Universidad de Buenos Aires

Facultad de Ciencias exactas y Naturales

Licenciatura en Ciencias de la computación

Apunte de Bases de Datos

Autor:
Julián Sackmann

10 de Septiembre de 2012

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Ciudad Universitaria - (Pabellón I/Planta Baja)

Intendente Güiraldes 2160 - C1428EGA

Ciudad Autónoma de Buenos Aires - Rep. Argentina

Tel/Fax: (54 11) 4576-3359

http://exactas.uba.ar

Índice

1 Algunas definiciones

- **Base de datos**: un conjunto de datos relacionados con un significado inherente (un conjunto aleatorio de datos no es considerado una base de datos). Una base de datos es diseñada y construida con un propósito específico.
- **Dato**: hecho conocido que puede ser registrado y tiene un significado implícito.
- **Minimundo**: aspecto acotado del mundo real representado por la base de datos. Sus cambios deben verse reflejados en la base.
- `DBMS`: database management system. Es un software de propósito general (o colección de) que permite a los usuarios crear y mantener una base de datos. EL `DBMS` no sólo contiene los datos en si mismos sino una definición completa de su estructura
- **Catálogo**: almacena información concerniente a la definición, estructura y demás metadata de la base de datos.
- **Estructura de la DB**: tipos de datos, relaciones y restricciones.
- **Construcción de una DB**: proeso de almacenar los datos en algún medio controlable por el DMBS.
- **Abstracción de datos**: ss la característica que permite la independencia del programa con los datos almacenados y las operaciones sobre ellos. El `DBMS` provee a los usuarios con una representación conceptual de la data abstrayendo los detalles de su almacenamiento.
- **Vista**: puede ser un subconjunto de la base de datos o puede contener datos virtuales (datos que se derivan de los datos existentes pero que no están realmente almacenados).
- **Transacción**: es un programa en ejecución que incluye uno o más accesos a una base de datos. Cada transacción se supone ejecuta un acceso correcto y completo.
- **ACID**:
- **DBA**: es la persona resposable de autorizar accesos a la base de datos, coordinar y monitorear su uso y adquirir software y hardware necesario. Es el último responsable de los problemas de la base de datos (sean de performance, seguridad, etc.)
- **Redundancia**: consiste en almacenar los mismo datos en múltiples lugares. En general puede ser problemático porque hay que mantener todos esos lugares actualizados y seguros, duplicando el esfuerzo (sin menciona el espacio gastado), pero a veces se hace en pos de performance.
- **Denormalización**: proceso de ubicar datos pegados para no tener que buscarlos en múltiples archivos.
- **Modelo de datos**: conjunto de conceptos usados para describir estructura de una base de datos. Provee los medios necesarios para lograr la abstracción.
- **Entidad**: representa un objeto o concepto del minimundo que será incluido en la base de datos.
- **Atributo**: representa una propiedad de interés que describe la entidad.
- **Relación**: representa una asociación entre las entidades.
- **Estado**: todos los datos de la base de datos en un momento particular. Es responsabilidad del `DBMS` asegurarse que todo estado de la base de datos sea válido (satisface la estructura y restricciones especificados en el esquema).
- **Esquemas**:

1. **Interno**: describe el almacenamiento físico de la estructura de la DB.
2. **Conceptual**: describe la estructura de la base de datos para un conjunto de usuarios.
3. **Externo (o de vista)**: describe una parte de la base de datos que interesa a un grupo particular de usuarios. Suele haber varios externos

- **Independencia *lógica* de datos**: capacidad de cambiar el esquema conceptual sin tener que cambiar el externo ni los programas de aplicación.
- **Independencia *física* de datos**: capacidad de cambiar el esquema interno sin tener que cambiar el conceptual.
- **DDL**: Data Definition Languje. Lenguaje usado en `DBMS`s sin clara separación entre esquemas para definir los esquemas interno y conceptual de la base de datos.
- **SDL**: Storage Definition Languaje. En `DBMS`s donde existe una separación explícita entre los esquemas interno y conceptual, el SDL se utiliza para especificar el esquema interno (y el DDL queda exclusivo para el conceptual.)
- **DML**: Data Manipulation Languaje. Lenguaje usado en los `DBMS` para obtener, agregar o modificar datos.

2 Claves

2.1 Superclave

Una **superclave** de un esquema $R = A_1, A_2, ..., A_n$ es un conjunto de atributos $S \subseteq R$ tal no pueden existir dos tuplas t_1 y t_2 en R tales que $t_1[S] = t_2[S]$.

Una **clave** es una superclave minimal. Si un esquema tiene más de una clave, a cada una se la llama **clave candidata** y se selecciona una de ellas para ser la clave **primaria**.

Se dice que un atributo $X \in R$ es **primo** si es parte de alguna clave candidata.

