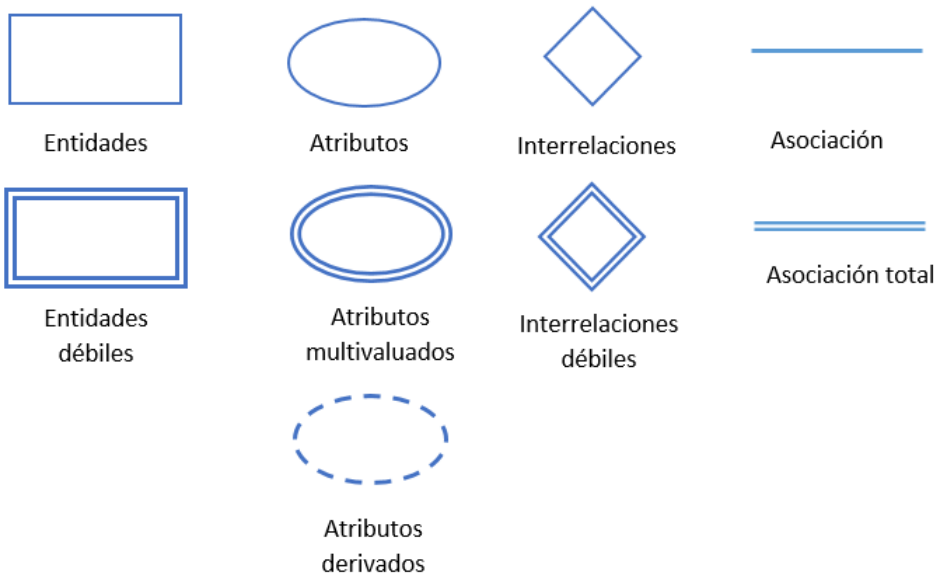


Figura 14: Simbología en los diagramas E-R

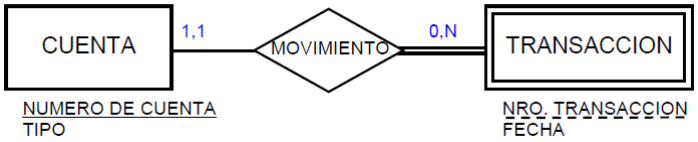


Figura 15: Simbología en los diagramas E-R para entidades fuertes (*cuenta*) y débiles (*transacción*). La clave primaria de la entidad fuerte se identifica con el atributo subrayado (*nro. cuenta*). La dependencia a la entidad fuerte se identifica con arcos dobles. El discriminador de la entidad débil se identifica con el atributo subrayado a medias (*nro. transacción*). El identificador de la entidad débil es la suma de su discriminador y de la clave primaria de la entidad fuerte (*nro. cuenta + nro. transacción*)

3.5 Cuestiones de diseño

1. Conjuntos de entidades vs Atributos?
   a) Algunos atributos no pueden transformarse en entidades. *Ejemplo: el nombre de una persona*
   b) Algunos atributos están mejor representados como entidades, para permitir atributos. *Ejemplo: teléfonos de una persona*
2. Conjuntos de entidades vs Conjuntos de interrelaciones?
   Las interrelaciones modelan "acciones" que ocurren entre entidades.
3. Conjuntos de interrelaciones binarias vs n-arias?
   Teorema es posible reemplazar cualquier conjunto de interrelaciones $n-$aria (con $n > 2$) con un número de interrelaciones binarias.
4. Localización de atributos de interrelaciones
   a) Relaciones $1:1 \implies$ en cualquiera de las dos entidades participantes
   b) Relaciones $1:N \implies$ en la segunda entidad
   c) Relaciones $N:N \implies$ en la interrelación

3.6 Reducción de un esquema E-R a un modelo relacional

Para cada entidad y para cada interrelación, se construye una única tabla. Cada tabla contiene múltiples columnas.

- Representación de conjuntos de entidades fuertes

  Sea E un conjunto de entidad fuerte con atributos $a_1, \ldots, a_n$. Se representa a la misma con una tabla llamada E con n columnas. Cada fila de la tabla es una entidad. El conjunto de todas las posibles filas de la tabla es el **producto cartesiano** entre los dominios de cada atributo:

  $$D_1 \times \ldots \times D_n$$

- Representación de conjuntos de entidades débiles

  Sea A un conjunto de entidad débil con atributos $a_1, \ldots, a_m$. Sea B el conjunto de entidad fuerte del cual depende A. La clave primaria de B está formada por los atributos $b_1, \ldots, b_n$. Se representa a A con una tabla con $m + n$ columnas $a_1, \ldots, a_m, b_1, \ldots, b_n$.

- Representación de conjuntos de interrelaciones

  Sea R un conjunto de interrelaciones entre las entidades $E_1, \ldots, E_n$ con atributos $r_1, \ldots, r_m$. Sean $a_1, \ldots, a_n$ el conjunto de atributos formado por la unión de las claves primarias de cada entidad participante. Se representa a R con una tabla con columnas $a_1, \ldots, a_n, r_1, \ldots, r_m$.

  - Si la interrelación relaciona un conjunto de entidades débil y otro fuerte, la tabla de R no es necesaria, es redundante.

- Representación de interrelaciones $1 : N$ entre entidades A y B

  - Si la participación de A en R es total (i.e. cada entidad en A debe participar en la relación R), se pueden unir las tablas de A y R

- Representación de atributos compuestos

  Sea el atributo compuesto x formado por $x_1, \ldots, x_n$. Se debe generar una columna para cada $x_1, \ldots, x_n$, pero *no* para x.

- Representación de atributos multivaluados

  Sea el atributo multivaluado x. Se debe crear una tabla T con una columna C que corresponde a x y el resto de las columnas son la clave primaria del conjunto de entidades o interrelación del cual x es un atributo.

- Representación de generalización

  Hay dos métodos:

  1. Una tabla para la superclase + una tabla para cada subclase, que incluya una columna con la clave primaria de la superclase
  2. Si la generalización es disjunta y completa, no se crea una tabla para la superclase. Se crea una tabla para cada subclase, que incluya todos los atributos de la superclase.

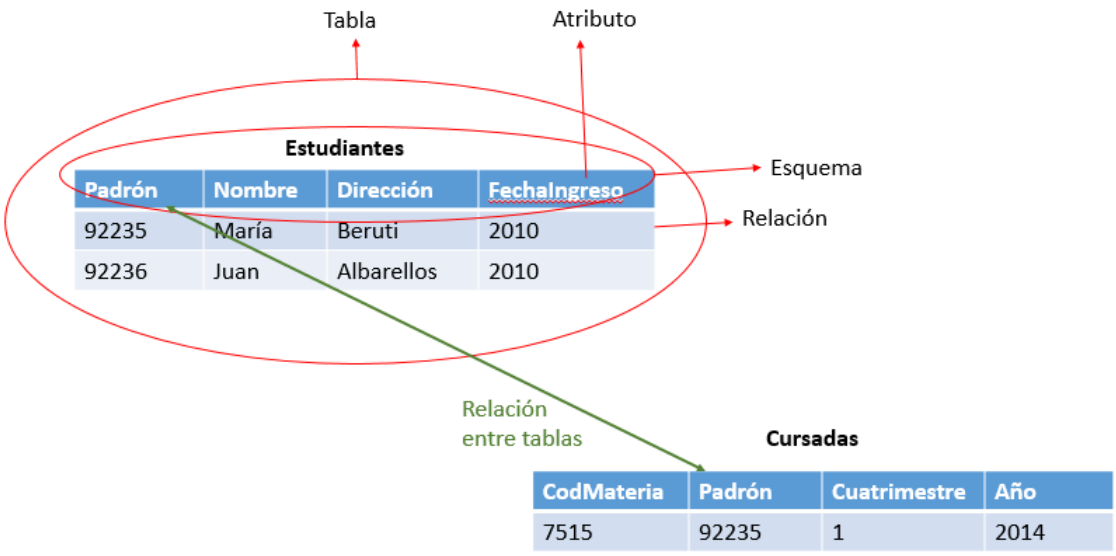
- Representación de agregación

  La tabla de la interrelación R entre la agregación A y el conjunto de entidades B incluye una columna por cada clave primaria de B y A.

4 Modelo Relacional

4.1 Estructura del modelo relacional

El modelo relacional es un ejemplo de modelo basado en registros. Usa un grupo de tablas para representar los datos y las relaciones entre los datos. Cada tabla tiene múltiples columnas. Cada tabla contiene registros de un tipo en particular. Cada registro tiene atributos (las columnas).



Padrón	Nombre	Dirección	FechaIngreso
92235	María	Beruti	2010
92236	Juan	Albarellos	2010

CodMateria	Padrón	Cuatrimestre	Año
7515	92235	1	2014

Figura 16: Tablas

Dominio valores posibles de un conjunto. *Ejemplo:* $D_{padron} = \{1, 2, \dots 100000\}$

Existe un valor que es posible en cualquier dominio: el valor **nulo**. Este valor significa "atributo desconocido" o "atributo inexistente". Se puede prohibir, para un atributo, que este tenga el valor nulo, ya que puede causar problemas al realizar consultas.

Esquema de relación descripción de una relación. Es el nombre de la relación seguido de la lista de los atributos con sus dominios. *Ejemplo: Estudiante (Padrón, nombre, dirección, fechaIngreso)*

Relación instancia de un esquema. Subconjunto del producto cartesiano de una lista de dominios. Sea el esquema de relación R, denotamos a la instancia r mediante $r(R)$.

Tabla representación de una relación. Se pueden relacionar entre sí al compartir atributos. Una tabla de n atributos es un subconjunto de $D_1 \times D_2 \times \cdots \times D_n$. Notar que al ser un conjunto, el orden no interesa.

Atributo nombre de cada columna de una tabla.

Esquema de BD relacional conjunto de esquemas de relación.

Tupla miembro de una relación. Son las filas de una tabla.

4.2 Restricciones del modelo relacional

- Regla de integridad de entidad

  Ningún atributo de la clave primaria de un esquema de relación puede tomar valor nulo.

- Regla de integridad referencial

  Si la relación r incluye una clave foránea F que es la clave primaria P de una relación s, entonces todo valor de F en r debe ser totalmente nulo o ser igual al valor de P en alguna tupla de s.

4.3 Operaciones del modelo relacional: álgebra relacional

Es un lenguaje **formal** y **procedural** que permite realizar consultas sobre un conjunto de relaciones, y produce relaciones como resultado.

4.3.1 Operadores básicos

Los siguientes operadores son linealmente independientes. Se utilizan para filtrar, cortar y combinar relaciones.

1. Operaciones unarias:

a) **Proyección** ($\pi_{a_1,\ldots,a_n}(r)$).
Dados los atributos $a_1, \ldots, a_n$ que pertenecen a r, devuelve una relación de esquema $\{a_1, \ldots, a_n\}$.
Se eliminan las tuplas repetidas.

b) **Selección** ($\sigma_p(r)$).
Devuelve una relación r' con el mismo esquema que r, en la que las tuplas son aquellas pertenecientes a r que satisfacen la condición o el predicado p. La condición puede ser de dos estilos:

1) *Atómica*: expresión lógica simple del tipo que utiliza un operador de comparación θ, tal que $\theta \in \{=, \neq, <, >, \leq, \geq\}$. Se permiten dos tipos de expresiones:
   a′ Atributo θ atributo
   b′ Atributo θ constante
2) *Compuesta*: utiliza conectores *and, or, not*.

c) **Renombre** ($\rho_{R(A_1,A_2,\ldots,A_n)}(E)$).
Cambia el nombre de la relación E a R, y también cambia el nombre de sus atributos a $A_1, A_2, \ldots, A_n$.

2. Operaciones binarias:

a) **Unión** ($r \bigcup s$).
Sean r y s instancias *homogéneas*[1] de las relaciones R y S respectivamente. r contiene m tuplas y s contiene n tuplas. La unión se define como:

$$r \bigcup s = \{x : x \in r \vee x \in s\}$$

Las tuplas repetidas se eliminan.

b) **Diferencia** ($r - s$).
Sean r y s instancias *homogéneas* de las relaciones R y S respectivamente. La diferencia se define como:

$$r - s = \{x : x \in r \wedge x \notin s\}$$

c) **Producto cartesiano** ($r \times s$).
Sean r y s instancias de las relaciones R y S respectivamente. El producto cartesiano es el conjunto de tuplas que resulta de concatenar cada tupla de r con cada tupla de s.

$$r \times s = \{(r_1, s_1); \ldots; (r_1, s_m); \ldots; (r_n, s_1) \ldots, (r_n, s_m)\}$$

Si r tiene n tuplas y s tiene m tuplas, $r \times s$ tendrá $n \cdot m$ tuplas. El grado de $r \times s$ es la suma de los grados de r y s.

4.3.2 Operadores secundarios

No proporcionan nuevas funcionalidades, solo simplifican las consultas.

1. **Intersección** ($r \bigcap s$)
Dadas las relaciones r_1 y r_2 con el mismo esquema, devuelve una relación r con igual esquema, tal que $r = \{x : x \in r_1 \wedge x \in r_2\}$

$$r \bigcap s \equiv r - (r - s)$$

2. **Junta (*join*)** ($r \bowtie s$)
La junta de dos relaciones r y s según un predicado P es una relación de esquema igual al producto cartesiano $r \times s$, cuyas tuplas son el conjunto de tuplas de $r \times s$ que satisfacen el predicado P.

$$r \bowtie_p s \equiv \sigma_p (r \times s)$$

a) ***Equi-join***: dadas las relaciones r y s, es el *join* según el predicado $r.A_i = s.A_j$

[1] Las relaciones homogéneas tienen: (a) la misma cantidad de atributos, y (b) para todo i, el dominio del atributo i de r es el mismo que el dominio del atributo i de s

b) ***Auto-join***: dada la relación r, es el *join* de r consigo mismo según el predicado $1.A_i\,\theta\,2.A_i$ donde 1 y 2 son aliases de r.

c) ***Natural join*** $(r * s)$. Requiere que haya atributos compartidos entre las relaciones r y s, y además que $R \bigcap S \neq \emptyset$.

$$r * s \equiv \pi_{R \cup S} \left(\sigma_{r.c_1 = s.c_1 \wedge r.c_2 = s.c_2 \wedge \cdots \wedge r.c_h = s.c_h} \left(r \times s\right)\right)$$

El *join* natural es una operación asociativa, por lo tanto, $(r * s) * t = r * (s * t)$.

Estudiante

Padrón	Nombre	Ciudad
92000	Juan	Caballito
95000	Pedro	Florida
97000	Lucas	Recoleta
99000	Facundo	San Martín

Cursa

Padrón	Curso
92000	Base de Datos
95000	Algoritmos y Programación IV
99000	Simulación

Estudiante ⋈ Cursa

Padrón	Nombre	Ciudad	Curso
92000	Juan	Caballito	Base de Datos
95000	Pedro	Florida	Algoritmos y Programación IV
99000	Facundo	San Martín	Simulación

Figura 17: Junta natural

3. **División** (r/s).

Sean las relaciones $r \in R$ con n atributos y $s \in S$ con m atributos, tal que $m < n$ y $S \subset R$. La división son todas las tuplas t tales que, multiplicadas por **todas** las filas u de s, me dan filas que están en r. El esquema de la división es $R - S$.

$$r/s \equiv \pi_{1,\ldots,n-m}(r) - \pi_{1,\ldots,n-m}\left(\left(\pi_{1,\ldots,n-m}(r) \times s\right) - r\right)$$

Puede Volar

Piloto	Tipo de Avión
Acosta	B757
Acosta	B777
Acosta	B787
Díaz	B767
Díaz	B777
Fernández	B747
Fernández	B777
Martínez	B747
Martínez	B777
Martínez	B787

Aviones

Tipo de Avión
B777
B787

Puede Volar / Aviones

Piloto
Acosta
Martínez

Figura 18: División

4. **Asignación** ($r \leftarrow s$)

   r no cambia al ejecutar operaciones de actualización sobre s.

4.4 Operadores extendidos

1. **Proyección generalizada**: permite operadores aritméticos como parte de una proyección.

$$\pi_{F_1,F_2,\ldots,F_n}(E)$$

   donde E es una expresión de álgebra relacional, y F_i es una expresión aritmética que involucra constantes y atributos del esquema E.

2. **Funciones agregadas**: son funciones que toman como parámetro un multiconjunto[2].

```
sum
count
min
max
avg
```

   La forma general es:

$$_{G_1,\ldots G_n}\mathcal{G}_{F_1(A_1),\ldots,F_m(A_m)}(E)$$

   donde E una expresión de álgebra relacional; G_i es un atributo para formar grupos, F_i es una función agregada, y A_i es un atributo. Las tuplas de la relación resultado de E se particionan en grupos de tal forma que

   a) Todas las tuplas en un grupo tienen el mismo valor para $G_i \, \forall i \in [1, n]$

   b) Tuplas en distintos grupos tienen distinto valor para $G_i \, \forall i \in [1, n]$

   Para cada grupo, la relación tendrá una tupla $(g_1, \ldots, g_n, a_1, \ldots, a_m)$ donde para cada i, a_i es el resultado de aplicar la función F_i en el multiconjunto, para el atributo A_i del grupo.

   Ejemplo: dada la relación EmpleadosPartTime(nombre,sucursal,salario), la siguiente expresión devuelve una relación con un solo atributo (sin nombre) y una sola fila, que contiene el valor de la suma de todos los salarios:

$$\mathcal{G}_{sum(salario)}EmpleadosPartTime$$

   Se puede usar el operador `distinct` para eliminar múltiples ocurrencias de un valor.

   Ejemplo: con la misma relación anterior, la siguiente expresión devuelve una relación con un solo atributo (sin nombre) y una sola fila, que contiene la cantidad de sucursales:

$$\mathcal{G}_{count-distinct(sucursal)}EmpleadosPartTime$$

   Se puede usar el operador de **grupos** para particionar una relación y computar una función agregada a cada grupo.

   Ejemplo: con la misma relación anterior, la siguiente expresión devuelve una relación con dos atributos (sin nombres), que contiene la suma de salarios de cada sucursal:

$$_{sucursal}\mathcal{G}_{sum(salario)}EmpleadosPartTime$$

3. **Join externo** (*outer join*): es una extensión de la operación de *join* que lidia con información faltante (*null*s).

   a) *Left outer join* ($r_1 ⟕ r_2$): hace un *join natural*, y toma todas las tuplas de r_1 que no matchearon con ninguna tupla de r_2, les agrega valores *null* a los atributos de r_2, y los agrega al resultado.

   b) *Right outer join* ($r_1 ⟖ r_2$): hace un *join natural*, y toma todas las tuplas de r_2 que no matchearon con ninguna tupla de r_1, les agrega valores *null* a los atributos de r_1, y los agrega al resultado.

[2] Un **multiconjunto** es un conjunto en el que un valor puede aparecer muchas veces.

c) *Full outer join* ($r_1 ⟗ r_2$): combina las dos operaciones anteriores.

Empleado

employee-name	street	city
Coyote	Toon	Hollywood
Rabbit	Tunnel	Carrotville
Smith	Revolver	Death Valley
Williams	Seaview	Seattle

employee-name	branch-name	salary
Coyote	Mesa	1500
Rabbit	Mesa	1300
Gates	Redmond	5300
Williams	Redmond	1500

TrabajaEn

Empleado FOJ TrabajaEn

employee-name	street	city	branch-name	salary
Coyote	Toon	Hollywood	Mesa	1500
Rabbit	Tunnel	Carrotville	Mesa	1300
Williams	Seaview	Seattle	Redmond	1500
Smith	Revolver	Death Valley	*null*	*null*
Gates	*null*	*null*	Redmond	5300

Figura 19: Ejemplo de *full outer join*

4.5 Actualizaciones a bases de datos

4.5.1 Delete

En álgebra relacional se expresa como $r \leftarrow r - E$, donde r es una relación y E es una expresión de consulta.
Solo se pueden borrar tuplas completas, no valores de atributos.

4.5.2 Insert

En álgebra relacional se expresa como $r \leftarrow r \cup E$, donde r es una relación y E es una expresión de consulta, o una relación constante con una sola tupla.

4.5.3 Update

En álgebra relacional se expresa como $r \leftarrow \pi_{F_1,\ldots,F_n}(r)$, donde r es una relación y cada F_i es, o el atributo i-ésimo de r, o una expresión que involucra solo constantes y el nuevo valor de r, que da el nuevo valor del atributo i-ésimo.

4.6 Vistas (*views*)

Vista relación que no es parte del modelo lógico de la base de datos, pero que es visible para el usuario como una relación "virtual".

```
CREATE VIEW v AS <expresion de consulta>
```

Limitaciones:

- No se pueden ejecutar operaciones de *update* sobre una vista, salvo que la vista sobre la relación R cumpla lo siguiente:
  - Deben usar `SELECT`, y no `SELECT DISTINCT`
  - La cláusula `WHERE` no debe incluir a R en una consulta anidada
  - La cláusula `WHERE` no debe incluir a otra relación
  - Los atributos del `SELECT` deben ser `NOT NULL`

  El *update* solo se ejecutará sobre los atributos del `SELECT`.
- A las operaciones de *delete* sobre una vista se les pasa la cláusula `WHERE` de la vista (para borrar solo las tuplas que se pueden ver en la vista).
- Si las relaciones subyacentes son modificadas, la vista queda desactualizada. Por eso, en los motores, en una expresión que involucra una vista, la relación de la vista se recalcula al evaluar la expresión.

Vista materializada vista que se evalúa y se almacena físicamente. Los cambios a estas vistas se realizan de forma *incremental* (no se reconstruye toda la vista desde cero).

5 Cálculo relacional

COMPLETAR

6 Lenguajes de consulta

Query expresión que solicita una porción de información.

Hay dos tipos de lenguajes de consulta (*query languages*):

1. **Procedurales:** el usuario indica una serie de operaciones a ejecutar sobre una base de datos para producir el resultado deseado. *Ejemplos: SQL.*
2. **No procedurales:** el usuario indica la información deseada, sin especificar el procedimiento para obtenerla. *Ejemplos: Datalog (similar a Prolog).*

7 Integridad y seguridad

En SQL, tenemos las siguientes restricciones:

- `NOT NULL`: indica que un atributo no puede tener el valor *null*
- `UNIQUE`: asegura que para todas las tuplas, el valor del atributo es único
- `PRIMARY KEY`: asegura que una o varias columnas tengan una identidad única, no nula
- `FOREIGN KEY`: asegura la integridad referencial de una tabla
- `CHECK`: asegura que el valor de un atributo satisface una condición especificada
- `DEFAULT`: especifica un valor por defecto cuando no se especifique ninguo

7.1 Restricciones de dominio

```
CREATE DOMAIN color AS VARCHAR(10)
        DEFAULT 'SinColor'
        CHECK (value IN ('SinColor', 'Azul', 'Rojo'));

CAST relacion.Atributo AS <dominio>;

DROP DOMAIN <dominio>;

ALTER DOMAIN <dominio>;
```

7.2 Restricciones de integridad referencial

Ejemplo:

```
CREATE TABLE hospital (
        IdHosp SMALLINT,
        PRIMARY KEY (IdHosp));

CREATE TABLE medico (
        IdHosp SMALLINT,
        IdMedico SMALLINT,
        Nombre VARCHAR(40)
        PRIMARY KEY (IdMedico));

ALTER TABLE medico
        ADD CONSTRAINT FK_medico
        FOREIGN KEY IdHosp
        REFERENCES hospital(IdHosp);
```

7.2.1 Acciones referenciales

Son llamadas SQL que se ejecutan de forma automática ante la eliminación o actualización de una restricción de integridad referencial, para mantener la misma.

```
ALTER TABLE medico
        ADD CONSTRAINT FK_medico
        FOREIGN KEY IdHosp
        REFERENCES hospital(IdHosp)
                ON [DELETE|UPDATE] [SET DEFAULT|SET NULL|CASCADE|NO ACTION];
```

De esta forma, cuando se elimine o actualice una fila en la tabla `Hospital`, el sistema de base de datos ejecutará la acción especificada en la tabla `Medico`:

- `SET DEFAULT / SET NULL`: se fija el valor predeterminado o nulo en la fila referenciante.
- `CASCADE`: al eliminar una tupla referenciada, las tuplas referenciantes son eliminadas. Al modificar la clave en la tabla referenciada, el correspondiente valor de la clave foránea es actualizado.

7.3 Seguridad y autorizaciones

Para brindar autorizaciones, se brindan privilegios. Los privilegios son tipos de autorizaciones para leer, insertar, actualizar o borrar datos. También existe el privilegio `references`, que permite declarar claves foráneas al crear relaciones nuevas.

Con la opción `WITH GRANT OPTION` se permite que el usuario que recibe el permiso se lo conceda a alguien más. El pase de autorizaciones de un usuario a otro se representa con un **grafo de autorización**, donde cada nodo es un usuario (la raíz es el DBA), y las aristas representan autorizaciones concedidas.

Un usuario tiene autorización sí y solo sí existe un camino desde la raíz del grafo hacia el nodo usuario.

```
GRANT <privilegios>
        ON <[relacion|vista]>
        TO <[usuario|rol]>
        [WITH GRANT OPTION];
```

Para eliminar autorizaciones se usa el comando `REVOKE`. Por defecto, este comando tiene el efecto de revocar los privilegios que el usuario le brindó a otros. Para evitar este efecto cascada, se usa la opción `RESTRICT`.

```
REVOKE <privilegios>
        ON <[relacion|vista]>
        FROM <[usuario|rol]>
        [RESTRICT];
```

Para crear roles:

```
CREATE ROLE instructor;
```

8 Triggers

Trigger comando que el sistema ejecuta automáticamente si un *evento* satisface cierta *condición*. Son un conjunto de acciones; por ejemplo, enviar un e-mail, ejecutar el *rollback* de una transacción, o crear una copia de la base de datos.

```
CREATE TRIGGER chequear_horario
[BEFORE|AFTER] [UPDATE|INSERT|DELETE] [OF atributo] ON curso
WHEN condicion
BEGIN
        acciones
END
```

Parte II

Diseño de bases de datos

Objetivos:

- Reducir la redundancia de la información (porque ocupa espacio y es más difícil de actualizar): repetición de información en varias tuplas
- Eliminar las anomalías:
  - De actualización: podría darse el caso de que actualizamos una tupla y no actualizamos otra tupla con el mismo dato
  - De inserción: no es posible insertar ciertos datos sin insertar otros (posiblemente no relacionados)
  - De eliminación: podría darse el caso de que eliminamos una tupla y, sin querer, borramos un dato importante
- Devolver información rápidamente

Notación:

- Nombre de esquemas: $R, S, R_1, \ldots, R_n, S_1, \ldots, S_n$
- Instancias de esquemas: $r, s, r_1, \ldots r_n, s_1, \ldots, s_n$
- Conjuntos de atributos: α, β, γ
- Atributos: A, B, C
- Tuplas: t_1, t_2

8.1 Dependencias funcionales

Dependencia funcional sea el esquema R, y sean $\alpha \subseteq R$ y $\beta \subseteq R$. La dependencia funcional $\alpha \to \beta$ se cumple si para cada par de tuplas t_1, t_2 tal que $t_1[\alpha] = t_2[\alpha]$, también se cumple que $t_1[\beta] = t_2[\beta]$.

Una dependencia funcional es **trivial** si $\beta \subseteq \alpha$. En particular, $\alpha \to \alpha$ es trivial.

Un conjunto K es superclave de R si $K \to R$.

Un conjunto J es clave candidata de R si $J \to R$ y además $\nexists \gamma \subset J / \gamma \to R$.

Una dependencia funcional, a diferencia de una diferencia multivaluada o de junta, prohíbe que ciertas tuplas existan.

Dependencias funcionales implicadas sea el esquema R. Una dependencia funcional f está lógicamente implicada por un conjunto de dependencias F, si para cada instancia de relación $r(R)$ que cumple F, también cumple f.

8.1.1 Axiomas de Armstrong

Se utilizan para hallar las dependencias funcionales implicadas, es decir, F^+ de un conjunto F.

1. Regla de reflexividad

   Si α es un conjunto de atributos y $\beta \subseteq \alpha$, entonces $(\alpha \to \beta) \in F^+$
2. Regla de augmentación

   Si $\alpha \to \beta$ es una dependencia funcional y γ es un conjunto de atributos, entonces $(\gamma\alpha \to \gamma\beta) \in F^+$
3. Regla de transitividad

   Si $\alpha \to \beta$ es una dependencia funcional y $\beta \to \gamma$ es otra dependencia funcional, entonces $(\alpha \to \gamma) \in F^+$

Propiedades deducidas:

1. Regla de unión

   Si se cumplen $\alpha \to \beta$ y $\alpha \to \gamma$, entonces también se cumple $\alpha \to \beta\gamma$

2. Regla de descomposición

   Si se cumple $\alpha \to \beta\gamma$, entonces también se cumplen $\alpha \to \beta$ y $\alpha \to \gamma$

3. Regla de pseudotransitividad

   Si se cumplen $\alpha \to \beta$ y $\gamma\beta \to \delta$, entonces también se cumple $\gamma\alpha \to \delta$

Teorema: R satisface la dependencia funcional $X \to Y$ sí y sólo sí R descompone sin pérdida sobre los esquemas XY y $X(R-XY)$, es decir, si R satisface la dependencia de junta $*(XY, X(R-XY))$

8.1.2 Clausura y superclaves

Clausura de F sea F un conjunto de dependencias funcionales sobre el esquema R. F^+ es el conjunto de dependencias funcionales implicadas por F.

$$F^+ = \{\alpha \to \beta / F \vDash \alpha \to \beta\}$$

Algoritmo 1 Computar F^+

Entrada: relación R, conjunto de dependencias funcionales F
Salida: F^+

1. $F^+ = F$
2. Repetir hasta que F^+ no varíe:
   a) Para cada dependencia funcional f en F^+:
   Aplicar reglas de reflexividad y aumentación a f
   Agregar las dependencias funcionales restantes a F^+
   b) Para cada par de dependencias funcionales f_1, f_2 en F^+:
   - Si f_1, f_2 pueden combinarse usando el axioma de transitividad:
     Agregar la dependencia funcional resultante a F^+
3. Devolver F^+

Para un esquema R con n atributos, hay 2^{n+1} posibles dependencias funcionales.
Clausura de α bajo F: es el conjunto de atributos de R que dependen funcionalmente de α.

$$\alpha_F^+ = \{\beta / \beta \subset R \wedge F \vDash \alpha \to \beta\}$$

Algoritmo 2 Computar α^+

Entrada: relación R, conjunto de dependencias funcionales F, atributo α
Salida: α_F^+

1. $a^+ = a$
2. Repetir hasta que a^+ no varíe:
   a) Para cada dependencia funcional $B \to Y \in F$:
   - Si $B \subseteq a^+$:
     $a^+ = a^+ \cup Y$
3. Devolver a^+

Usos del algoritmo de cálculo de la clausura de a^+:

1. Para determinar si α es una superclave de R: debe cumplirse que $\alpha^+ = R$.
2. Para determinar si la dependencia funcional $\alpha \to \beta \in F^+$: debe cumplirse que $\beta \subseteq \alpha^+$.
3. Es una forma alternativa de calcular F^+

Algoritmo 3 Cálculo de claves candidatas mediante un grafo

Entrada: relación R, conjunto de dependencias funcionales F
Salida: claves candidatas de R

1. Dibujar el grafo de dependencias funcionales (si $X \to Y \in F$, dibujar una arista entre X e Y)
2. $V_{ni} =$ atributos sin aristas entrantes
3. $V_{oi} =$ atributos que solo tienen aristas entrantes
4. $K = \{\}$
5. Repetir hasta que no se pueda agregar nada a K:
   a) $CC =$ todos los atributos de V_{ni}
   b) Agregar a CC un atributo de R que no esté en V_{oi} ni V_{ni}
   c) $K = K \cup CC$
6. Devolver K

8.1.3 Cubrimiento minimal

Atributo extráneo atributo de una dependencia funcional que se puede quitar de la misma sin cambiar la clausura del conjunto de dependencias funcionales.

Sea el conjunto de dependencias funcionales F y la dependencia funcional $\alpha \to \beta$ en F.

- El atributo A es extráneo en α si, cuando lo sacamos de α, la dependencia funcional se sigue cumpliendo ($B \in (\alpha - A)^+_F$). Formalmente:
  - $A \in \alpha$, y
  - F implica lógicamente a $(F - \{\alpha \to B\}) \cup \{(\alpha - A) \to \beta\}$
- El atributo B es extráneo en β si, cuando lo sacamos de β, la dependencia funcional se sigue cumpliendo ($\alpha \to (\beta - B)$) Formalmente:
  - $B \in \beta$, y
  - El conjunto $(F - \{\alpha \to \beta\}) \cup \{\alpha \to (\beta - B)\}$ implica lógicamente a F

Cubrimiento minimal El cubrimiento minimal F_{min} para F es un conjunto de dependencias funcionales tales que F implica todas las dependencias en F_{min}, y F_{min} implica todas las dependencias en F.

F_{min} debe tener las propiedades siguientes:

- Ninguna dependencia funcional en F_{min} contiene un atributo extráneo
- No existen dos dependencias funcionales $\alpha_1 \to \beta$ y $\alpha_2 \to \gamma$ en F_{min} tales que $\alpha_1 = \alpha_2$. Es decir, los lados izquierdos deben ser únicos

Algoritmo 4 Cálculo de cubrimiento minimal

Entrada: conjunto de dependencias funcionales F
Salida: cubrimiento minimal F_{min}

1. Usar el axioma de la descomposición para dejar un solo atributo en el lado derecho de cada dependencia funcional.
2. Eliminar los atributos extráneos de los lados izquierdos.

   Sea $Y \to B$ una dependencia funcional en F, con al menos dos atributos en Y. Sea Z igual a Y pero con algún atributo de menos. Si F implica a $Z \to B$, entonces reemplazar $Y \to B$ con $Z \to B$.
3. Eliminar las dependencias funcionales redundantes (es decir, que se deducen de los axiomas de Armstrong).

Ejemplo: sea el conjunto de dependencias $F = \{A \to BD, B \to CD, AC \to E\}$. Calcular el cubrimiento minimal.

1. *Dejar todos los lados derechos con un único atributo.*

   $F_{min} = \{A \to B, A \to D, B \to C, B \to D, AC \to E\}$
2. *Eliminar todos los atributos redundantes del lado izquierdo.*

   a) *Hay que analizar solamente la dependencia que tiene lado izquierdo compuesto: $AC \to E$.*

   b) *¿C es redundante en $AC \to E$? Hay que verificar si E está en A_F^+. Como $A_F^+ = ABCDE$, entonces C es redundante en $AC \to E$ y podemos reemplazar esta dependencia funcional por $A \to E$.*

   $F_{min} = \{A \to B, A \to D, B \to C, B \to D, A \to E\}$
3. *Eliminar las dependencias redundantes.*

   - *Hay que revisar una por una todas las dependencias de F_{min}. En caso de que una sea redundante, se la elimina de F_{min} y se sigue analizando el resto tomando como referencia el nuevo F_{min}.*
   - *¿$A \to D$ es redundante en F_c? Hay que verificar si D está en $A^+_{F_c - \{A \to D\}}$*
     - $F_{min} - \{A \to D\} = \{A \to B, B \to C, B \to D, A \to E\}$
     - $A^+_{F_{min} - \{A \to D\}} = ABCDE$
     - *Como $D \in F_{min} - \{A \to D\}$, $A \to D$ es redundante*

   $F_{min} = \{A \to B, B \to C, B \to D, A \to E\}$

   Puede comprobarse que el resto de las dependencias funcionales no es redundante.

8.2 Primera forma normal (1FN)

Dominio atómico los elementos del dominio se consideran unidades indivisibles.

Una relación con esquema R está en 1FN si los dominios de sus atributos son atómicos, y si no hay atributos similares repetidos.

Ejemplo: la siguiente relación no está en 1FN.

Título Libro	Autor 1	Autor 2	Autor 3
Database System Concepts	Avi Silberschatz	Henry Korth	S. Sudarshan
Applied Cryptography	Bruce Schneier	NULL	NULL

Para normalizarla, hay que dividirla en dos relaciones.