3 Dependencias Funcionales

Una dependencia funcional (notada como $X \to Y$) entre dos conjuntos de atributos X e Y subconjuntos de un esquema R especifica una restricción *semántica* al conjunto de tuplas que pueden formar un estado r de R. La restricción es que para todo par de tuplas t_1 y t_2 en R tal que $t_1[X] = t_2[X]$ necesariamente debe valer que $t_1[Y] = t_2[Y]$. Informalmente, si dos tuplas tienen el mismo valor de X, necesariamente deben tener el mismo valor en Y.

Observaciones:

- Si X es superclave de R, entonces $X \to Y$ es cierto para todo Y.
- Una dependencia funcional es una propiedad **semántica**, del significado de los atributos. **No es posible determinar una dependencia funcional de una instancia de un esquema**.

3.1 Dependencias funcionales completas

Una dependencia funcional $X \to Y$ se dice **completa** si al sacar cualquier atributo de X la dependencia se rompe. Formalmente, $X \to Y$ es completa sii (X
$\{A\} \to Y$ es falso para todo A.

Una dependencia funcional es **parcial** si no es completa.

3.2 Dependencias funcionales triviales

Una dependencia funcional $X \to Y$ en R es trivial si $Y \subseteq X$.

3.3 Dependencias funcionales transitivas

Una dependencia funcional $X \to Y$ en R es **transitiva** si existe un conjunto de atributos $Z \subseteq R$ tal que:

- No es subconjunto de ninguna clave.
- $X \to Z$.
- $Z \to Y$.

3.4 Clausura de dependencias funcionales

F se infiere $X \to Y$ si y sólo si $Y \subseteq X^+$.

3.5 Cubrimiento minimal

Un **cubrimiento minimal** F' de un conjunto de dependencias funcionales F es un conjunto de dependencias funcionales que cumple:

- $F'^+ = F^+$
- El lado derecho de todas las dependencias en F' tiene un solo atributo.
- No hay atributos redundantes en el lado izquierdo.
- No hay dependencias funcionales redundantes.

Siempre hay un cubrimiento minimal. Se usa en el algoritmo de descomposición

4 Formas Normales

4.1 Buen diseño

Por si solas, las formas normales no garantizan un buen diseño de bases de datos. No es suficiente decir que un esquema en tercer forma normal para concluir que es un buen esquema. Para eso, el proceso de normalización debe garantizar la existencia de dos propiedades:

- **Lossless join**: garantiza que no se generen tuplas espurias a la hora de realizar un *natural join*.
- **Conservación de dependencias funcionales**: garantiza que cada dependencia funcional quede representada en una sola tabla.

La propiedad de *lossless join* se considera extremadamente crítica y debe ser lograda siempre. Sin embargo, a veces se sacrifica la propiedad de connservación de dependencias funcionales ya que no es tan vital.

4.2 Primer forma normal

Un esquema R está en **primer forma normal** si sus atributos incluyen sólo **valores atómicos**. En primer forma normal se prohibe tener tuplas o conjuntos como atributos de relación.

4.3 Segunda forma normal

Una relación está en segunda forma normal si todo atributo no primo tiene una dependencia funcional total con la clave primaria de R.

4.4 Tercer forma normal

Una relación está en tercer forma normal si para toda dependencia funcional $X \to Y$ en R *no trivial* vale **una** de las siguientes condiciones:

- X es una superclave de R.
- Y es un atributo primo de R.

Observemos que una relación que viola la tercer forma normal es aquella en la que ambas condiciones son falsas. Esto puede pasar con dos tipos de dependencias funcionales: un atributo no primo determinando funcionalmente otro atributo no primo **o** un sobconjunto de una clave determinando funcionalmente un atriuto no primo.

4.5 Forma normal de Boyce Codd

Una relación está en tercer forma normal si para toda dependencia funcional $X \to Y$ en R *no trivial*, X es una superclave de R.

Observemos que la única forma que una relación esté en tercer forma normal, pero no en forma normal de Boyce Codd es si en ella existe una dependencia funciona $X \to Y$ en la que X no sea una superclave e Y sea un atributo primo.

Observemos también que cualquier esquema de relación con sólo dos atributos está inmediatamente en forma normal de Boyce Codd.

5 Almacenamiento físico

5.1 Heap File

Un **heap file** es la forma más básica de almacenamiento: los registros son almacenados en archivos en el orden en el que son insertados. La inserción de registros nuevos es extremadamente eficiente, pero al no estar ordenados buscar la existencia de un registro implica hacer una búsqueda lineal en todos los registros. Similarmente, borrar registros tiene problemas de dejar espacio inútil en el medio.

5.2 Sorted File

Un **sorted file** es una forma de almacenamiento de registros que los ordena por alguno de sus atributos. Los archivos ordenados tienen la ventaja de que es mucho más eficiente la búsqueda puesto que se puede hacer búsqueda binaria (siempre que se esté buscando por el atributo de ordenamiento).