ID libro	Título Libro
1	Database System Concepts
2	Applied Cryptography

ID libro	Autor
1	Avi Silberschatz
1	Henry Korth
1	S. Sudarshan
2	Bruce Schneier

8.3 Descomposición de una relación

Descomposición de una relación Sea el esquema relacional R. El conjunto de esquemas $\{R_1, R_2, \ldots, R_n\}$ es una descomposición de R si

$$R = R_1 \cup R_2 \cup \cdots \cup R_n$$

Sea la relación $r(R)$. Sean $r_i = \pi_{R_i}(r)$ para $i = 1, 2, \ldots, n$. Es decir, $\{r_1, r_2, \ldots, r_n\}$ es la base de datos que resulta de descomponer R en $\{R_1, R_2, \ldots, R_n\}$. Siempre se verifica que

$$r \subseteq r_1 \bowtie r_2 \bowtie \ldots \bowtie r_n$$

Es decir, r es un subconjunto de la junta de la descomposición (la junta podría tener tuplas que no están en r).

Propiedades deseadas de la descomposición:

1. **Sin pérdida de información:** verifica que $r = \pi_{R_1}(r) \bowtie \pi_{R_2}(r) \bowtie \ldots \bowtie \pi_{R_n}(r)$.
2. **Preservación de dependencias**

   Una descomposición que verifica que $F'^{+} = F^{+}$ preserva dependencias, donde $F' = F_1 \cup F_2 \cup \cdots \cup F_n$ son las dependencias proyectadas.

   Esta propiedad es deseable porque significa que podemos chequear si se cumple la dependencia con solo verificarla en una relación individual, y no necesitamos hacer un *join* de las relaciones de la descomposición.
3. Poca o nula redundancia de información

Algoritmo 5 Cálculo de la proyección de dependencias F_i

Entrada: relación R, relación R_i, conjunto de dependencias funcionales F
Salida: conjunto de dependencias funcionales que se verifican en R_i (F_i)

1. $F_i = \{\}$
2. Para cada conjunto de atributos X que es un subconjunto de atributos de R_i:
   a) Calcular X_F^+
   b) $F_i = F_i \cup \{X \to A\}$, tal que $A \subseteq X_F^+$ y $A \subseteq R_i$
3. Devolver F_i

Algoritmo 6 Cálculo para saber si una descomposición preserva una dependencia

Entrada: dependencia funcional $A \to B$, conjunto de dependencias funcionales F, descomposición $\{R_1, \ldots, R_n\}$
Salida: "verdadero" si la descomposición preserva $A \to B$

1. $Z = A$
2. Repetir hasta que Z no varíe o hasta que Z incluya a B:
   - Para cada R_i: $Z = Z \cup \left[(Z \cap R_i)^+ \cap R_i\right]$
3. Si $B \subseteq Z$: devolver "verdadero"
4. Si no: devolver "falso".

8.3.1 Descomposición SPI en 2 relaciones

Sea el esquema relacional R, y sea F el conjunto de dependencias funcionales sobre R. Sean R_1 y R_2 una descomposición de R. Esta descomposición es *sin pérdida de información* (SPI) si al menos una de las siguientes dependencias funcionales está en F^+:

- $R_1 \cap R_2 \to R_1 - R_2$

- $R_1 \cap R_2 \to R_2 - R_1$

Ejemplo: ¿La descomposición de $R(A,B,C,D,E,F)$ en $R_1 = (A,D,E,F)$ y $R_2 = (B,C,D)$ es SPI respecto de $F = \{A \to BD, B \to CD, AC \to E\}$?

- $R_1 \cap R_2 = D$
- $R_1 - R_2 = AEF$
- $R_2 - R_1 = BC$
- *Como $D_F^+ = \{D\}$, vemos que no se cumplen ni $R_1 \cap R_2 \to R_1 - R_2$ ni $R_1 \cap R_2 \to R_2 - R_1$. Por lo tanto, esta descomposición no es SPI.*

8.3.2 Descomposición SPI en más de 2 relaciones

El algoritmo **Chase Tableau** se utiliza para verificar si una descomposición de R en $R_1, \ldots, R_k$ $(k > 2)$ es SPI. Es decir, si la proyección de un esquema restringido por dependencias en una descomposición, puede recuperarse haciendo el *join* de las proyecciones.

Algoritmo 7 Algoritmo Chase Tableau

Entrada: esquema de relación R, conjunto de dependencias funcionales F, descomposición $\{R_1, \ldots, R_k\}$
Salida: "verdadero" si $\{R_1, \ldots, R_n\}$ descompone SPI.

1. Armar una matriz $T^{(0)}$ con las relaciones R_i en las filas $(i = 1, \ldots, k)$ y los atributos A_j en las columnas. Cada elemento de la matriz tendrá valor:

$$m_{ij} = \begin{cases} a_j & \text{si } A_j \in R_i \text{ (variable distinguida)} \\ b_{ij} & \text{si } A_j \notin R_i \text{ (variable ligada)} \end{cases}$$

2. $i = 1$
3. Obtener $T^{(i)}$:
   - Para cada dependencia funcional $X \to Y \in F$:
     - Si $\exists$ tuplas t_1, t_2 tal que $t_1[X] = t_2[x]$, realizar un cambio de variable en $t_1[Y]$ y en $t_2[Y]$
       - Variables ligadas por distinguidas
       - Si hay dos ligadas, poner la de menor subíndice
4. Si hay una fila sólo con variables distinguidas, devolver "verdadero".
5. Si $T^{(i)} = T^{(i-1)}$ y además no hay una fila sólo con variables distinguidas, devolver "falso".
6. Si no, $i \leftarrow i + 1$ y volver a 3.

Ejemplo: sea el esquema de relación $R(A,B,C,D)$ con $F = \{A \to B, B \to C, CD \to A\}$. Sea la descomposición $R_1 = (A,D)$, $R_2 = (A,C)$, $R_3 = (B,C,D)$. ¿La descomposición es SPI?

	A	B	C	D
$R_1(A,D)$	a_1	b_{12}	b_{13}	a_4
$R_2(A,C)$	a_1	b_{22}	a_3	b_{24}
$R_3(B,C,D)$	b_{31}	a_2	a_3	a_4

Aplicamos $A \to B$. Como a_1 es igual en las primeras dos filas, pero b_{12} y b_{22} no, cambiamos b_{22} por b_{12} en la segunda fila.

	A	B	C	D
$R_1(A,D)$	a_1	b_{12}	b_{13}	a_4
$R_2(A,C)$	a_1	$\mathbf{b_{12}}$	a_3	b_{24}
$R_3(B,C,D)$	b_{31}	a_2	a_3	a_4

Aplicamos $B \to C$. Las filas 1 y 2 coinciden en b_{12} pero no en c. Entonces, cambiamos b_{13} por a_3 en la fila 1.

	A	B	C	D
$R_1(A,D)$	a_1	b_{12}	$\mathbf{a_3}$	a_4
$R_2(A,C)$	a_1	b_{12}	a_3	b_{24}
$R_3(B,C,D)$	b_{31}	a_2	a_3	a_4

Aplicamos $CD \to A$. Las filas 1 y 3 tienen mismo valor para c y d, pero distinto valor para a. Por lo tanto, cambiamos b_{31} por a_1 en la tercera fila.

	A	B	C	D
$R_1(A,D)$	a_1	b_{12}	a_3	a_4
$R_2(A,C)$	a_1	b_{12}	a_3	b_{24}
$R_3(B,C,D)$	$\mathbf{a_1}$	a_2	a_3	a_4

En este punto notamos que la fila 3 tiene valores de la forma a_j. Como son todas variables distinguidas, la descomposición es sin pérdida de información.

8.4 Segunda forma normal (2FN)

Atributo primo atributo de R que pertenece a una clave candidata.

Sea R una relación y F el conjunto de sus dependencias funcionales. R estará en 2FN si para toda dependencia funcional $X \to Y$ que está en en F^+, se cumple al menos una de las siguientes condiciones:

1. $X \to Y$ es trivial
2. Y es un atributo primo,
3. X no es un subconjunto de una clave candidata

Ejemplo: la siguiente relación no está en 2FN porque *Dirección de local* depende sólo de *ID de local*, que no es una clave.

ID de cliente	ID de local	Dirección de local
1	1	Los Angeles
1	3	San Francisco
2	1	Los Angeles

$F = \{Id\,cliente, Id\,local \to Direccion\,local; Id\,local \to Direccion\,local\}$. *La segunda dependencia funcional no es trivial, el lado derecho no pertenece a una clave candidata, y el lado izquierdo es un subconjunto de la clave candidata.*

Para normalizarla, hay que dividirla en dos relaciones.

ID de cliente	ID de local
1	1
1	3
2	1

ID de local	Dirección de local
1	Los Angeles
3	San Francisco

8.5 Forma Normal Boyce-Codd (FNBC)

Un esquema de relación R está en FNBC con respecto a un conjunto de dependencias funcionales F si, para todas las dependencias funcionales en F^+ de la forma $\alpha \to \beta$, donde $\alpha \subseteq R$ y $\beta \subseteq R$, se cumple alguna de las siguientes:

- $\alpha \to \beta$ es trivial (es decir, $\beta \subseteq \alpha$)
- α es una superclave para R

Propiedades:

- Cualquier relación de dos atributos está en FNBC
- Brinda eliminación de anomalías
- Brinda una descomposición SPI
- No garantiza que se preserven las dependencias funcionales

Para verificar si una dependencia funcional $\alpha \to \beta$ viola FNBC, alcanza con computar α^+ y ver si es R.
Para verificar si una relación R está en FNBC, alcanza con ver si las dependencias de F no violan FNBC.
Para verificar si una descomposición R_i está en FNBC, para cada subconjunto α de atributos de R_i, verificar que α_F^+ o no incluya atributos de $R_i - \alpha$, o incluya todos los atributos de R_i.

Algoritmo 8 Descomposición FNBC

Entrada: una relación R, un conjunto de dependencias funcionales F
Salida: una descomposición $\rho = \{R_1, \ldots, R_n\}$ tal que R_i está en FNBC con respecto a F_i

1. $\rho = R$
2. Repetir hasta que no haya esquemas en ρ que violen FNBC
   a) Elegir una dependencia funcional $X \to Y$ que viole FNBC sobre ρ_i
   b) Descomponer ρ_i en dos relaciones:
      - $R_1(XY)$
      - $R_2(X(R - XY))$
   c) En ρ, reemplazar ρ_i por R_1 y R_2
3. Devolver ρ

8.6 Tercera forma normal (3FN)

> *A statement of Codd's definition of 3NF, paralleling the traditional pledge to give true evidence in a court of law, was given by Bill Kent: "[Every] non-key [attribute] must provide a fact about the key, the whole key, and nothing but the key." A common variation supplements this definition with the oath: "so help me Codd".*
>
> *Requiring existence of "the key" ensures that the table is in 1NF; requiring that non-key attributes be dependent on "the whole key" ensures 2NF; further requiring that non-key attributes be dependent on "nothing but the key" ensures 3NF.*

Un esquema de relación R está en 3FN con respecto a un conjunto de dependencias funcionales F si, para todas las dependencias funcionales en F^+ de la forma $\alpha \to \beta$, donde $\alpha \subseteq R$ y $\beta \subseteq R$, se cumple alguna de las siguientes:

- $\alpha \to \beta$ es trivial (es decir, $\beta \subseteq \alpha$)
- α es una superclave para R
- β es un atributo primo

Dicho de otra forma, R no debe tener dependencias funcionales transitivas: todos los atributos que no son clave deben depender funcionalmente de la clave primaria.
Propiedades:

- No tiene pérdida de información
- No hay pérdida de dependencias
- No garantiza la eliminación de anomalías (es decir, tiene redundancia)
- Es menos estricta que FNBC (FNBC $\implies$ 3FN)

Algoritmo 9 Descomposición 3FN

Entrada: una relación R, un cubrimiento minimal F_{min}
Salida: una descomposición $\rho = \{R_1, \ldots, R_n\}$ tal que R_i está en 3FN con respecto a F_i

1. $\rho = \{\}$
2. Para cada dependencia funcional $X \to Y \in F_{min}$:
   $\rho = \rho \cup R_i\,(XY)$
3. Si existen $R_i, R_j \in \rho$ tal que $R_i \subseteq R_j$:
   $\rho = \rho - R_i$
4. Si ninguna relación es una superclave de R, y K es una clave para R
   $\rho = \rho \cup K$
5. Devolver ρ

Ejemplo: la siguiente relación no está en 3FN porque *Fecha de nacimiento del ganador* depende sólo de *Ganador*, que depende de la clave candidata *{Torneo, Año}*.

Torneo	Año	Ganador	Fecha de nacimiento de ganador
Indiana Invitational	1998	Al Fredrickson	21 Julio 1975
Cleveland Open	1999	Bob Albertson	28 Septiembre 1968
Des Moines Masters	1999	Al Fredrickson	21 Julio 1975
Indiana Invitational	1999	Chip Masterson	14 Marzo 1977

$F = \{Torneo, Año \to Ganador, Ganador \to Fecha\,nacimiento\,ganador\}$. La segunda dependencia funcional no es trivial, el lado derecho no pertenece a una clave candidata (no es atributo primo), y el lado izquierdo no es una superclave.

Para normalizarla, hay que dividirla en dos relaciones.

Torneo	Año	Ganador
Indiana Invitational	1998	Al Fredrickson
Cleveland Open	1999	Bob Albertson
Des Moines Masters	1999	Al Fredrickson
Indiana Invitational	1999	Chip Masterson

Ganador	Fecha de nacimiento
Chip Masterson	14 Marzo 1977
Al Fredrickson	21 Julio 1975
Bob Albertson	28 Septiembre 1968

8.7 Dependencias multivaluadas

Una dependencia multivaluada expresa que dos conjuntos de atributos en un esquema son mutuamente independientes.

Sean los conjuntos de atributos *disjuntos* A y B de la relación $R(A, B, U - A - B)$. Sean las tuplas t_1 y t_2. La dependencia multivaluada $A \twoheadrightarrow B$ indica que si $t_1[A] = t_2[A]$, entonces podemos encontrar tuplas t_3 y t_4 tales que:

- $t_1[A] = t_2[A] = t_3[A] = t_4[A]$
- $t_3[B] = t_1[B]$
- $t_3[U - A - B] = t_2[U - A - B]$
- $t_4[B] = t_2[B]$
- $t_4[U - A - B] = t_1[U - A - B]$

	α	β	$R-\alpha-\beta$
t_1	$a_1 \dots a_i$	$a_{i+1} \dots a_j$	$a_{j+1} \dots a_n$
t_2	$a_1 \dots a_i$	$b_{i+1} \dots b_j$	$b_{j+1} \dots b_n$
t_3	$a_1 \dots a_i$	$a_{i+1} \dots a_j$	$b_{j+1} \dots b_n$
t_4	$a_1 \dots a_i$	$b_{i+1} \dots b_j$	$a_{j+1} \dots a_n$

Figura 20: Dependencia multivaluada $\alpha \twoheadrightarrow \beta$

Ejemplo: *sea $R(curso, libro, profesor)$ una relación que indica una lista de cursos universitarios, los libros que se usan en el curso, y los profesores que la dictan.*

Curso	*Libro*	*Profesor*
Álgebra II	David Lay - Algebra Lineal	Acero
Álgebra II	Strang - Algebra Lineal	Acero
Álgebra II	David Lay - Algebra Lineal	Piortrowsky
Algebra II	Strang - Algebra Lineal	Piortrowsky

Dado que los profesores de un curso y los libros de un curso son independientes entre sí, esta relación tiene una DMV. Si tuviésemos que agregar un nuevo libro al curso "Álgebra II", deberíamos agregar una tupla para cada profesor del curso. Es decir, formalmente, hay dos DMV en esta relación: $\{curso\} \twoheadrightarrow \{libro\}$ y $\{curso\} \twoheadrightarrow \{profesor\}$. Esta relación exhibe redundancia.

Dependencia multivaluada no existen en el esquema original pero son satisfechas por la descomposición del mismo.

8.7.1 Propiedades de las DMVs

- $\alpha \twoheadrightarrow \beta$ es **trivial** si $\beta \subseteq \alpha$ ó si $R = \alpha\beta$.
- Una dependencia multivaluada garantiza que ciertas tuplas existan.
- **Regla de transitividad multivaluada**: Si $\alpha \twoheadrightarrow \beta$ y $\beta \twoheadrightarrow c$, entonces $\alpha \twoheadrightarrow c - \beta$
- **Regla de replicación**: si $\alpha \to \beta$, entonces $\alpha \twoheadrightarrow \beta$. La inversa **no** es cierta.
- **Regla de interacción**[3]: si $\begin{cases} \alpha \twoheadrightarrow \beta \\ \gamma \to Z \end{cases}$ y existe un γ tal que $\begin{cases} Z \subseteq \beta \\ \gamma \cap \beta = \emptyset \end{cases}$, entonces $\alpha \to Z$
- **Regla de aumentación**: si $\alpha \twoheadrightarrow \beta$ entonces $\alpha w \twoheadrightarrow \beta$.
- **Regla del complemento**: si $\alpha \twoheadrightarrow \beta$, entonces $\alpha \twoheadrightarrow R - \alpha\beta$
- **Regla de unión**: si $\alpha \twoheadrightarrow \beta$, y $\alpha \twoheadrightarrow c$, entonces $\alpha \twoheadrightarrow \beta c$
- **Regla de descomposición**: si $\alpha \twoheadrightarrow \beta$, y $\alpha \twoheadrightarrow c$, entonces $\begin{cases} \alpha \twoheadrightarrow \beta - c \\ \alpha \twoheadrightarrow c - \beta \\ \alpha \twoheadrightarrow \beta \cap c \end{cases}$
- **Regla de pseudotransitividad:** si $\alpha \twoheadrightarrow \beta$ y $\beta c \twoheadrightarrow \gamma$, entonces $\alpha c \twoheadrightarrow \gamma - \beta c$
- Si $R = \{A_1, \dots, A_n, B_1, \dots, B_m\}$, entonces $\{A_1, \dots, A_n\} \twoheadrightarrow \{B_1, \dots, B_m\}$

Teorema: las únicas DMVs implicadas por un conjunto de DFs son de la forma $X \twoheadrightarrow Y$, donde $Y \subseteq X^+$ o $R - XY \subseteq X^+$.

Teorema: las únicas DFs implicadas por un conjunto de DMVs son las triviales.