Los *sorted file* no proveen ventajas para búsquedas aleatorias u ordenadas por atributos que no sean los de ordenamiento. En esos casos se hace una búsqueda lineal, al igual que en el heap file.

Insertar y borrar registros en *sorted files* son operaciones caras, puesto que deben ser insertados (o eliminados) de los lugares correctos, manteniendo el invariante.

6 Índices

Un **índice** es una estructura de acceso utilizada para acelerar la obtención de registros si se cumplen determinadas condiciones de búsqueda. Se implementan mediante archivos adicionales que proveen accesos

secundarios sin afectar la distribución original de los registros en la tabla (evitando tener problemas de duplicación de información del estilo *cache*).

Un índice se construye en base a uno o más atributos, que determinan la condición sobre la que se puede buscar más eficientemente usando el índice. Es importante notar que se pueden crear cuantos índices se quieran sobre una tabla, pero no necesariamente vale que más índices $\Rightarrow$ mejor performance. Los índices, como cualquier estructura de datos tienen costos de *bookkeeping* que es necesario contrastar contra las ventajas de acceso. Particularmente los índices suelen hacer más costosas las operaciones de inserción y borrado.

6.1 Factor de bloqueo

El **factor de bloqueo** de una tabla es un valor que permite calcular la cantidad de bloques de disco ocupa una determinada tabla. Informalmente, representa cuántas entradas de una tabla entran en un bloque de disco.

Se calcula como:

$$\boxed{fbr = \left\lceil \frac{\text{tamaño de un registro}}{\text{tamaño del bloque}} \right\rceil}$$

Para buscar por rango como el archivo está ordenado simplemente hay que encontrar el primer registro que cumple la condición mínima y avanzar linealmente hasta obtener la máxima. Es el mismo costo que una búsqueda con igualdad (considerando la cantidad de registros a devolver posiblemente mayor).

6.2 Índice primario

Un índice es **primario** si en el índice se guarda toda la tupla que corresponde y no sólo un identificador y el puntero.

6.2.1 Costo

Para ubicar un registro en una tabla que posee un índice primario se puede realizar una búsqueda binaria, requiriendo $log_2(b_i) + 1$ accesos a disco.

Luego, podemos estimar la cantidad de accesos a disco como

$$\boxed{log_2\left(\left\lfloor \frac{\#registros}{\text{factor de bloqueo}} \right\rfloor\right) + 1}$$

Para buscar por rango como el archivo está ordenado simplemente hay que encontrar el primer registro que cumple la condición mínima y avanzar linealmente hasta obtener la máxima. Es el mismo costo que una búsqueda con igualdad (considerando la cantidad de registros a devolver posiblemente mayor).

$$\boxed{log_2\left(\left\lfloor \frac{\#registros}{\left\lceil \frac{\text{tamaño de un registro}}{\text{tamaño del bloque}} \right\rceil} \right\rfloor\right) + 1}$$

Para buscar por rango como el archivo está ordenado simplemente hay que encontrar el primer registro que cumple la condición mínima y avanzar linealmente hasta obtener la máxima. Es el mismo costo que una búsqueda con igualdad (considerando la cantidad de registros a devolver posiblemente mayor).

6.3 Índices densos o esparsos

Un índice es **denso** si tiene una entrada por cada registro en la tabla de datos.

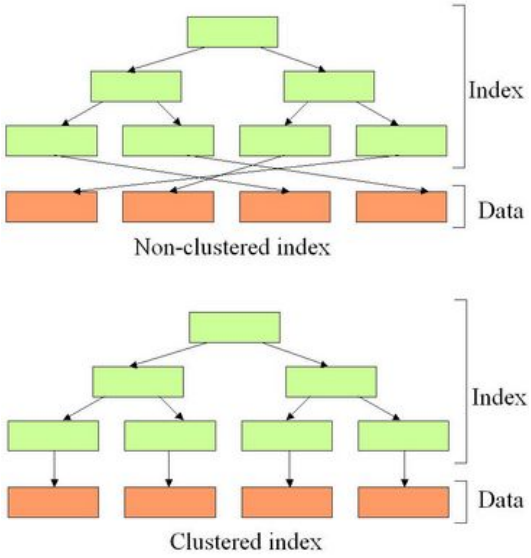
Por otro lado, un índice **esparso** tiene entradas sólo para algunos registros.

6.4 Índice clustered

Si los datos del archivo están ordenados físicamente en el mismo orden que uno de sus índices, decimos que ese índice es **clustered**. Caso contrario es **unclustered**.

Los archivos de datos a lo sumo pueden tener un índice clustered, en tanto que la cantidad de índices unclustered es ilimitada.

Observemos que por definición, todo índice clustered es esparso.

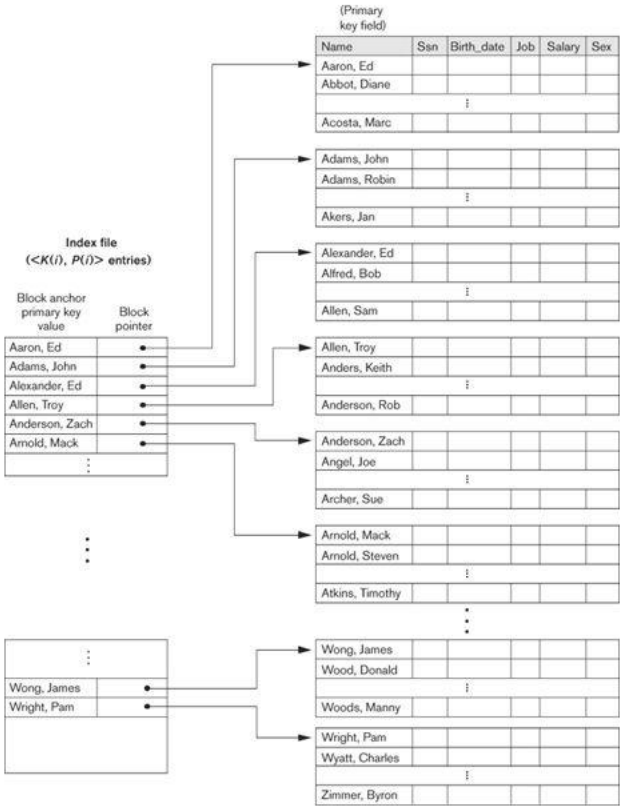


6.5 Índice secundario

Un índice **secundario** provee una herramienta para acceder más eficientemente a una tabla para la que ya existe un índice primario. O sea, **el archivo no está ordenado en función del campo para el que creo un índice secundario** (esto puede ser porque sea un *sorted file* ordenado por otro campo o porque estemos creando un índice sobre un *heap file*).

Se pueden crear muchos índices secundarios en una misma tabla.

Un índice secundario puede tener en su estructura interna un puntero a bloque de disco o a registro.



6.6 Índice B+

Un índice B+ es árbol balanceado con una cantidad variable de hijos por nodo. Las hojas están doblementen enlazadas para poder recorrerlas linealmente (evitando tener que bajar por toda la altura del árbol cada vez).

6.6.1 Clustered

Los índices árbol B+ clustered son aquellos para los cuales el archivo de datos asociado está ordenado en el mismo orden que dicho índice.

- No ayudan para exploración completa. Su costo es acceder a todos los bloques del archivo.
- Para buscar por igualdad, se debe recorrer el árbol desde la raíz hasta la hoja (peor caso $altura_arbol$ accesos a disco) y luego se debe obtener secuencialmente todos los registros que matcheen con el criterio de búsqueda.

$$\text{altura arbol} + \left\lceil \frac{\text{cantidad tuplas}}{\left\lfloor \frac{\text{tamaño registro}}{\text{tamaño bloque}} \right\rfloor} \right\rceil$$

- Para buscar por rango como el archivo está ordenado simplemente hay que encontrar el primer registro que cumple la condición mínima y avanzar linealmente hasta obtener la máxima. Es el mismo costo que una búsqueda con igualdad (considerando la cantidad de registros a devolver posiblemente mayor).

6.7 Tabla de costo de índices

Tipo de archivo / índice	Costo de exploración completa	Costo de búsqueda por igualdad (A = k)	Costo de búsqueda por rango ($k_1 <= A <= k_2$)
Heap file	B_R	B_R	B_R
Sorted file	B_R	$\log_2(B_R) + \lceil T' / FB_R \rceil$	$\log_2(B_R) + \lceil T' / FB_R \rceil$
Índice B+ clustered sobre A	-	$X_I + \lceil T' / FB_R \rceil$	$X_I + \lceil T' / FB_R \rceil$
Índice B+ unclustered sobre A	-	$X - 1 + \lceil T' / FB_I \rceil + T'$	$X - 1 + \lceil T' / FB_I \rceil + T'$
Índice hash estático sobre A	-	$MBxB_I + T'$	-