Teorema: un conjunto de DFs de tipo $x_i \to y_i$ sólo implica DMVs de tipo $x_i \twoheadrightarrow y_i$.

Teorema: sea r una instancia del esquema de relación R, X, Y subconjuntos de R, y $Z = R - XY$. La relación r satisface la DMV $X \twoheadrightarrow Y$ sí y solo sí $R_1 = XY$ y $R_2 = X \cup Z$ descomponen sin pérdida de información a r.

[3] Esta regla puede agregar dependencias funcionales a M

Algoritmo 10 Verificar si r satisface la dependencia multivaluada $X \twoheadrightarrow Y$

Entrada: relación $r(R)$
Salida: "verdadero" si la dependencia multivaluada $X \twoheadrightarrow Y$ se satisface

1. Proyectar r en $R_1(XY)$ y $R_2(X, R - XY)$
2. Calcular $R_1 \bowtie R_2$
3. Si $r = R_1 \bowtie R_2$ devolver "verdadero"

Algoritmo 11 Proyección de dependencias funcionales y multivaluadas

Entrada: conjunto de dependencias funcionales y multivaluadas D^+, descomposición $\{R_1, \ldots, R_n\}$
Salida: conjunto de dependencia funcionales y multivaluadas D_i que se satisfacen en R_i

1. $Z = \{\}$
2. Agregar a Z las dependencias funcionales de D^+ que solo incluyan atributos de R_i
3. Agregar a Z las dependencias multivaluadas de la forma $A \rightarrow B \cap R_i$, donde $A \twoheadrightarrow B$ esté en D^+ y $A \subseteq R_i$
4. Devolver Z

8.7.2 Preservación de dependencias multivaluadas

Una descomposición de R en los esquemas $R_1, \ldots, R_n$ es una descomposición que **preserva las dependencias** con respecto a un conjunto D de dependencias funcionales y multivaluadas si, para cada relación $r_1(R_1), \ldots, r_n(R_n)$ tal que para cada i, r_i satisface D_i, entonces existe una relación $r(R)$ que satisface D y para el cual $r_i = \pi_{R_i}(r)$ para todo i.

8.8 Base minimal de Dependencias e Implicación de DMVs

Base minimal sea la colección de conjuntos $S = \{S_1, \ldots, S_p\}$ donde $U = S_1 \cup \ldots \cup S_p$. La base minimal de S es $Base(S)$ y es una partición de U $\{T_1, \ldots, T_q\}$ tal que:

1. Cada S_i es la unión de algunos de los T_j.
2. No existe una partición de U con menos elementos que cumpla la primera propiedad.

Ejemplo: sea $S = \{ABCD, CDE, AE\}$ y $U = ABCDE$. Entonces la base de S es $Base(S) = \{A, B, CD, E\}$.

Base de dependencias: sea M un conjunto de dependencias multivaluadas sobre R, y sea $X \subseteq R$. Sea $G = \{Y / M \models X \twoheadrightarrow Y\}$La base de dependencias de X con respecto a M es:

$$Bdep(X) = Base\,(G)$$

Ejemplo: sea $M = \{A \twoheadrightarrow BC, DE \twoheadrightarrow C\}$ un conjunto de dependencias multivaluadas sobre $R(ABCDE)$. Entonces $G = \{A, BC, DE, C, BDE, B, BCDE, CDE\}$y la base de A es

$$Base(A) = \{A, B, C, DE\}$$

.

Algoritmo 12 Cálculo de la base de dependencias de X

Entrada: conjunto de dependencias multivaluadas M, conjunto de atributos X
Salida: $Bdep(X)$

1. $T = R - X$
2. Mientras T varíe:
   - Si $\exists V \in T$ y $Y \twoheadrightarrow Z \in M$ tal que $(V \cap Y = \emptyset)$ y $(V \cap Z \neq \emptyset)$
     Reemplazar V por $\{V \cap Z\}$ y $\{V - Z\}$
3. $Bdep(X) = T \cup \{A / A \in X\}$

Implicación de dependencias: dado un conjunto M de dependencias multivaluadas, una dependencia multivaluada $X \twoheadrightarrow Y$ pertenece a M^+ sí y solo sí Y se puede expresar como la unión de algunos componentes de $Bdep(X)$.

Ejemplo: sea $R(A, B, C, D, E, I)$, $M = \{A \twoheadrightarrow EI, C \twoheadrightarrow AB\}$. *¿Se cumple* $AC \twoheadrightarrow BI$ *?*

La base de dependencias de AC *es:*

1. $T^{(0)} = BDEI$
   a) $V = BDEI$. Entonces $A \twoheadrightarrow EI$, $Y = A$, $Z = EI$
   b) $V \cap Z = EI$
   c) $V - Z = BD$
2. $T^{(1)} = \{EI, BD\}$
   a) Si $V = EI$, $Y = A$, $Z = EI$, $V \cap Z = EI$, $V - Z = \{\}$ y no nos aporta nada.
   b) $V = BD$. Entonces $C \twoheadrightarrow AB$, $Y = C$, $Z = AB$
   c) $V \cap Z = B$
   d) $V - Z = D$
3. $T^{(2)} = \{EI, B, D\}$

$Bdep(AC) = \{EI, B, D, A, C\}$ y entonces $AC \twoheadrightarrow BI$ no se cumple porqueBI no se puede expresar como la unión de componentes de $Bdep(AC)$.

8.9 Cuarta forma normal (4FN)

Un esquema de relación R está en 4FN con respecto a un conjunto de dependencias funcionales y multivaluadas M si, para todas las dependencias multivaluadas en M^+ de la forma $\alpha \twoheadrightarrow \beta$, donde $\alpha \subseteq R$ y $\beta \subseteq R$, se cumple alguna de las siguientes:

- $\alpha \twoheadrightarrow \beta$ es trivial
- α es una superclave para R

Propiedades:

- No tiene pérdida de información
- Una descomposición 4FN no garantiza la preservación de dependencias

Ejemplo: la siguiente relación no está en 4FN. La clave es $\{N°\,Vuelo, Dia\,de\,la\,semana, Tipo\,de\,avión\}$.

N° Vuelo	Dia de la semana	Tipo de avión
106	Lunes	B737
106	Martes	B737
106	Lunes	A380
106	Martes	A380

$F = \{N°Vuelo \twoheadrightarrow Dia\,de\,la\,semana, N°\,Vuelo \twoheadrightarrow Tipo\,de\,avión\}$. La primera dependencia multivaluada no es trivial, y el lado izquierdo no es una superclave de R.

Para normalizarla, hay que dividirla en dos relaciones.

N° Vuelo	Dia de la semana	N° Vuelo	Tipo de avión
106	Lunes	106	B737
106	Martes	106	A380

8.9.1 Descomposición SPI en 2 relaciones

Sea el esquema relacional R, y sea M el conjunto de dependencias funcionales y multivaluadas sobre R. Sean R_1 y R_2 una descomposición de R. Esta descomposición es *sin pérdida de información* (SPI) si al menos una de las siguientes dependencias multivaluadas está en M^+:

1. $R_1 \cap R_2 \twoheadrightarrow R_1 - R_2$
2. $R_1 \cap R_2 \twoheadrightarrow R_2 - R_1$

8.9.2 Descomposición SPI en más de 2 relaciones

El algoritmo es idéntico al de descomposición en BCFN, excepto que se usan dependencias multivaluadas y se restringe R_i al conjunto M^+.

Algoritmo 13 Descomposición 4FN

Entrada: una relación R, un conjunto de dependencias multivaluadas M^+
Salida: una descomposición $\rho = \{R_1, \ldots, R_n\}$ tal que R_i está en 4FN con respecto a M_i

1. $\rho = R$
2. Repetir hasta que no haya esquemas en ρ que violen 4FN
   a) Elegir una dependencia multivaluada $X \twoheadrightarrow Y$ que viole 4FN sobre ρ_i
   b) Descomponer ρ_i en 2 relaciones:
      - $R_1\,(XY)$
      - $R_2\,(X(R - XY))$
   c) En ρ, reemplazar ρ_i por R_1, R_2
3. Devolver ρ

9 Dependencias de junta

Dependencia de junta sea un esquema de relación R y sea la descomposición $R_1, \ldots, R_n$. La dependencia de junta $*(R_1, \ldots, R_n)$ restringe el conjunto de relaciones posibles a aquellas para las cuales $R_1, \ldots R_n$ es una descomposición sin pérdida de información.

Formalmente, si $R = R_1 \cup \cdots \cup R_n$, una relación $r(R)$ satisface la dependencia de junta $*(R_1, \ldots, R_n)$ si

$$r = \pi_{R_1}(r) \bowtie \cdots \bowtie \pi_{R_n}(r)$$

La dependencia de junta $*(R_1, \ldots, R_k)$ se satisface para R si restringe los valores de toda instancia $r(R)$ tal que, si existen k tuplas $t_1, \ldots, t_k$ que satisfacen $t_i\,[R_i \cap R_j] = t_j\,[R_i \cap R_j]$, luego también existe en r otra tupla t_{k+1} definida por $t_{k+1}\,[R_i] = t_i\,[R_i]$ para $1 \leq i \leq k$.

Propiedades:

- Si una de las R_i es R, la dependencia de junta es trivial.
- Toda dependencia de junta $*(R_1, R_2)$ es equivalente a la dependencia multivaluada $R_1 \cap R_2 \twoheadrightarrow R_2$.

- No existen un conjunto de reglas para inferir dependencias de juntas.

Teorema: R satisface la dependencia multivaluada $X \twoheadrightarrow Y$ sí y sólo sí R descompone sin pérdida sobre los esquemas $X \cup Y$ y $X(R - XY)$, es decir, si R satisface la dependencia de junta $*(XY, X(R - XY))$

Teorema: si una relación R puede descomponerse SPI en 3 esquemas, pero no puede hacerlo sobre 2, esa relación solo satisface DMVs triviales.

***Ejemplo**: sea la relación $R(A, B, C)$ y la dependencia de junta $*\left(\underbrace{AB}_{R_1}, \underbrace{BC}_{R_2}, \underbrace{AC}_{R_3}\right)$. ¿Qué tupla debe tener r para satisfacer la misma?*

r	A	B	C
t_1	1	2	3
	4	5	6
t_2	4	2	7
t_3	1	8	7

Dado que se cumplen:

- $t_1[R_1 \cap R_2] = t_2[R_1 \cap R_2] = 2$
- $t_1[R_1 \cap R_3] = t_3[R_1 \cap R_3] = 1$
- $t_2[R_2 \cap R_3] = t_3[R_2 \cap R_3] = 7$

Entonces para que se satisfaga la dependencia de junta deberá existir una tupla t_4 tal que:

- $t_4[R_1] = t_1[R_1] = \{1, 2\}$
- $t_4[R_2] = t_2[R_2] = \{2, 7\}$
- $t_4[R_3] = t_3[R_3] = \{1, 7\}$

Es decir, deberá existir la tupla $(1, 2, 7)$.

9.1 Dependencias de junta embebidas

Dependencia de junta embebida Un esquema de relación $r(R)$ satisface la dependencia de junta embebida $DJE*(R_1, \ldots, R_p)$ si $\pi_S(r)$ satisface la dependencia de junta $*(R_1, \ldots, R_p)$, donde $S = R_1 R_2 \ldots R_p$.

Notar que r puede no satisfacer la DJE.

10 Quinta forma normal (5FN)

Una relación R está en 5FN con respecto a un conjunto de dependencias funcionales, multivaluadas y de junta D, si para todas las dependencias de junta en D^+ de la forma $*(R_1, \ldots, R_n)$ donde cada $R_i \subseteq R$ y $R = R_1 \cup \cdots \cup R_n$, al menos una de las siguientes condiciones se cumple:

- La dependencia de junta $*(R_1, \ldots, R_n)$ es trivial
- Cada R_i es una superclave de R

Propiedades:

- Toda relación que está en 5FN está en 4FN, y está en BCNF.
- Una descomposición 5FN no garantiza la preservación de dependencias.

Algoritmo 14 Descomposición 5FN

Entrada: una relación R, un conjunto de dependencias D^+
Salida: una descomposición $\rho = \{R_1, \ldots, R_n\}$ tal que R_i está en 5FN con respecto a D_i

1. $\rho = R$
2. Repetir hasta que no haya esquemas en ρZ que violen 5FN
   a) Elegir una dependencia de junta $*(DJ_1, \ldots DJ_k)$que viole 5FN sobre ρ_i
   b) Descomponer ρ_i en k relaciones:
      - $R_1\,(DJ_1)$
      - $R_k\,(DJ_k)$
   c) En ρ, reemplazar ρ por $R_1, \ldots, R_k$
3. Devolver ρ

Ejemplo: *sea* $D = \begin{cases} *(ABCD, CDE, BDI) \\ *(AB, BCD, AD) \\ A \to BCDE \\ BC \to AI \end{cases}$ *sobre el esquema de relación* $R = ABCDEI$.

Las clave candidata de R *son, por las dependencias funcionales,* $\{A, BC\}$.

R *no está en 5FN porque la dependencia de junta* $*(ABCD, CDE, BDI)$ *no es trivial, y* CDE, BDI *no son superclaves de* R.

La descomposición $R = \begin{cases} R_1 = & ABCD \\ R_2 = & CDE \\ R_3 = & BDI \end{cases}$ *sí está en 5FN porque:*

- *Para* R_1 *aplica la dependencia de junta* $*(AB, BCD, AD)$, *y todos los miembros son superclaves de* R, *por lo tanto está en 5FN.*
- *Para* R_2 *solo aplican dependencias de junta triviales.*
- *Para* R_3 *solo aplican dependencias de junta triviales.*

10.1 Implicación de dependencias

Utilizar el algoritmo Chase Tableau.

Example 3.36: Suppose we have a relation $R(A, B, C, D)$ with given dependencies $A \to B$ and $B \twoheadrightarrow C$. We wish to prove that $A \twoheadrightarrow C$ holds in R. Start with the two-row tableau that represents $A \twoheadrightarrow C$:

A	B	C	D
a	b_1	c	d_1
a	b	c_2	d

Notice that our target row is (a, b, c, d). Both rows of the tableau have the unsubscripted letter in the column for A. The first row has the unsubscripted letter in C, and the second row has unsubscripted letters in the remaining columns.

We first apply the FD $A \to B$ to infer that $b = b_1$. We must therefore replace the subscripted b_1 by the unsubscripted b. The tableau becomes:

A	B	C	D
a	b	c	d_1
a	b	c_2	d

Next, we apply the MVD $B \twoheadrightarrow C$, since the two rows now agree in the B column. We swap the C columns to get two more rows which we add to the tableau, which becomes:

A	B	C	D
a	b	c	d_1
a	b	c_2	d
a	b	c_2	d_1
a	b	c	d

We have now a row with all unsubscripted symbols, which proves that $A \twoheadrightarrow C$ holds in relation R. Notice how the tableau manipulations really give a proof that $A \twoheadrightarrow C$ holds. This proof is: "Given two tuples of R that agree in A, they must also agree in B because $A \rightarrow B$. Since they agree in B, we can swap their C components by $B \twoheadrightarrow C$, and the resulting tuples will be in R. Thus, if two tuples of R agree in A, the tuples that result when we swap their C's are also in R; i.e., $A \twoheadrightarrow C$." $\square$

Parte III

SQL

Componentes:

1. DDL (*Data Definition Language*)
2. DML (*Data Manipulation Language*)
3. CL (*Control Language*)

Para identificar unívocamente a una relación, se utiliza el formato `<catalogo>.<esquema>.<relación>`

11 DDL

DDL Lenguaje de definición de datos para especificar el esquema de la base de datos, eliminar relaciones, modificar esquemas, crear vistas, crear restricciones de integridad, especificar derechos de acceso, etc.

11.1 Tipos de dominios

- `char(`n`)`: texto de longitud n
- `varchar(`n`)`: texto de longitud variable, con un máximo de n caracteres
- `nvarchar(`n`)`: texto de longitud variable, con un máximo de n caracteres, utilizando el formato Unicode
- `int`
- `smallint`
- `numeric(`p`,`d`)`: p dígitos con signo, y d de los p dígitos están a la derecha del punto decimal.
- `real, double precision`
- `float(`n`)`
- `date`
- `time`
- `timestamp`
- `clob(n)`: character large object de n bytes
- `blob(n)`: binary large object de n bytes

11.2 Create table

```
CREATE TABLE r(A1 D1, A2 D2, ..., An Dn,
               <restriccion-integridad1>,
               ...,
               <restriccion-integridadK>);
```

La instrucción anterior crea una tabla llamada *r* con atributos A_i, y D_i es el dominio del atributo A_i.

Las restricciones de integridad pueden ser:

- `primary key` $(A_{j1}, A_{j2}, \ldots, A_{jm})$: los atributos A_{ji}, con $i \in [1, m]$ forman la clave primaria de cada tupla. No pueden ser *null*, y deben ser únicos.
- `check` (P): todas las tuplas de r deben satisfacer el predicado P.
- `foreign key`:

```
FOREIGN KEY (atr) REFERENCES rel
        ON [update|delete] [cascade|set null|set default];
```

Se establece que el atributo `atr` es una clave primaria en la relación `rel`. Se puede especificar lo que sucede si se produce una actualización o un borrado de dicho valor en la relación `rel`:

- **Cascade**: se actualiza/elimina la tupla en esta relación
- **Set null**: la clave foránea se establece en *null*
- **Set default**: la clave foránea se establece en su valor por default

- `assert`: define una restricción aplicable a varias tablas.

```
CREATE ASSERTION nombre_asercion
        CHECK P
```

11.3 Drop table

La instrucción `DROP TABLE r` elimina todas las tuplas de la tabla r y el esquema de relación r.

11.4 Alter table

La instrucción puede utilizarse para agregar atributos nuevos (con valor *null*) o para eliminarlos.

```
ALTER TABLE r
        ADD <Atributo> <Dominio>;
ALTER TABLE r
        DROP <Atributo>;
```

Diccionario de datos Contiene **metadatos** (información sobre los datos), en particular, sobre el esquema de la misma.

12 DML

DML Lenguaje de manipulación de datos para expresar las consultas a la base de datos.

Operaciones CRUD (*Create*, *Read*, *Update*, *Delete*)

Álgebra relacional	***SQL***
$\pi_{attr}(r)$	`SELECT attr FROM r`
$\sigma_{cond}(r)$	`SELECT * FROM r WHERE cond`
$r \times s$	`SELECT * FROM r,s`

La expresión

```
SELECT a1, a2, ..., an
FROM r1, r2, ..., rm
WHERE p;
```

es equivalente a la expresión $\pi_{a_1,a_2,\ldots,a_n}(\sigma_p(r_1 \times r_2 \times \cdots \times r_m))$.

- El resultado de una expresión SQL **puede** contener tuplas duplicadas. Para eliminar los duplicados, se utiliza la cláusula `SELECT DISTINCT`.
- Las funciones agregadas no se pueden componer. Por ejemplo, `max(count(*))` no es válido.
- Una cláusula de tipo `SELECT *` indica que se seleccionan todos los atributos de las relaciones en la cláusula `FROM`.
- El operador de renombre as: `SELECT viejo-nombre AS nuevo-nombre`
- El operador `like` se puede usar dentro de una cláusula `where` para buscar valores que cumplan un patrón, especificado con los siguientes caracteres especiales:

- %: matchea cualquier sub string
- _: matchea un caracter cualquiera

- El operador `order by <atributo(s)> [asc|desc]` ordena el resultado de una consulta por uno o varios de los atributos, de forma ascendente o descendente (ascendente por default).
- Operadores sobre conjuntos:
  - `Union`
  - `Intersect`
  - `Except`
- El operador `group by` permite agrupar tuplas en base a uno o más atributos (las tuplas con el mismo valor para los atributos en esta cláusula se ponen en un mismo grupo). Los atributos en la cláusula `select`, por fuera de las funciones agregadas, deben estar en el operador `group by`. La sintaxis es:

```
SELECT expr1, ..., exprN, funcionAgregada(expr)
FROM tablas
WHERE condiciones
GROUP BY expr1, ..., exprN;
```

  Ejemplo: encontrar el promedio del saldo de las cuentas en cada sucursal de un banco.

```
SELECT sucursal-nombre, avg(saldo)
FROM cuentas
GROUP BY sucursal-nombre;
```

- El operador `having` permite seleccionar a grupos formados con `group by` que cumplan una condición. Cualquier atributo que esté presente en la cláusula `having` sin agregar, debe aparecer en el operador `group by`.

```
SELECT columna, funcion_agregada(columna)
FROM tabla
WHERE columna operator value
GROUP BY columna
HAVING funcion_agregada(columna) operator value;
```

  Ejemplo: encontrar los nombres de las sucursales del banco que tengan un promedio general de saldo mayor a $1200.

```
SELECT sucursal-nombre
FROM cuentas
GROUP BY sucursal-nombre
HAVING avg(saldo) > 1200
```

Nota: si se utilizan el operador `where` y el operador `having`, primero se aplica el `where` y luego se filtra por `having`.

12.1 Consultas anidadas

En una consulta anidada, el `select` interno puede tener variables de tuplas definidas dentro del `select`, o en cualquier consulta que incluya a ésta.

- El conector `in` permite verificar la pertenencia de un valor a un conjunto que es resultado de una cláusula `select`. De forma equivalente, existe el conector `not in`.

  Ejemplo: encontrar los nombres de los clientes que tienen un préstamo y una cuenta en el banco.

```
SELECT DISTINCT cliente-nombre
FROM pidio-prestamo
WHERE cliente-nombre IN (SELECT cliente-nombre
                         FROM tiene-cuenta);
```

- El operador de comparación `some` en una cláusula `where` de un `select` externo devuelve verdadero si el valor del atributo en cuestión es mayor o menor que al menos un valor de la cláusula `select` interna.

  Ejemplo: encontrar los nombres de las sucursales del banco que tienen activos mayores que los de al menos una sucursal en Brooklyn.

```
SELECT sucursal-nombre
FROM sucursales
WHERE activos > SOME (SELECT activos
                      FROM sucursales
                      WHERE sucursal-nombre = 'Brooklyn');
```

- El operador de comparación `all` en una cláusula `where` de un `select` externo devuelve verdadero si el valor del atributo en cuestión es mayor a todos los valores de la cláusula `select` interna.

  Ejemplo: encontrar los nombres de las sucursales del banco que tienen activos mayores que los todas las sucursales en Brooklyn.

```
SELECT sucursal-nombre
FROM sucursales
WHERE activos > ALL (SELECT activos
                      FROM sucursales
                      WHERE sucursal-nombre = 'Brooklyn');
```

- El operador any devuelve verdadero si existe al menos un valor para

  Ejemplo: encontrar los clientes que pidieron un préstamo de monto mayor a al menos los activos de una sucursal.

```
SELECT cliente-nombre
FROM pidio-prestamo
WHERE cantidad > ANY (SELECT activos
                      FROM sucursales);
```

- El operador exists devuelve verdadero si la consulta interna no es vacía.

  Ejemplo: encontrar los clientes que tienen una cuenta y un préstamo en el banco.

```
SELECT cliente-nombre
FROM pidio-prestamo
WHERE EXISTS (SELECT *
                FROM tiene-cuenta
            WHERE pidio-prestamo.cliente-nombre = tiene-cuenta.cliente-nombre);
```

- El operador unique devuelve verdadero si la consulta interna no contiene tuplas duplicadas.

SQL no ofrece en forma nativa la posibilidad de ejecutar el operador división. Sin embargo, este se puede implementar usando consultas anidadas.

Ejemplo: dados los siguientes esquemas relacionales

```
Estudiante (Enro, Enombre, Carrera, Anio_cursa, Edad)
Curso (Cnombre, Horario, Aula, Pid)
Cursa (Enro, Cnombre)
Profesor (Pid, Pnombre, departamento)
```

Hallar los nombres de todos los profesores que enseñan en todas las aulas en las que se dicta algún curso.

```
SELECT P.Pnombre
FROM Profesor P
WHERE NOT EXISTS (SELECT C.Aula
                  FROM Curso C1
                  WHERE NOT EXISTS (SELECT *
                                    FROM Curso C2
                                    WHERE C2.Aula = c1.Aula AND
                                        C2.Pid = P.Pid));
```

12.2 Delete

Sintaxis para el borrado de tuplas de la relación r para las cuales P es verdadera:

```
DELETE FROM r
WHERE P
```

- Sólo se borra de una relación.
- Sólo se borran tuplas (no atributos de tuplas).
- Primero se buscan todas las tuplas que satisfacen P, y luego se borran todas ellas.

12.3 Insert

Sintaxis para la inserción de tuplas en la relación r:

```
INSERT INTO r (nombreAtributo1, ..., nombreAtributoN)
VALUES (valor1, ..., valorN)

INSERT INTO r
        SELECT (...)
```

- Primero se buscan todas las tuplas que satisfacen el `select`, y luego se insertan todas ellas.

12.4 Update

Sintaxis para la actualización de un atributo de una tupla de r:

```
UPDATE r
SET atributo = valor
WHERE condicion

UPDATE r
SET atributo = CASE
                                WHEN condicion1 THEN valor1
                                WHEN condicion2 THEN valor2
                                ELSE valor3
                           END
```

Parte IV

Procesamiento de Consultas

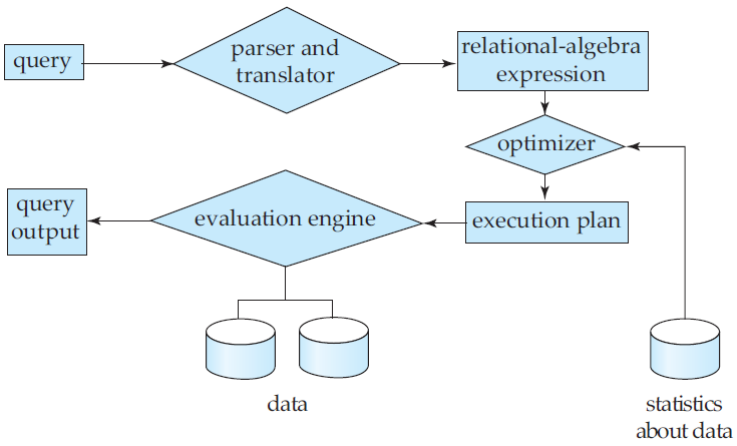


Figura 21: Pasos en el procesamiento de consultas

Asumimos que toda la relación se almacena de forma contigua en disco.
Metadatos que se almacenan para cada base de datos:

1. n_r = cantidad de tuplas en la relación r
2. l_r = tamaño de una tupla, en bytes
3. b_r = cantidad de bloques que ocupa la relación r.

$$b_r = \frac{l_r \times n_r}{tam\, bloque\, en\, bytes}$$

4. f_r = factor de bloqueo de la relación r (es decir, cuantas tuplas caben en un bloque)

$$f_r = \left\lceil \frac{n_r}{b_r} \right\rceil$$

5. $V(A,r)$ =cantidad de valores distintos que toma el atributo A en la relación r
6. $MIN(A,r)$ =mínimo valor que toma el atributo A en la relación r
7. $MAX(A,r)$ =máximo valor que toma el atributo A en la relación r

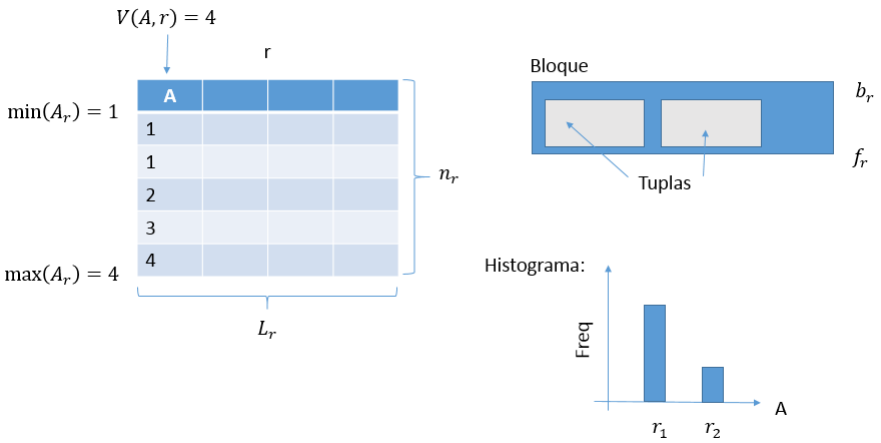


Figura 22: Estadísticas que se almacenan de la base de datos

13 Índices

Ciertas consultas sobre algunos atributos son más frecuentes que otras. Para acelerar estas consultas, se utilizan los índices. Podemos tener más de un índice sobre una relación.

Índice Estructura de datos (usualmente árbol B+) que permiten encontrar rápidamente las tuplas de una relación que tengan un valor especifico para un atributo o atributos (el/los que el índice almacena), con la desventaja de que cada modificación a la relación requiere actualizar el índice.

Un registro de un índice tiene un valor de la clave de búsqueda, y punteros a uno o más tuplas con esa clave de búsqueda.

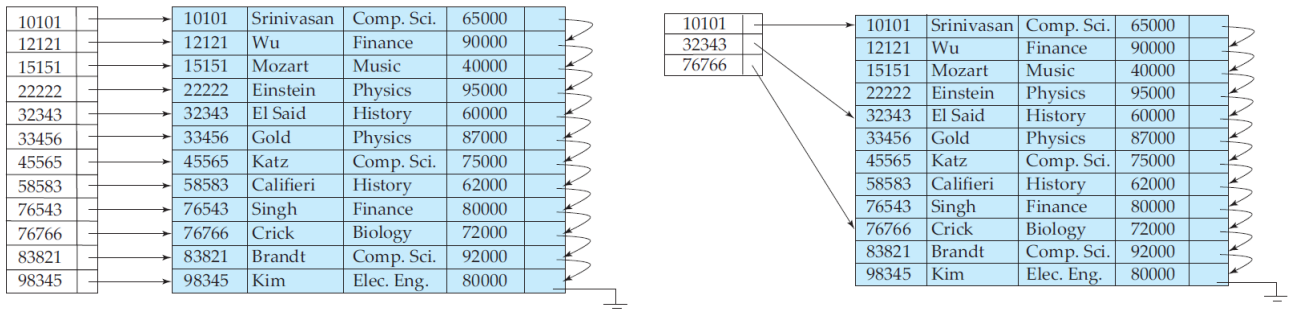
```
CREATE [CLUSTERED] INDEX estudianteID_indice
        ON estudiante
{
        ID asc
};
```

Tipos de índices:

- Según la clave:
  - **Clusterizado / primario**: índice cuya clave de búsqueda define el orden secuencial del archivo de la tabla. Sólo puede haber uno de estos índices para cada relación
  - **Clusterizado / secundario**.
    - El orden físico de las tuplas no es igual al orden del índice
    - Las columnas que se indexan son, típicamente, atributos no claves
    - Siempre son densos
- Según la cantidad de registros que tiene:
  - **Denso**: tiene un registro para cada valor posible de la clave de búsqueda. Si es clusterizado, tiene un puntero al primer registro con ese valor. Si es no clusterizado, tiene punteros a todos los registros con ese valor.
  - **Esparcido**: tiene registros para algunos valores de la clave de búsqueda. Solo se pueden usar si el archivo de datos está ordenado por la clave de búsqueda.

$$Esparcido \implies Primario$$

$$Secundario \implies Denso$$



(a) Índice denso (b) Índice esparcido

Figura 23: Índices densos y esparcidos

Índice compuesto índice cuya clave de búsqueda está formada por más de un atributo. Conviene que los atributos de más a la izquierda sean más discriminantes que los de la derecha. *Ejemplo: para una relación* $película(nombre, año)$*, es esperable que haya más consultas sobre el* $nombre$ *que sobre el* $año$*. Entonces, un índice* $(nombre, año)$ *sería mejor que* $(año, nombre)$.

Índice cubridor Es un índice que contiene todas las columnas de una consulta, y por ende no se necesita realizar búsquedas adicionales en el índice clusterizado.

13.1 Estructuras de índices

- Árbol B+
- Tabla de hash

	Ventajas	Desventajas
Árbol B+	Soporta búsquedas por rango	Operaciones más costosas
Tabla de hash	La búsqueda típica requiere solo una operación de I/O	No soporta búsquedas por rango.

14 Formas de realizar consultas

1. Búsqueda lineal sobre toda la relación
2. Usando índices
3. *Hashing*
4. Ordenamiento

Los algoritmos pueden asumir que una relación entra completamente en memoria, o que son muy grandes.

14.1 Operadores

- *Scan*
  - *Table scan*: se leen todos los bloques de la relación en disco.
  - *Index scan*: se leen todos los bloques de la relación en disco, utilizando un índice.
  - *Sort scan:* ordena una relación R sobre un atributo a.
    - Si hay un índice sobre a, se lo usa.
    - Si R entra en memoria, se la trae a memoria con *table* o *index scan*, y se la ordena en memoria
    - Si R no entra en memoria, se la ordena externamente.

15 Algoritmos

15.1 Selección

$\sigma_{A=a}(r)$

- Búsqueda lineal sobre r:
  - Si la relación es clusterizada: b_r
  - Si la relación no está clusterizada: n_r
- Búsqueda con índice sobre los atributos de A:
  - Si el índice es clusterizado: costo $\frac{b_r}{V(A,r)}$
  - Si el índice no es clusterizado: costo $\frac{n_r}{V(A,r)}$

15.2 Junta

$R(X,Y) \bowtie S(Y,Z)$, donde R es la relación más pequeña. Suponemos que la memoria está formada por M buffers.

- Iteración ingenua (*tuple based nested-loop join*):
  - costo $n_s \times n_r$

```
para cada tupla s en S:
        para cada tupla r en R:
                si (s join r = t):
                        devolver t
```

- Iteración por bloques (*block based nested-loop join*):
  - Costo $b_r + \frac{b_r}{m-1} \times b_s$ (R se lee una vez, S se lee por cada pedazo de R, cada pedazo es de tamaño $\frac{b_r}{m-1}$)
  - Conviene que la relación del ciclo externo sea la más pequeña

```
desde i=1 hasta i=(bR / m-1):
        leer R en memoria (ocupar M-1 buffers)
        para cada bloque bs en S:
                leer bs en memoria (ocupar 1 buffer)
                para cada tupla t en bs:
                        buscar las tuplas de R en que hacen join con t
                        devolver t
```

- *Indexed nested-loop join*: suponer que se tiene un índice sobre S del atributo Y.
  - Si el índice es clusterizado: costo $b_r + \frac{n_r b_s}{V(Y,S)}$
  - Si el índice no es clusterizado: costo $b_r + \frac{n_r n_s}{V(Y,S)}$
  - Este método es eficiente cuando S no está almacenada en forma contigua en disco.

```
para cada bloque br de R:
        para cada tupla t de br:
                buscar con el indice las tuplas de S que hacen join con t
                para cada tupla s join t:
                        agregar <s,t> al resultado
```

- Sort-merge (*sort based join*):
  - Costo $b_s + b_r + b_s \log(b_s) + b_r \log(b_r)$

```
ordenar R con respecto a Y
ordenar S con respecto a Y
mergear R y S:
        encontrar min(R) y min(S)
        si min(R) > min(S)
                quitar las tuplas de S con Y como atributo
                leer S
        si min(R) < min(S)
                quitar las tuplas de R con Y como atributo
                leer R
        si min(R) == min(S)
                agregar r join s al resultado
                leer R
                leer S
```

- Método Simple de Junta Hash: utiliza una función de hash h. Requiere que la relación R entre en memoria. $h \to [0, M-1]$.
  - Costo: $2\,(b_r + b_s)$

```
TH = {} // en memoria

// fase constructiva
para cada bloque de R:
        para cada tupla r del bloque:
                calcular h(r[Y])
                guardar TH[Y] = r

// fase exploratoria
para cada bloque de S:
        para cada tupla s del bloque:
                calcular h(s[Y])
                agregar al resultado las tuplas de TH[Y] join "s"
```

- Método GRACE:

  El método de junta Hash versión GRACE se usa para calcular la junta $R \bowtie_{R.A=S.A} S$ cuando ninguna de las tablas entran en memoria. Lo que se hace es generar 2 conjuntos de M particiones. Se utilizan dos funciones de hash: una para generar las particiones $h : A \to [0, M-1]$, y otra para calcular qué tuplas hacen *join*.