7 Procesamiento y optimización de queries

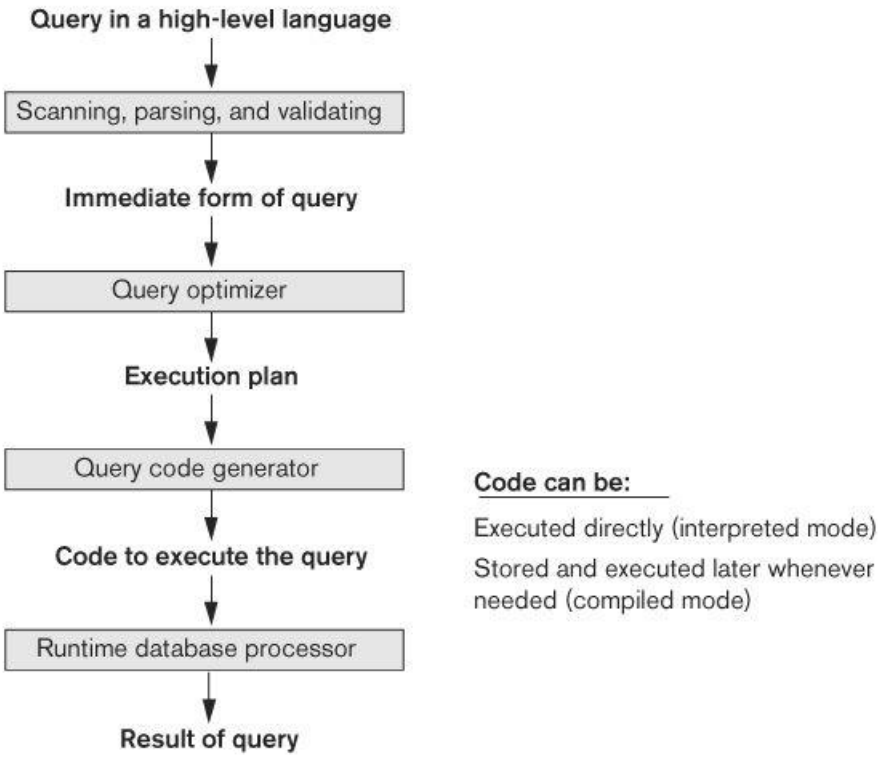
Una *query* suele tener muchas posibles formas de ser ejecutada para obtener el set de datos deseado. Cada una de esas formas de ejecución se llama **query plan**.

Uno de los componentes del `DBMS`, el **query optimizer** es el encargado de seleccionar cual de esos planes ejecutar. Como encontrar el plan óptimo es un problema NP-Completo, el *query optimizer* utiliza otras técnicas para encontrar uno suficientemente bueno en un tiempo razonable.

La optimización de la *query* tiene varios aspectos:

- Utilización de heurísticas: se aplican transformaciones del álgebra relacional que tienen la propiedad de mantener los resultados obtenidos. Las heurísticas suelen mejorar la performance, pero no es una garantía.
- Estimación de selectividad: el *optimizer* utiliza la información almacenada en el **catálogo** de la base de datos para estimar el grado de selectividad que tiene la query (*"cuántas tuplas devuelve"*). De esta forma se puede tener una medida estimativa de cuan "cara" va a ser la ejecución de dicha query.
- Índices y tipo de archivo: el plan de ejecución elegido depende muy fuertemente de los índices que se disponga en la tabla y cómo esté ordenado físicamente el archivo en disco.

Una vez obtenido el plan de ejecución, el **code generator** genera el código correspondiente a ese plan y el **runtime database processor** lo ejecuta.



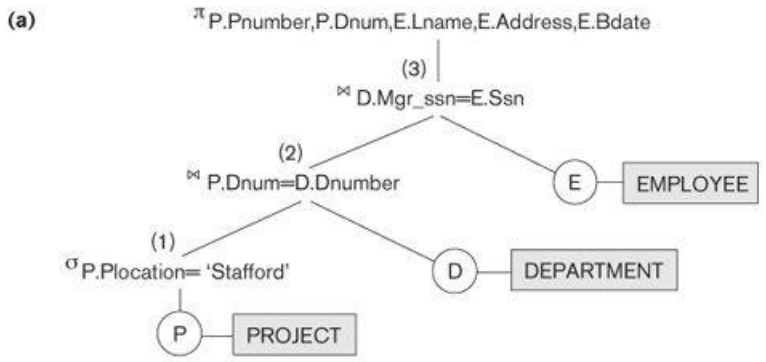
7.1 Pipeline

El camino para ejecutar una *query* comienza con un parseo y traducción de la *query* a una expresión del álgebra relacional, compuesta de muchas *operaciones*. Si ejecutáramos individualmente cada operación, y dado que las tablas no suelen entrar enteras en memoria, sería necesario grabar a archivos temporales en el disco para almacenar los resultados intermedios. Esto es extremadamente costoso por los accesos adicionales que requiere.

Para subsanar esto se aplican heurísticas que ordenan las operaciones de forma de que el input de una se pueda alimentar como output de la siguiente, reduciendo la cantidad de grabaciones a disco. No las eliminan por completo porque hay operaciones que requieren la *materialización* (bajada a disco) (por ejemplo los *join*). Esta técnica se conoce como **pipelining**.

7.2 Query tree

Un **query tree** (o árbol de *query*) es una estructura de datos que corresponed a una expresión del álgebra relacional. Las hojas coresponden a las relaciones mientras que los nodos intermedios representan operaciones.



7.3 Implementando queries

7.3.1 Select

Existen varias formas de ejecutar un `SELECT` dependiendo de los índices y la organización del archivo.

- Búsqueda lineal.
- Búsqueda binaria.
- Usando un índice primario.
- Usando una clave hash.
- Usando un índice B+.
  - Clustered.
  - Unclustered.