  En una primera etapa, por cada tupla t de cada bloque de R se aplica la función de hashing $h(t)$ para saber a cuál partición enviar la tupla. Se envía cada tupla a esa partición (un archivo) en disco. Al finalizar la etapa, se tienen M archivos.

  En una segunda etapa, se hace lo mismo con S, pero con otros M archivos. La clave está en que si dos tuplas hacen *join*, entonces necesariamente deben estar en el mismo número de partición.

  Finalmente, se leen las i particiones. A cada tupla leída de la partición i de R se le aplica la función de hash h_2 para saber dónde guardarla en una tabla de hash en memoria. Luego, se calcula h_2 para cada tupla de la partición i de S. Este valor nos dirá la posición en la tabla de hash donde están las tuplas de R que hacen *join* con dicha tupla. De esta forma se obtienen las tuplas que hacen *join*, y se las graba en disco.

  - Costo: $3\,(b_s + b_r)$

```
// particionamiento de R. Costo: 2*bR
para cada tupla r de R:
        num buffer = calcular h1(r[Y])
        guardar "r" en el buffer
        si (buffer completo)
                guardarlo en disco Ri
// particionamiento de S. Costo: 2*bS
para cada tupla s de S:
        num buffer = calcular h1(s[Y])
        guardar "s" en el buffer
        si (buffer completo)
                guardarlo en disco Si

// metodo simple de hash. Costo: bR + bS
para i desde 0 a M-1:
        leer Ri
        para cada tupla r en Ri:
                calcular h2(r[Y])
                guardar TH[r[Y]] = r
        leer Si
        para cada tupla s en Si:
                calcular h2(s[Y])
                agregar al resultado las tuplas de TH[r[Y]] join "s"
```

16 Evaluación de expresiones

La forma obvia de evaluar una expresión que tiene varias operaciones es evaluar cada operación en el orden indicado, materializando en disco cada resultado intermedio como relaciones temporales.

Otra alternativa, más eficiente, es evaluar las operaciones en simultáneo en un **pipeline**, donde los resultados de una operación pasan a la próxima, y se elimina la necesidad de almacenar relaciones temporales. Esto se puede utilizar para evaluar juntas múltiples: $R_1 \bowtie R_2 \bowtie \cdots \bowtie R_k$.

Ventajas de *pipeline:*

1. Elimina el costo de leer y escribir relaciones temporales.
2. Puede empezar a generar resultados inmediatamente.

Uso de *pipelining*:

- *Sort*: no se puede aplicar. El resultado del sort no se puede mostrar hasta que no se hayan procesado todas las tuplas.
- *Join*: el método GRACE no se puede aplicar porque requiere leer y particionar todas las tuplas antes de que pueda producirse un resultado. Sin embargo, el método *indexed nested-loop* puede aprovechar del *pipelining*. Si las relaciones están ordenadas por los atributos de join, merge join también puede aprovechar del *pipelining*.

17 Optimización de consultas

Plan de evaluación conjunto de operaciones de álgebra relacional a ejecutar, junto con las instrucciones para llevarlas a cabo (por ejemplo, los índices a usar).

Optimización de consultas proceso de seleccionar el mejor plan de evaluación. El costo de cada plan se evalúa teniendo en cuenta información estadística de las relaciones. Lo que se busca minimizar es la cantidad de transferencias de bloques del disco a la memoria y la cantidad de *seeks* que se hacen en el disco.

Para ello se tiene en cuenta:

1. ¿Cuál de la de las expresiones equivalentes a la consulta es más eficiente para resolver la misma?
2. ¿Qué algoritmo se utilizará para implementar la operación?
3. ¿Cómo deberían las operaciones pasarse datos entre sí? (*Pipeline*, buffers de memoria, disco)

```
EXPLAIN consulta
```

17.1 Reglas de equivalencia

Reglas relacionadas con la selección:

1. $\sigma_{a \wedge b}(r) = \sigma_a(\sigma_b(r)) = \sigma_b(\sigma_a(r))$
2. $\sigma_{a \vee b}(r) = \sigma_a(r) \cup \sigma_b(r)$
3. $\sigma_a(r \cup s) = \sigma_a(r) \cup \sigma_a(s)$
4. $\sigma_a(r - s) = \sigma_a(r) - s = \sigma_a(r) - \sigma_a(s)$
5. $\sigma_a(r \bowtie s) = \sigma_a(r) \bowtie \sigma_a(s)$
6. Si a solo tiene atributos de S: $\sigma_a(r \times s) = r \times \sigma_a(s)$

Reglas relacionadas con la proyección:

1. $\pi_L(R \cup S) = \pi_L(R) \cup \pi_L(S)$
2. $\pi_L(\sigma_a(R)) = \pi_L(\sigma_a((\pi_M(R))))$ donde M es la lista de atributos que figuran en L o en a

Reglas relacionadas con el *join*:

1. $r_1 \bowtie r_2 = r_2 \bowtie r_1$
2. $(r_1 \bowtie r_2) \bowtie r_3 = r_1 \bowtie (r_2 \bowtie r_3)$

17.2 Heurísticas de optimización de consultas

1. Ejecutar las selecciones (σ) lo antes posible
2. Ejecutar las proyecciones (π) lo antes posible
3. El orden de las operaciones de *join* ($\bowtie$) es importante
4. Evitar cuando sea posible los productos cartesianos ($\times$)

18 Estimación de tamaño de consultas

A continuación se describe la estimación de cantidad de tuplas para cada tipo de consulta.

Se utiliza el concepto de **selectividad** de una condición A como la "probabilidad de que una tupla satisfaga una condición A". Sea $s_i = V(\sigma_i, r)$ la cantidad de tuplas que satisfacen la condición σ_i. La probabilidad de satisfacer σ_i es $\frac{s_i}{n_r}$.

- Selección
  - $\sigma_{A=a}(r)$
    - Si A distribuye uniformemente: $\frac{n_r}{V(A,r)}$
    - Si se dispone de un histograma para A, y $a \in rango$: $\frac{freq_{rango}(A,r)}{cant_{rangos}}$
  - $\sigma_{A \leq v}(r)$ con v conocido
    - Si A distribuye uniformemente y $v < \text{mín}(A, r)$: 0
    - Si A distribuye uniformemente y $v \geq \text{máx}(A, r)$: n_r
    - En cualquier otro caso: $n_r \cdot \frac{v - min(A,r)}{max(A,r) - min(A,r)}$
  - $\sigma_{a \wedge b \wedge \cdots \wedge z}(r)$ donde hay x selectores: $n_r \times P(a) \times \cdots \times P(z) = n_r \cdot \frac{s_a \cdot s_b \cdots s_z}{(n_r)^x}$
  - $\sigma_{a \vee b \vee \cdots \vee z}(r)$ donde hay x selectores independientes entre sí:

$$n_r \times [P(a) + \cdots + P(z)] = n_r \cdot \left[1 - \left(1 - \frac{s_a}{n_r}\right) \times \cdots \times \left(1 - \frac{s_z}{n_r}\right)\right]$$

  - $\sigma_{\neg a}(r)$: $n_r - V(a, r)$
- Junta natural[4]
  - Si $R \cap S = \emptyset$: $n_r \times n_s$
  - Si $R \cap S = PK_R$: $\leq n_S$ (porque podría haber *nulls*)
  - Si $R \cap S = PK_S$: $\leq n_R$ (porque podría haber *nulls*)
  - Si $R \cap S = FK_S$: n_S
  - Si $R \cap S = FK_R$: n_R
  - Si $\#(R \cap S) = 1$: $\frac{n_s \times n_r}{\text{máx}(V(S,R \cap S), V(R,R \cap S))} \approx \frac{n_s \times n_r}{V(S,R \cap S)} \approx \frac{n_s \times n_r}{V(R,R \cap S)}$
  - Si $\#(R \cap S) = 2$ y el *join* es de tipo $R.A_1 = S.A_2 \wedge R.B_1 = S.B_2$:

$$\frac{n_r \times n_s}{\text{máx}\left(V(R, A_1), V(S, A_2) \times \text{máx}\left(V(R, B_1), V(S, B_2)\right)\right.}$$

- Proyección: $\pi_A(r) : V(A, r)$
- Agregación ${}_A\varrho_f(r) : V(A, r)$
- Unión $R \cup S$: $n_r + n_s$
- Intersección $R \cap S$: $\text{mín}\left(n_r, n_s\right)$ (cota superior)
- Diferencia $R - S$: r (cota superior)
- $V\left(A, \sigma_{A \, op \, v}(r)\right) : V(A, r) \times s_{A \, op \, v}$
- $V\left(A, r \bowtie s\right)$:
  - Si $A \in R$: $\text{mín}\left(V(A, r), n_{r \bowtie s}\right)$
  - Si $a_1 \in A$ y $a_2 \in A$, $\text{mín}\left(V(a_1, r) \times V(a_2 - a_1, s); V(a_1 - a_2, r) \times V(a_2, S); n_{r \bowtie s}\right)$

[4] Se introducen dos simplificaciones:

1. Si $R(X, Y)$ y $S(Y, Z)$, y $V(R, Y) \leq V(S, Y)$ entonces cada valor de Y en R será un valor de Y en S.
2. Si A es un atributo de R pero no de S, $V\left(R \bowtie S, A\right) = v\left(R, A\right)$

18.1 Cálculo de juntas con histogramas

Un sistema de bases de datos puede computar un histograma de valores para un atributo dado. Si $V(R, A)$ no es muy grande, el histograma puede consistir de la cantidad de tuplas que tienen cada posible valor del atributo. Si $V(R, A)$ es muy grande, entonces podría guardarse solamente los valores más frecuentes, o agruparlos en rangos.

Los tipos de histogramas más frecuentes son:

1. **Igual ancho:** se escoge un ancho w y una constante v_0. Se almacena la cantidad de tuplas con valores v en los rangos $v_0 \leq v < v_0 + w$, $v_0 + w \leq v < v_0 + 2w$, etcétera.
2. **Valores más frecuentes**: se listan los valores más frecuentes y la cantidad de tuplas que tienen esos valores. También se puede proporcionar la cantidad de tuplas que tienen "otros" valores.

Ejemplo: sea la junta $R(A, B) \bowtie S(B, C)$. Sabemos que $V(R, B) = 14$ y que $V(S, B) = 13$. Tenemos los histogramas de valores más frecuentes para $R \cap S = B$.

	0	*1*	*2*	*5*	*"Otros"*
$R.B$	*150*	*200*	*?*	*100*	*550 (11 valores)*
$S.B$	*100*	*80*	*70*	*?*	*250 (10 valores)*

Suponemos que cada valor que aparece en la relación con menos valores de B (en este caso, S) también aparecen en la otra relación (en este caso, R). También suponemos que la distribución de los valores dentro de "Otros" es uniforme, y estimamos la frecuencia de los datos desconocidos:

	0	*1*	*2*	*5*	*Otros*
$R.B$	*150*	*200*	$\frac{550}{11} = 50$	*100*	*550 (10 valores)*
$S.B$	*100*	*80*	*70*	$\frac{250}{10} = 25$	*250 (9 valores)*

Entonces, el tamaño de la junta es:

$$\begin{aligned} T[R \bowtie S] &= T[R \bowtie_{B=0} S] + T[R \bowtie_{B=1} S] + T[R \bowtie_{B=2} S]\, T[R \bowtie_{B=5} S] + T[R \bowtie_{B \neq 0,1,2,5} S] \\ &= (150 \times 100) + (200 \times 80) + (50 \times 70) + (100 \times 25) + \text{mín}(9, 10) \times \left(\frac{550}{11} \times \frac{250}{10}\right) \\ &= 15{,}000 + 16{,}000 + 3{,}500 + 2{,}500 + 9 \times 1250 \\ &= 48{,}250 \end{aligned}$$

Ejemplo: sean las relaciones $Enero(dia, temp)$ y $Julio(dia, temp)$. Sea la consulta

```
SELECT Enero.dia, Julio.dia
FROM Enero, Julio
WHERE Enero.temp = Julio.temp
```

Suponer los siguientes histogramas de igual ancho:

Rango C°	*Enero*	*Julio*
-17, -13	*0*	*40*
-13, -9	*0*	*60*
-8, -4	*0*	*80*
-3, 1	*0*	*50*
2, 6	*5*	*10*
7, 11	*20*	*5*
12, 16	*50*	*0*
17, 21	*100*	*0*
22, 26	*60*	*0*
27, 31	*10*	*0*

Sabemos que si dos bandas tienen T_1 y T_2 tuplas respectivamente, y la cantidad de valores de la banda es V, entonces la estimación del tamaño de la junta es $\frac{T_1 T_2}{V}$.

En el ejemplo anterior, las únicas bandas que contribuyen al resultado son las de [2,6] y [7,11]. Entonces, el tamaño de la junta es:

$$\begin{aligned} T\left[R \bowtie S\right] &= T\left[R \bowtie_{temp \in [2,6]} S\right] + T\left[R \bowtie_{temp \in [7,11]} S\right] \\ &= \frac{5 \times 10}{4} + \frac{20 \times 5}{4} \\ &= 12,5 + 25 \\ &= 37,5 \end{aligned}$$

Parte V

Control de Concurrencia

Ítem de dato elemento al que accede una transacción. Puede ser un registro de una base de datos, un bloque de disco, un campo de un registro, o incluso toda la base de datos. Cada ítem tiene un nombre único que lo identifica (por ejemplo, la dirección física de un bloque de disco).

Modelo de concurrencia intercalada la CPU ejecuta una transacción a la vez, pero varias transacciones en un período de tiempo.

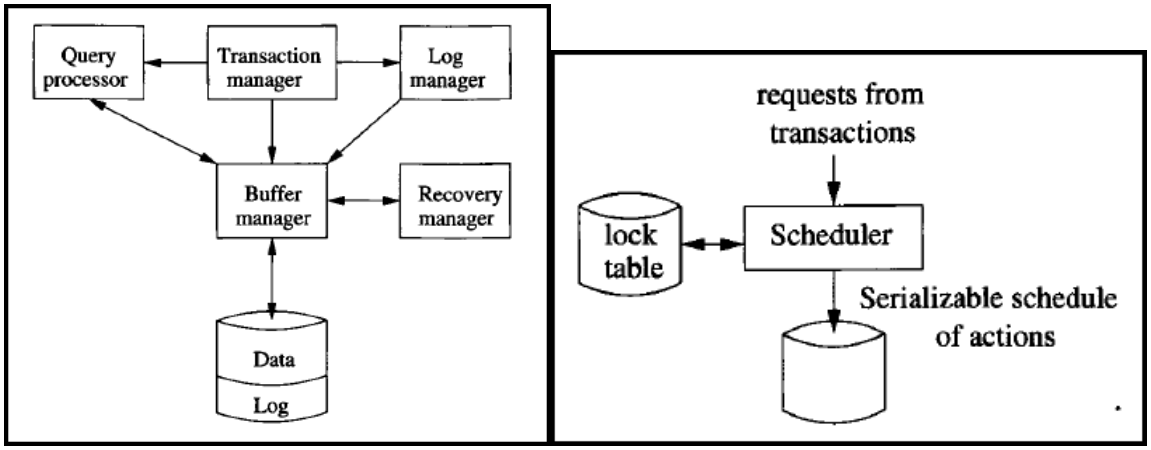


Figura 24: El manejador de transacciones (a) le emite órdenes al manejador del log, (b) se asegura que transacciones concurrentes no interfieran entre ellas. El scheduler permite o bloquea transacciones

19 Transacciones

Transacción unidad lógica de procesamiento. Está formada por una o más operaciones que acceden a la base de datos. Tiene un identificador único de transacción.

Debe tener las siguientes propiedades **ACID**:

Atomicity	**La transacción debe ejecutarse en su totalidad o no debe ejecutarse**	**Responsabilidad del gestor de recuperación, que mantiene un log donde guarda los valores viejos sobrescritos por una transacción**
Consistency	**La transacción debe llevar a la base de datos de un estado consistente a otro (i.e. que respeten las reglas de integridad de la base de datos)**	**Responsabilidad de los programadores**
Isolation	**La ejecución de una transacción no debe interferir con otras**	**Responsabilidad del gestor de concurrencia**
Durability	**Los efectos de una transacción commiteada deben persistir en la base de datos. Es decir, luego de commiteada, no deberíamos necesitar ejecutar un *rollback***	**Gestor de recuperación, que debe garantizar que el log esté en disco antes de que termine la transacción**

How transactions interact with the database. There are three address spaces that interact in important ways:

1. *The space of disk blocks holding the database elements.*
2. *The virtual or main memory address space that is managed by the buffer manager.*
3. *The local address space of the transaction.*

Cada transacción está formada por una o más operaciones:

- `start`

- `leer(X)`: lee un item de dato de la base de datos a una variable local a la transacción, que está en memoria
- `escribir(X)`: escribe una variable de programa en memoria, en la base de datos
- `commit`: marca el fin exitoso de una transacción. Los cambios que introdujo son seguros para guardar en la base de datos.
- `abort`: marca el fin con errores de una transacción. Los cambios que introdujo se deben revertir.

Estados posibles de una transacción:

- ***Partially committed***: luego de que se ejecutó la última instrucción pero antes de ejecutar el *commit* o *abort*
- ***Commited***: sus efectos fueron almacenados permanentemente en la base de datos
- ***Aborted***: se ejecutó un *roll back* de la transacción y la base de datos se restauró a su estado original

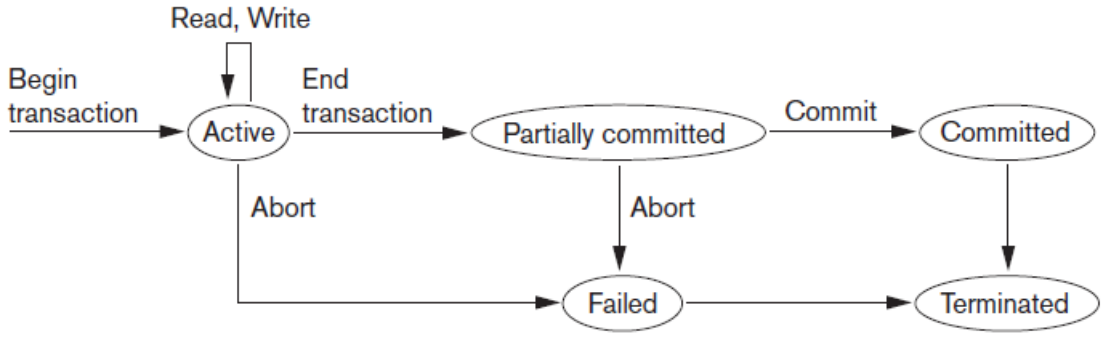


Figura 25: Diagrama de estados de una transacción

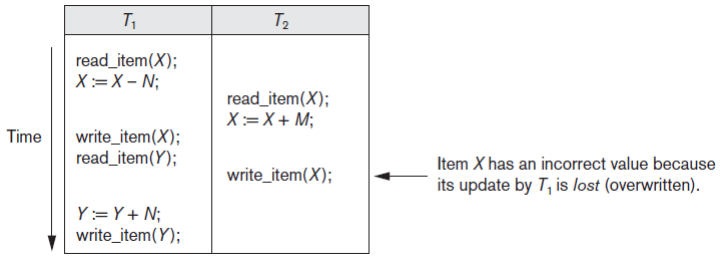
Una transacción llega al **punto de commit** cuando todas sus operaciones se ejecutaron correctamente y se grabaron todos sus registros de operaciones en el log. Luego de este punto, la transacción está **commiteada** y se debe almacenar permanentemente en la base de datos. Esto se marca agregando un registro `[commit,T]` en el log.

1. Si ocurre una falla y la transacción aún no grabó `[commit,T]`, se debe ejecutar un *rollback* de esta transacción.
2. Si ocurre una falla y la transacción ya había grabado `[commit,T]` en el log, se debe *rehacer* esta transacción.

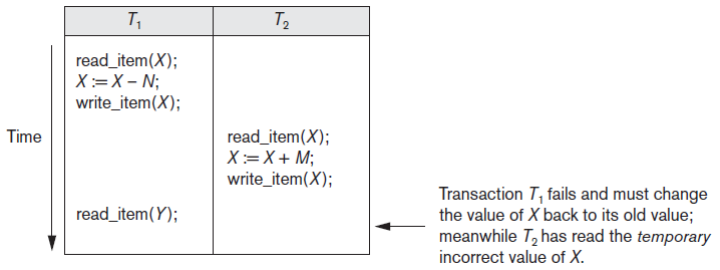
El protocolo **WAL (*Write-Ahead Logging*)** indica que antes de commitear una transacción se debe guardar el log en memoria al log en disco.

20 Problemas de Concurrencia

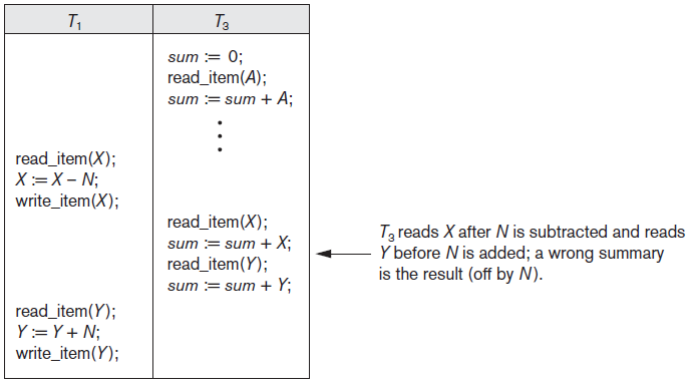
1. ***The Lost Update Problem***: una `Escritura` que sobrescribe a otra.
2. ***The Dirty Read Problem***: una `Lectura` de un valor incorrecto.
3. ***The Incorrect Summary Problem***: una `Lectura` de muchos valores incorrectos.
4. ***The Unrepeatable Read Problem***: dos `Lecturas` consecutivas que producen resultados distintos, por haber una transacción intermedia que cambió el valor.



(a) Lost Update



(b) Dirty Read



(c) Incorrect Summary

Figura 26: Probemas de concurrencia

20.1 Atributos de una Transacción

- Modo de acceso: `READ ONLY`, `READ WRITE`
- Nivel de *isolation*:

Nivel de isolation	*Tipo de violación que se puede producir para dos transacciones T y S*			
	Escritura sucia (T escribe el valor X después de que una transacción S lo escribiera, y S no comitió ni abortó)	**Lectura Sucia** (T lee el valor$X$ después de una transacción S que no comitió ni abortó)	**Lectura No Repetible** (T lee el valor X, luego S lo actualiza, luego T lee X y es un nuevo valor)	***Phantoms*** (T lee varios datos que cumplen una condición, luego S agrega un dato que también lo verifica. Si T lee de nuevo, encontrará un nuevo dato que antes no estaba)
READ UNCOM-MITTED	No	Si	Si	Si
READ COM-MITTED	No	No	Si	Si
REPEATABLE READ	No	No	No	Si
SERIALIZABLE	No	No	No	No

Cuadro 1: Violaciones que se pueden producir en cada nivel

21 Schedules de Transacciones

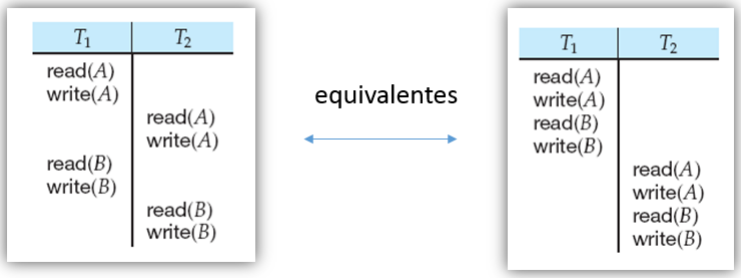


Figura 27: *Schedules* equivalentes

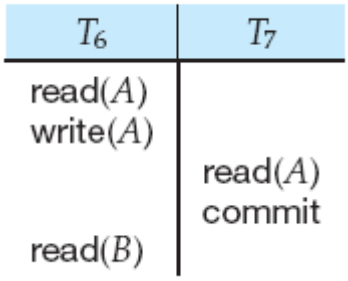


Figura 28: *Schedule* no recuperable

Schedule ordenamiento cronológico de las operaciones de n transacciones. Se pueden intercalar operaciones de distintas transacciones, pero las operaciones de una transacción se deben ejecutar en forma secuencial.

Un *schedule* es **recuperable** si, para cada par de transacciones T_i y T_j tal que T_j lee un ítem de dato previamente escrito por T_i, el `commit` de T_i aparece *antes* del `commit` de T_j.

Un *schedule* es **sin cascada** (***avoids cascading rollback*, AVR**) si, para cada par de transacciones T_i y T_j tal que T_j lee un ítem de dato previamente escrito por T_i, el `commit` de T_i aparece *antes* del `read` de T_j.

Un *schedule* que usa *locks* es **estricto** si cada transacción commitea o aborta, y luego libera todos sus *locks* exclusivos.

$$\text{estricto} \implies \text{sin cascada} \implies \text{recuperable}$$

$$\text{estricto} \implies \text{serializable}$$

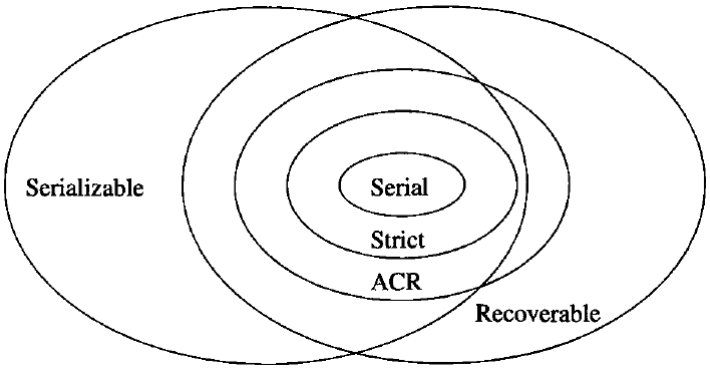


Figura 29: Relación de *schedules*

Conflicto dos *operaciones* en un *schedule* entran en conflicto cuando se cumplen todas las condiciones siguientes:

1. Corresponden a distintas transacciones
2. Acceden al mismo ítem de dato X
3. Al menos una de esas operaciones es `escribir(X)`

Dicho de otra forma, están en conflicto cuando, si alteramos el orden de ejecución, cambia el resultado final de X.

Un schedule es **serial** cuando ejecuta primero todas las operaciones de T_1, luego todas las operaciones de T_2, ..., y finalmente todas las operaciones de T_n. Cualquier esquema serial preserva la consistencia de la base de datos.

Un *schedule* es **serializable** cuando es equivalente a algún *schedule* serial con las mismas transacciones.

Dos *schedules* son **conflicto-equivalentes** si el orden de las operaciones en conflicto es la misma en ambos. Es decir, si podemos transformar uno en el otro mediante una secuencia de *swaps* de acciones adyacentes que no están en conflicto.

$$\begin{array}{l}
r_1(A);\ w_1(A);\ r_2(A);\ \underline{w_2(A)};\ \underline{r_1(B)};\ w_1(B);\ r_2(B);\ w_2(B);\\
r_1(A);\ w_1(A);\ \underline{r_2(A)};\ \underline{r_1(B)};\ w_2(A);\ w_1(B);\ r_2(B);\ w_2(B);\\
r_1(A);\ w_1(A);\ r_1(B);\ r_2(A);\ \underline{w_2(A)};\ \underline{w_1(B)};\ r_2(B);\ w_2(B);\\
r_1(A);\ w_1(A);\ r_1(B);\ \underline{r_2(A)};\ \underline{w_1(B)};\ w_2(A);\ r_2(B);\ w_2(B);\\
r_1(A);\ w_1(A);\ r_1(B);\ \underline{w_1(B)};\ r_2(A);\ w_2(A);\ r_2(B);\ w_2(B);
\end{array}$$

Figura 30: Convirtiendo un schedule conflicto-serializable en uno serial mediante *swaps* de acciones adyacentes. A cada paso están subrayadas las acciones a punto de ser *swappeadas*

Un *schedule* es **conflicto-serializable** si es conflicto-equivalente a algún *schedule* serial.

$$Conflicto\,serializable \implies Serializable$$

Para verificar si un *schedule* S es conflicto-serializable se utiliza un grafo dirigido $G = (N, E)$:

- N es el conjunto de transacciones $\{T_1, T_2, \ldots, T_n\}$ que forman el *schedule*
- E es el conjunto de aristas de la forma $T_i \to T_j$. Cada arista significa que T_i debe preceder a T_j en cualquier *schedule* serial equivalente a S

Algoritmo 15 Verificación de serializabilidad de un *schedule*

```
Entrada: schedule S
Salida: "verdadero" si es conflicto serializable

        para cada transaccion Ti en S:
                crear un nodo Ti en el grafo de precedencia
        para cada caso donde Ti ejecuta READ(X) y luego Tj ejecuta WRITE(X):
                agregar una arista (Ti -> Tj) en el grafo de precedencia
        para cada caso donde Ti ejecuta WRITE(X) y luego Tj ejecuta READ(X):
                agregar una arista (Ti -> Tj) en el grafo de precedencia
        para cada caso donde Ti ejecuta WRITE(X) y luego Tj ejecuta WRITE(X):
                agregar una arista (Ti -> Tj) en el grafo de precedencia

        si el grafo es aciclico:
                devolver "verdadero"
        si el gafo es ciclico:
                devolver "falso"
```

Si el grafo de precedencia es acíclico, el *schedule* serial equivalente puede construirse utilizando un **orden topológico** de la siguiente forma: siempre que exista una arista $T_i \rightarrow T_j$, T_i debe preceder a T_j en el *schedule* serial.

En la práctica, los DBMS no utilizan este algoritmo para garantizar la serializabilidad (porque habría que verificarlo para cada *schedule*).

22 Protocolos de Control de Concurrencia: Protocolo de Locking

Una forma de garantizar el aislamiento de transacciones y prevenir comportamiento no serializable es mediante el uso de *locks*. Notar que este mecanismo no funciona bien cuando los *locks* se mantienen durante días, o cuando las decisiones humanas son parte de una transacción.

Lock variable que se asocia a un ítem de dato. Describe el estado del ítem con respecto a las posibles operaciones que pueden aplicarse a el. Generalmente hay un *lock* por item de dato.

Tipos de *locks*:

1. **Binarios**: pueden tener dos valores: *locked* (1) o *unlocked* (0). Un *lock* binario impone la exclusión mutua en el item de dato asociado, porque solo puede haber a lo sumo una transacción con un lock para un mismo dato.

   El registro de este tipo de *lock* tiene la forma `<Item de dato, LOCK, Transacción con lock>`

   Reglas que deben aplicarse a cada transacción:

   a) `lock(X)` antes de cualquier `leer(X)` o `escribir(X)`
   b) `unlock(X)` después de **todos** los `leer(X)` y `escribir(X)`
   c) no se permite ejecutar `lock(X)` si ya se tiene un *lock* de X
   d) no se permite ejecutar `unlock(X)` si no se tiene el *lock* de X

2. **Compartidos/Exclusivos (Lectura/Escritura)**: es más laxo que los *locks* binarios porque permite que varias transacciones que solo van a ejecutar `leer(X)` tengan acceso a X. El *lock* puede tener dos valores: *read-locked* o *write-locked*.

   - **Lock exclusivo**: puede leer y escribir el dato X.
   - **Lock compartido**: puede leer pero no puede escribir X.

   El registro este tipo de *lock* tiene la forma `<Item de dato, LOCK, # de lecturas, Transaccion(es) con lock>`

   Reglas que deben aplicarse a cada transacción:

   a) `read_lock(X)` o `write_lock(X)` antes de cualquier $leer(X)$
   b) `write_lock(X)` antes de cualquier $escribir(X)$
   c) luego de todos los $leer(X)$ y $escribir(X)$, ejecutar $unlock(X)$
   d) si ya tiene un *lock* (compartido o exclusivo) de X, no se puede ejecutar ni `read_lock(X)` ni `write_lock(X)`

e) no se permite ejecutar $unlock(X)$ si no se tiene un *lock* de X (compartido o exclusivo)

Problemas con el uso de *locks*:

1. El uso por sí solo de *locks* no garantiza la serializabilidad de *schedules*.

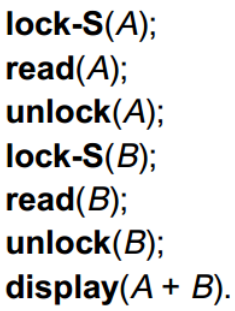


Figura 31: El uso de *locks* no garantiza la serializabilidad. Si se produce una actualización a A entre $unlock(A)$ y $lock(B)$, $A + B$ sería incorrecto

2. ***Deadlock***: ocurre cuando cada transacción en un *schedule* está esperando que se libere un *lock* que fue adquirido por otra transacción del *schedule*.

   Para detectar y/o prevenir un *deadlock*, el gestor de concurrencia puede mantener un **grafo de espera**. Cuando una transacción T está esperando a que la transacción S libere un *lock*, se dibuja una arista $T \to S$. Cuando S libera el *lock*, la arista se borra. El deadlock se detecta cuando el grafo es cíclico.

   Cuando ocurre un *deadlock*, el sistema debe ejecutar el *rollback* de alguna de las dos transacciones, y liberar los *locks* que ésta poseía.

   También se pueden arreglar *deadlocks* especificando un *timeout* para cada transacción. Si se ejecutan por más tiempo que este *timeout*, es abortada y sus *locks* se liberan.

3. ***Starvation***: ocurre cuando una transacción se queda esperando indefinidamente a que se libere un *lock*. Se puede evitar de la siguiente forma: cuando una transacción T_i solicita un *lock* sobre X en un modo M, el gestor de concurrencia se lo provee si se verifica que:
   a) No hay otra transacción con un *lock* sobre X en un modo que conflictúe con M.
   b) No hay otra transacción esperando obtener un *lock* sobre X antes que T_i

Lock de update un *lock* de *update* le da a una transacción el privilegio para leer X, pero **no** para escribirlo. Sin embargo, este tipo de *lock* se puede actualizar a un *lock* exclusivo más tarde.

Matriz de compatibilidad de locks se puede otorgar un *lock* de tipo C si y solo si, para cada fila R tal que ya se otorgó a otra transacción un lock de tipo R, hay un "Si" en la columna C.

		Se solicita un lock de tipo....	
		Exclusivo	Compartido
Se otorgó un lock de tipo...	Exclusivo	No	No
	Compartido	No	Si

(a) Para locks compartidos y exclusivos

		Se solicita un *lock* de tipo....		
		Exclusivo	Compartido	Actualización
Se otorgó un *lock* de tipo...	Exclusivo	No	No	No
	Compartido	No	Si	Si
	Actualización	No	No	No

(b) Para locks compartidos, exclusivos y de actualización

Cuadro 2: Ejemplos de matrices de compatibilidad

Lock table tabla que utiliza el gestor de concurrencia para almacenar el estado de los *locks* activos.

Para cada ítem de dato con *locks* activos, mantiene una lista de transacciones que solicitaron el *lock*, en el orden en que llegaron los pedidos. Se registra qué transacción es y qué modo de *lock* solicitó. También se registra si el *lock* le fue concedido.

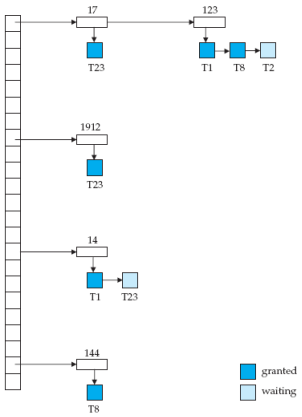


Figura 32: *Lock table*

22.1 Protocolo de dos fases (*two-phase locking*)

Una transacción cumple con el protocolo ***two-phase locking*** **(2PL)** si todas las operaciones de *lock* (`read_lock`, `write_lock`) preceden al primer *unlock* de la transacción. Se dice entonces que la transacción se divide en dos fases: la creciente (donde se adquieren los *locks*) y la decreciente (donde se liberan los *locks*).

Si cada transacción de un *schedule* cumple el protocolo 2PL, se garantiza que el *schedule* es serializable. De hecho, las transacciones se pueden ordenar de acuerdo a sus *lock points* (el punto donde termina la fase creciente).

Este protocolo limita la cantidad de concurrencia que se permite, porque una transacción no puede liberar un *lock* hasta que haya adquirido un *lock* para todos los demás ítems, y entonces puede haber muchas otras transacciones esperando que se libere el primer *lock*. Además, no se previene el *deadlock*, ni los *rollbacks* en cascada.