7.3.2 Join

Al igual que con *SELECT*, existen varias formas de ejecutar un `JOIN`:

- **Block Nested Loop Join** (`BNLJ`): es el approach de fuerza bruta. Si se tienen B bloques de memoria, se llenan $B-2$ con bloques de una de las tablas. Otro bloque se usa para ir iterando todos los bloques de la otra tabla y el restante para el resultado. Siempre conviene poner el archivo con menos bloques en la iteración exterior (o sea llenar los $B-2$ con él).
- **Index Nested Loop Join** (`INLJ`): se utiliza cuando se tiene un índice en una de las tablas que coincida con el atributo del join. Se itera sobre la otra tabla, buscando los atributos coincidentes utilizando el índice. El costo depende del tipo de índice.
- **Sort Merge Join** (`SMJ`): se ordenan ambas relaciones (si no estaban ordenadas) y se recorren ordenadamente. El costo es el de ordenar ambas relaciones (algoritmo de sorting en disco) y luego hacer el merge (lineal en ambos tamaños).

7.4 Heurísticas

- Cascada de σ: una conjunción de selecciones puede ser rota en operaciones individuales:

$$\sigma_{c_1 \text{ AND } c_2 \text{ AND ... AND } c_n} = \sigma_{c_1}(\sigma_{c_2}(...(\sigma_{c_n}(R))))$$

- Conmutatividad de σ: la operación de selección es conmutativa.

$$\sigma_{c_1}(\sigma_{c_2}(R)) = \sigma_{c_2}(\sigma_{c_1}(R))$$

- Cascada de π: en una secuencia de proyecciones, sólo es relevante la última.

$$\pi_{L_1}(\pi_{L_2}(...(\pi_{L_n}(R)))) = \pi_{L_1}(R)$$

- Conmutatividad de σ con π: si la condición de selección sólo involucra atributos de la lista, las dos operaciones se pueden conmutar.

$$\pi_{A_1,A_2,...,A_n}(\sigma_C(R)) = \sigma_C(\pi_{A_1,A_2,...,A_n}(R))$$

- Conmutatividad de $\bowtie$ y $\times$:

$$R \bowtie S = S \bowtie R$$
$$R \times S = S \times R$$

- Conmutatividad de σ con $\bowtie$: si todos los atributos de la condición de selección involucran (c) sólo atributos de una de las relaciones (digamos R), entonces las operaciones se pueden conmutar.

$$\sigma_c(R \bowtie S) = (\sigma_c(R)) \bowtie S$$

  La versión general de esto es que si la condición c se puede escribir como $c_1 AND c_2$ donde c_1 pertenecen sólo a R y c_s sólo a S, entonces:

$$\sigma_{c_1 \text{ AND } c_s}(R \bowtie S) = (\sigma_c 1(R) \bowtie \sigma_{c_2}(S))$$

- Conmutatividad de π con $\bowtie$. Idem anterior.
- Conmutatividad de operaciones de conjuntos.
- Asociatividad de $\times$, $\bowtie$, $\bigcup$ y $\bigcap$.
- Conversión de $(\sigma_c, \times)$ en un $(\bowtie_c)$: si tenemos una selección inmediatamente después de un producto cartesiano se puede convertir a un join.

$$\sigma_c(R \times S) = (R \bowtie_c S)$$

8 Transacciones

Una **transacción** es un programa en ejecución que forma una unidad lógica de procesamiento de base de datos. Incluye uno o más operaciones de acceso a la base de datos, que pueden ser inserciones, modificaciones, borrados, etc.

Las cuatro operaciones básicas de una transacción son: **read**, **write**, **commit** y **abort**.

8.1 Propiedades ACID

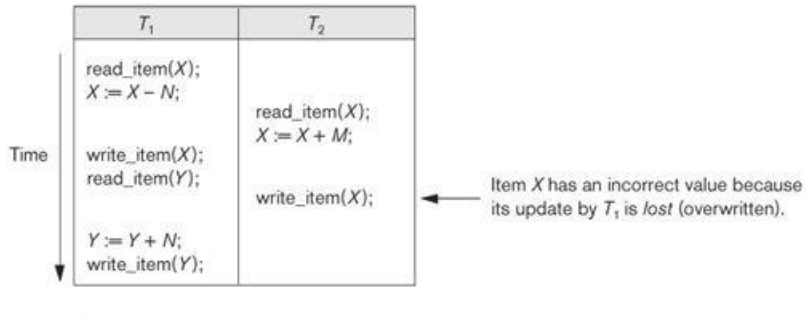
Las transacciones deben cumplir las propiedades `ACID`:

- **A**tomicity: una transacción debe ejecutarse completa o no ejecutarse del todo.
- **C**onsistency: una transacción debe tomar una base de datos en estado válido y dejarlo en un estado válido.
- **I**solation: ejecuciones concurrentes de transacciones deben arrojar el mismo resultado que si se hubieran ejecutado esas transacciones linealmente.
- **D**urability: una vez que una transacción commitea, los cambios que realizó son permanentes y no deben ser perdidos aún en el caso de fallas futuras (eléctricas, físicas, lógicas, etc).

8.2 Problemas de control de concurrencia

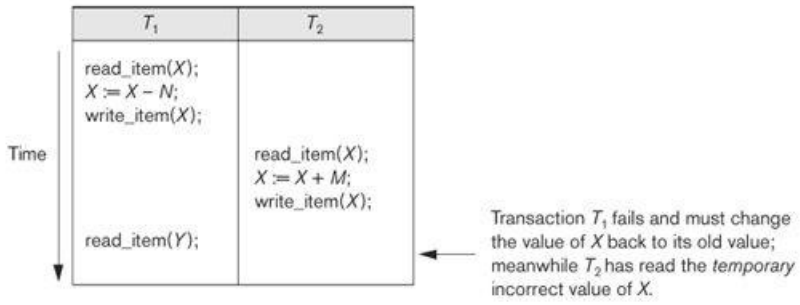
8.2.1 Lost update

El problema de **lost update** ocurre cuando dos transacciones que acceden al mismo ítem de la base de datos ven sus operaciones entrelazadas de tal forma que uno lee un valor para modificarlo y el otro lo lee antes que el primero pueda escribirlo:



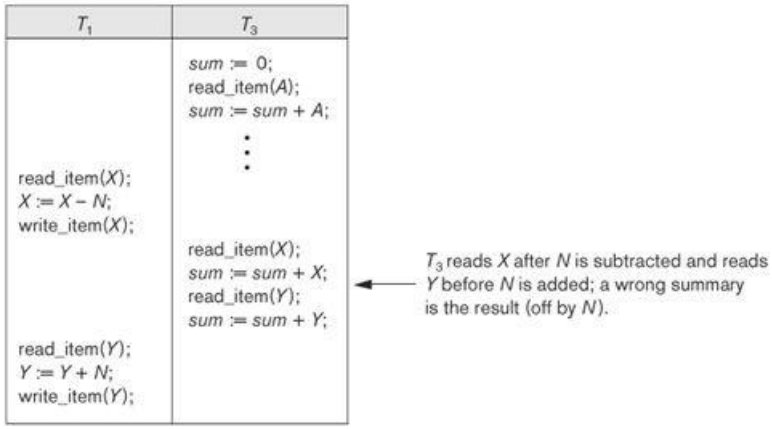
8.2.2 Dirty read

El problema de **dirty read** ocurre cuando una transacción actualiza un valor de la base de datos y luego aborta. Entonces si otra transacción usó ese valor para realizar algún cómputo, ese cómputo deja de ser válido.



8.2.3 Incorrect summary

El problema de **incorrect summary** ocurre cuando una transacción está calculando una suma de agregación en algunas tuplas de la base de datos mientras que otra transacción está actualizando las mismas tuplas.



8.2.4 Unrepeatable read

El problema de **unrepeatable read** ocurre cuando una transacción lee el mismo ítem dos veces consecutivas y obtiene distintos valores. Esto ocurre cuando una transacción no obtiene un lock individual sobre un ítem antes de acceder a él.

8.2.5 Phantom read

Se conoce como problema **phantom read** (o lectura fantasma) a la situación en la que se corren dos queries exactamente iguales y estas arrojan resultados distintos. Esto ocurre cuando las transacciones no obtienen locks de las tablas enteras (o de rangos) antes de realizar *queries*. Es un caso particular de *unrepeatable read*.

8.3 Niveles de acceso

Se definen los **niveles de aislamiento** de una transacción en función de qué problemas pueden ocurrirle:

	Dirty read	**Non repeatable read**	**Phantom read**
Read uncommited	Si	Si	Si
Read commited	No	Si	Si
Repeatable read	No	No	Si
Serializable	No	No	No

8.4 Historias

Un **schedule** (o historia) de n transacciónes $T_1, T_2, ..., T_n$ (llamado S) es un ordenamiento de las operaciones de las antedichas transacciones que mantiene el orden relativo de las operaciones de cada transacción. Esto significa que si dos operaciones o_1 y o_2 de una transacción T ocurrían o_1 antes que o_2 en T entonces necesariamente ocurrirá o_1 antes que o_2 en S.

8.4.1 Conflicto

Dos operaciones en una historia se dice que están en **conflicto** si se las siguientes tres condiciones:

- Pertenecen a transacciones diferentes.
- Acceden el mismo item.

- Al menos una de ellas es de escritura.

Intuitivamente, esta definición redunda en que dos operaciones son conflictivas si alterar su orden puede afectar el resultado.