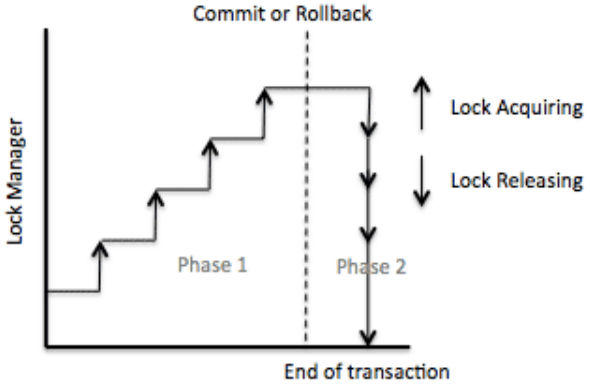


Figura 33: 2PL

T_3	T_4
lock-X(B)	
read(B)	
$B := B - 50$	
write(B)	
	lock-S(A)
	read(A)
	lock-S(B)
lock-X(A)	

Figura 34: Schedule 2PL con *deadlock*

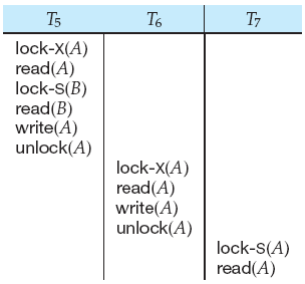


Figura 35: Schedule 2PL con *cascading rollback* si T_5 falla al final de todo

Para prevenir el problema de *deadlock*, este protocolo puede ampliarse y exigir que las transacciones adquieran por adelantado todos los *locks* que se necesitan; si un *lock* no se puede conseguir, no se adquiere ninguno. Esto limita aún más la concurrencia.

Para prevenir el problema de *rollbacks* en cascada, existe el protocolo **2PL estricto**. Este modo, además de requerir lo mismo que 2PL, requiere que todos los *locks* exclusivos adquiridos por una transacción se mantengan hasta que la misma ejecute *commit*.

22.2 Protocolo de árbol

Protocolo de árbol protocolo especializado para transacciones que acceden a datos en forma de árbol (ejemplo: un índice B). El protocolo viola 2PL, pero utiliza el hecho de que acceder a elementos debe ser hacia abajo para garantizar serializabilidad.

Si una transacción quiere insertar un registro, debería adquirir un *lock* exclusivo de la raíz, ergo de todo el árbol, y por ende estaría bloqueando a todas las demás transacciones.

Puede aprovecharse la estructura de árbol del índice de la siguiente forma:

- Cuando se adquiere un *read_lock* en un nodo hijo, el *lock* del nodo padre puede liberarse porque no se usará más.
- Cuando se adquiere un *write_lock* en un nodo hoja (para realizar una inserción), se debe adquirir un *lock* exclusivo en el nodo hoja.

Utilizar la técnica de ***index locking*** soluciona el problema de registros *phantom*.

El protocolo de árbol garantiza un orden serial en las transacciones. El orden de precedencia se define así: si T_i y T_j adquieren un *lock* sobre X y T_i adquiere el *lock* primero, entonces $T_i \rightarrow T_j$.

Algoritmo 16 Protocolo de árbol

1. El primer lock de una transacción puede hacerse sobre cualquier nodo del árbol.
2. Los locks subsiguientes pueden otorgarse sólo si se posee un lock sobre el nodo padre.
3. Se puede ejecutar *unlock* de un nodo en cualquier momento.
4. No se puede adquirir *lock* de un nodo 2 veces, incluso cuando se tiene un *lock* sobre el nodo padre.

Parte VI

Técnicas de Recuperación

23 Necesidad de Recuperación

Tipos de fallas que pueden ocurrir:

1. Fallas de la computadora: por ejemplo, una desconexión en la red
2. Fallas de transacciones: por ejemplo, una transacción que intenta dividir por cero
3. Errores locales: por ejemplo, una transacción que no encuentra datos
4. Aplicación de procedimientos de control de concurrencia: por ejemplo, una transacción abortada porque viola la serializabilidad o para resolver un estado de *deadlock*
5. Fallas del disco
6. Catástrofes

Los algoritmos de recuperación tienen dos partes:

1. Acciones que se toman durante el procesamiento normal de transacciones, para asegurar que, en caso de falla, se dispone de suficiente información para recuperar
2. Acciones que se toman después de una falla para devolver la base de datos a un estado consistente.

Idealmente, la base de datos en disco debería contener, para cada ítem de dato, el último valor escrito por una transacción que ejecutó *commit*.
En la práctica, la base de datos podría:

- Contener valores escritos por transacciones no commiteadas
- No contener valores escritos por transacciones commiteadas

24 Archivo de Log

La base de datos se almacena en disco. Éste está formado por bloques. Como todos los bloques no caben en memoria principal, se necesita una forma de trabajar con la base de datos en memoria. Por ende, se necesita una forma de organizar los bloques en memoria y luego copiarlos en el disco (*flush*).

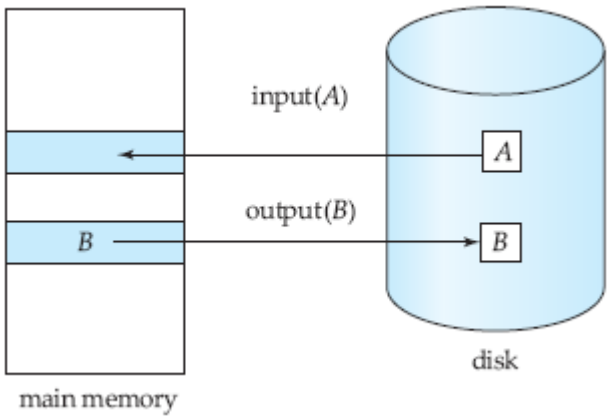


Figura 36: Operaciones sobre bloques

Para alcanzar el objetivo de transacciones atómicas, primero se debe guardar en disco información sobre las modificaciones, sin modificar la base de datos en sí. Para eso se utiliza un archivo de *log*. Este archivo permite:

- Deshacer (*undo*) cambios hechos por transacciones que deben ser abortadas
- Rehacer (*redo*) cambios hechos por transacciones que ejecutaron *commit* pero cuyos cambios no fueron almacenados en la base de datos en disco.

Log archivo secuencial al que solo se le pueden agregar registros.

Decimos que una transacción ejecutó **commit** cuando su registro `[commit,T]` fue almacenado en disco. Si hay una falla luego de esto, se ejecuta un **redo**. Si hay una falla antes de esto, se hace ***undo.***

- **Redo** debe hacerse en el orden en el que los cambios fueron hechos originalmente.
- **Undo** escribe registros especiales llamados "redo-only", que no tienen el valor viejo del item de dato. Al finalizar las escrituras, se escribe un registro `<abort,T>` para indicar que el *undo* finalizó.

24.1 Checkpoints

Cuando se produce una caída del sistema, hay que consultar el *log* para determinar aquellas transacciones que deben rehacerse o deshacerse. En principio, habría que buscar en todo el *log* para determinar esta información. Hay dos grandes dificultades con este enfoque:

1. El proceso de búsqueda lleva mucho tiempo.
2. La mayor parte de las transacciones ya han escrito sus actualizaciones en la base de datos.

Para reducir este tipo de gastos generales, se usan los ***checkpoints***. La periodicidad con que se ejecutan *checkpoints* la decide el administrador de base de datos.

24.1.1 Checkpoints bloqueantes

Se describe a continuación un esquema de control simple que (a) no permite realizar ningún cambio mientras la operación está en curso, y (b) se escriben en disco todos los *buffers* en memoria modificados.

Algoritmo 17 Checkpoint bloqueante en un log UNDO

1. Stop accepting new transactions.
2. Wait until all currently active transactions commit or abort and have written a `COMMIT` or `ABORT` record on the log.
3. Flush the log to disk.
4. Write a log record `<CHECKPOINT>`, and flush the log again.
5. Resume accepting transactions.

24.1.2 Checkpoints no bloqueantes

Algoritmo 18 Checkpoint no bloqueante en un log UNDO

1. Write a log record `<START CHECKPOINT (T1, ... , Tk)>`. Here, $T_1, \ldots, T_k$ are the identifiers for all the active transactions (i.e., transactions that have not yet committed and written their changes to disk).
2. Flush the log.
3. Wait until all of $T_1, \ldots, T_k$ commit or abort, but do not prohibit other transactions from starting.
4. When all of $T_1, \ldots, T_k$ have completed, write a log record `<END CHECKPOINT>`.
5. Flush the log.

Algoritmo 19 Checkpoint no bloqueante en un log REDO

1. Write a log record `<START CHECKPOINT (T1, ... , Tk)>`. Here, $T_1, \ldots, T_k$ are the identifiers for all the active transactions (i.e., transactions that have not yet committed).
2. Flush the log.
3. Write to disk all database elements that were written to buffers but not yet to disk by transactions that had already committed when the <START CHECKPOINT> record was written to the log.
4. Write a log record `<END CHECKPOINT>`.
5. Flush the log.

Algoritmo 20 Checkpoint no bloqueante en un log UNDO/REDO

1. Write a log record `<START CHECKPOINT (T1, ... , Tk)>`*. Here,* $T_1, \ldots, T_k$ *are the identifiers for all the active transactions (i.e., transactions that have not yet committed).*
2. Flush the log.
3. Write to disk ***all*** *buffers that are dirty (not just those written by committed transactions)*
4. Write a log record `<END CHECKPOINT>`*.*
5. Flush the log.

25 Protocolos de Recuperación

- Para fallas de tipo 5 y 6, se necesita haber grabado con anterioridad un *backup* de la base de datos y reconstruir la misma con el archivo de log hasta el momento de la falla.
- Para fallas de tipo 1 a 4, existen dos técnicas de recuperación:
  - ***Deferred update***

    Las transacciones y el manejador de buffers obedecen 1 regla:
    - **Write-Ahead Logging (WAL)**: se deben grabar en disco todos los registros de log sobre las actualización (incluyendo `<UPDATE T,X,Nuevo_valor>` y `<COMMIT T>`), y luego actualizar X en disco.

    Tipos de registros:
    - `<START T>`
    - `<COMMIT T>`
    - `<ABORT T>`
    - `<UPDATE T,X,Valor_Nuevo>`

Algoritmo 21 Procedimiento de recuperación **REDO** sin checkpoints

```
transacciones_commiteadas = identificarlas
transacciones_incompletas = identificarlas

desde el comienzo del log hasta el fin:
        si hay un registro <UPDATE <T,X,Valor_Nuevo>:
                si T está en "transacciones_commiteadas":
                        escribir "Valor_Nuevo" en X
                si no:
                        no hacer nada

para cada T en "transacciones_incompletas":
        escribir <ABORT T>

flush_log()
```

Algoritmo 22 Procedimiento de recuperación **REDO** con checkpoints no bloqueantes

```
transacciones_commiteadas = identificarlas
transacciones_incompletas = identificarlas

desde el comienzo del log hasta el fin:
        si hay un registro <UPDATE <T,X,Valor_Nuevo>:
                si T está en "transacciones_commiteadas":
                        escribir "Valor_Nuevo" en X
                si no:
                        no hacer nada
        si el ultimo registro de checkpoint es <END CHECKPOINT>:
                // se debe mirar el log hasta el <START Ti> que se ejecuto primero
        si el ultimo registro de checkpoint es <START CHECKPOINT T1,...,TK>:
                // la caida se produjo durante el checkpointing
                // buscar el registro <END CHECKPOINT> anterior y su par <START CHECKPOINT S1,..
                // rehacer todas las transacciones commiteadas que empezaron despues de ese STAR
                //      o son Si
                // (si no hay, rehacer desde el principio del log)

para cada T en transacciones_incompletas:
        escribir <ABORT T>

flush_log()
```

El algoritmo anterior puede ser más eficiente si se ejecuta, para cada ítem de dato, sólo el último REDO existente (porque todos los anteriores serían sobrescritos por éste).

Este mecanismo garantiza:

- Que no se deban hacer *rollbacks* de transacciones (porque las mismas sólo escriben en la base de datos luego de ejecutar commit)
- Que no se deban hacer *rollbacks* en cascada (porque los ítems tienen locks que no permiten leerlos antes de que una transacción que los escribió no ejecute commit)

- ***Immediate update***: a su vez tiene dos variantes:
  1. **UNDO/REDO**: Las transacciones y el manejador de buffer obedecen 1 regla:
     - Si la transacción T modifica el ítem de dato X, el registro de log <UPDATE T,X,Valor_Viejo,Valor_Nuevo> debe ser escrito al disco **antes** que el ítem X sea escrito al disco.

     Tipos de registros:
     - <START T>
     - <COMMIT T>
     - <ABORT T>
     - <UPDATE T,X,Valor_Viejo,Valor_Nuevo>

Algoritmo 23 Procedimiento de recuperación **UNDO/REDO**

```
mantener dos listas de transacciones:
1) transacciones commiteadas desde el ultimo checkpoint
2) transacciones activas

ejecutar REDO de todos los ESCRIBIR de las transacciones de la primera lista,
en el orden en que fueron escritos en el log

ejecutar UNDO de todos los ESCRIBIR de las transacciones de la segunda lista,
en el orden inverso en que fueron escritos en el log
```

  2. **UNDO**: Las transacciones y el manejador de buffer obedecen 2 reglas:
     a) Si la transacción T modifica el ítem de dato X, el registro de log <UPDATE T,X,Valor_Viejo> debe ser escrito al disco **antes** que el ítem X sea escrito al disco.
     b) ***Force Log at Commit*** **(FLC)**: si la transacción ejecuta commit, los ítems de datos actualizados por la transacción se deben escribir en el disco, y luego el registro de log <COMMIT T> debe ser escrito al disco.

Tipos de registros:

- <START T>
- <COMMIT T>
- <ABORT T>
- <UPDATE T,X,Valor_Viejo>

Algoritmo 24 Procedimiento de recuperación **UNDO** con checkpoint bloqueante

```
transacciones_completas = {}
transacciones_incompletas = {}

desde el fin del log hasta el comienzo: // o hasta que se encuentre un registro <CHECKPOINT>
        si hay un registro <COMMIT T> o <ABORT T>:
                transacciones_completas += T
        si hay un registro <UPDATE <T,X,Valor_Viejo>:
                si T está en "transacciones_completas":
                        no hacer nada
                sino:
                        transacciones_incompletas += T
                        escribir "Valor_Viejo" en "X"

para cada T en "transacciones_incompletas":
        escribir <ABORT T>

flush_log()
```

Algoritmo 25 Procedimiento de recuperación **UNDO** con checkpoint no bloqueante

```
transacciones_completas = {}
transacciones_incompletas = {}

desde el fin del log:
        si hay un registro <COMMIT T> o <ABORT T>:
                transacciones_completas += T
        si hay un registro <UPDATE <T,X,Valor_Viejo>:
                si T está en "transacciones_completas":
                        no hacer nada
                sino:
                        transacciones_incompletas += T
                        escribir "Valor_Viejo" en "X"
        si hay un registro <END CHECKPOINT>:
                // se debe mirar el log hasta el <START CHECKPOINT> correspondiente
        si hay un registro <START CHECKPOINT T1,...,TK> pero no un <END CHECKPOINT>:
                // la caida se produjo durante el checkpointing
                // se debe mirar el log hasta el comienzo de la primera transacción incompleta

para cada T en "transacciones_incompletas":
        escribir <ABORT T>

flush_log()
```

Es necesario que las operaciones de UNDO y REDO sean **idempotentes**: ejecutarlas muchas veces debe ser igual a ejecutarlas muchas veces. De hecho, todo el proceso de recuperación debe ser idempotente para garantizar que si existe una falla durante la recuperación de una falla, la misma se pueda recuperar también.

Es necesario que el DBMS mantenga:

- Lista de transacciones activas
- Lista de transacciones que ejecutaron commit
- Lista de transacciones abortadas desde el último checkpoint

Step	Action	t	M-A	M-B	D-A	D-B	Log
1)							<START T>
2)	READ(A,t)	8	8		8	8	
3)	t := t*2	16	8		8	8	
4)	WRITE(A,t)	16	16		8	8	<$T, A, 8$>
5)	READ(B,t)	8	16	8	8	8	
6)	t := t*2	16	16	8	8	8	
7)	WRITE(B,t)	16	16	16	8	8	<$T, B, 8$>
8)	FLUSH LOG						
9)	OUTPUT(A)	16	16	16	16	8	
10)	OUTPUT(B)	16	16	16	16	16	
11)							<COMMIT T>
12)	FLUSH LOG						

(a) Log UNDO

Step	Action	t	M-A	M-B	D-A	D-B	Log
1)							<START T>
2)	READ(A,t)	8	8		8	8	
3)	t := t*2	16	8		8	8	
4)	WRITE(A,t)	16	16		8	8	<$T, A, 16$>
5)	READ(B,t)	8	16	8	8	8	
6)	t := t*2	16	16	8	8	8	
7)	WRITE(B,t)	16	16	16	8	8	<$T, B, 16$>
8)							<COMMIT T>
9)	FLUSH LOG						
10)	OUTPUT(A)	16	16	16	16	8	
11)	OUTPUT(B)	16	16	16	16	16	

(b) Log REDO

Step	Action	t	M-A	M-B	D-A	D-B	Log
1)							<START T>
2)	READ(A,t)	8	8		8	8	
3)	t := t*2	16	8		8	8	
4)	WRITE(A,t)	16	16		8	8	<$T, A, 8, 16$>
5)	READ(B,t)	8	16	8	8	8	
6)	t := t*2	16	16	8	8	8	
7)	WRITE(B,t)	16	16	16	8	8	<$T, B, 8, 16$>
8)	FLUSH LOG						
9)	OUTPUT(A)	16	16	16	16	8	
10)							<COMMIT T>
11)	OUTPUT(B)	16	16	16	16	16	

(c) Log UNDO/REDO

Figura 37: Ejemplos de logs

UNDO	REDO	UNDO/REDO
$Logs\ X \implies Bd\ X \implies Commit\ al\ log$	$Logs\ X\ \&\ Commit\ al\ log \implies Bd\ X$	$Log\ X \implies Bd\ X$
Immediate update	*Deferred update*	*Immediate update*