8.4.2 Tipos de historias

Las historias se pueden categorizar de acuerdo a la siguiente jerarquía:

- **Serial**: una historia es **serial** si se ejecutan las transacciones secuencialmente, sin entrelazar sus operaciones.
- **Serializable**: una historia es **serializable** si su resultado final es equivalente a alguna ejecución serial de las mismas transacciones.
- **Recuperable**:
  - **Recuperable**: una historia es **recuperable** si ninguna transacción $T \in S$ *commitea* hasta que toda otra transacción $T' \in S$ que haya escrito un valor que T leyó comitee.
  - **Cascadeless**: se conoce como **aborto en cascada** al fenómeno que ocurre en algunas historias recuperables en los que una transacción que todavía no hizo su *commit* tiene que ser deshecha porque leyó de un ítem de una transacción que fue abortada. Luego, una transacción se dice que evita el aborto en cascada (es *cascadeless*) si cada transacción en la historia sólo lee aquellos items que fueron escritos por transacciones que ya commitearon.
  - **Estrícto**: un schedule es **estrícto** si las transacciones no pueden escribir ni leer un ítem x hasta que la última transacción que escribió el valor de x haya *commiteado* o abortado.

Dos historias se dicen **equivalentes en conflicto** si el orden de todo par de operaciones conflictivas es el mismo en ambas.

8.4.3 Grafo de precedencias

El **grafo de precedencias** de una historia es un grafo dirigido $G = (N, E)$ en el que los nodos son las transacciones $N = T_1, T_2, ..., T_n$. Existe un eje entre las transacciones j y k ($e = (T_j \to T_k)$) si una operación de T_j aparece en la historia antes que alguna operación de conflicto en T_k.

Informalmente, en el grafo de presedencia de una historia S un eje de T_i a T_j significa que T_i debe venir antes que T_j en cualquier operción serial equivalente a S porque dos operaciones conflictivas aparecen en ese orden.

Una historia es **serializable** si y sólo si su grafo de precedencias no tiene ciclos.

8.5 Logging y recuperación

El **system log** es el componente del `DBMS` encargado de mantener un registro de todas las operaciones realizadas por las transacciones que afectan los valores de los ítems de la base de datos, así como otra información de las transacciones útil para recuperar la consistencia de la base de datos en caso de falla. Este log es un archivo en el disco que se graba secuencialmente y sólo permite agregar información.

8.6 Locking

Un **lock** es una variable asociada con un item que describe el estado del ítem con respecto a las posibles operaciones que se pueden realizar sobre él.

El esquema de lockeo binario es demasiado restrictivo porque impone que una sola transacción puede leer el mismo ítem a la vez.

8.6.1 Binario vs Shared

Un **lock binario** es aquel que puede tener dos estados: bloqueado o liberado.
Un **shared lock** (o **read/write lock**) puede tener tres estados:

- *read_lock(X)*
- *write_lock(X)*
- *unlocked(X)*

8.6.2 Optimista vs Pesimista

Un lock es **optimista** si permite realizar todas las operaciones, pero falla cuando se va a *commitear* esos cambios. En un lockeo optimista una operación de *read* o *write* puede desencadenar un *rollback* de la transacción. Esto puede generar un *livelock* a una transacción. Es un modelo más probabilístico: funciona mejor cuando hay pocos conflictos.

Si el lock impide el acceso a los datos en el momento que se solicitan (y no al momento de *commitear*). En el lockeo pesimista puede haber *deadlocks*.

8.6.3 Two phase locking

Una transacción se dice que sigue **two phase locking** (o `2PL`) si todas las operaciones de *lock* preceden a la primer operación de *unlock*. Si todos los recursos a lockear tienen un orden (común a todas las transacciones) y todas las transacciones de una historia S piden los recursos en orden se puede demostrar que S es serializable y no hay *deadlock*.

Otra forma de evitar el *deadlock* con `2PL` es que durante la fase de obtención de locks, si una transacción no puede obtener un lock, libera todos los que posee y vuelve a intentarlo después de un tiempo. Sólo comienza la fase de procesamiento si puede obtener todos los locks.

9 Recuperabilidad

9.1 Sin Checkpoint

9.1.1 Undo

Regla: una transacción hace write. Ahí mismo el *transaction manager* puede, si quiere hacer los cambios a disco. Cuando la transacción hace commit, antes de escribir el commit en el log, todos los cambios que hizo la transacción tienen que haber sido bajadas a disco.

Problema: Estás coartándole las libertades al transaction manager porque todos los cambios tienen que haber sido mandados a disco antes que se escriba el commit de la transacción en el log.

9.1.2 Redo

Regla: una transacción hace write. El *transaction manager* no puede escribir nada a disco hasta que la transacción hace commit. Una vez que la transacción hace commit, se escribe inmediatamente el commit al log (y se *flushea*) y ahí el transaction manager puede ir bajando las cosas a disco cuando quiera.

Problema: necesita demasiado buffer. Y también acota al transaction manager porque lo obliga a esperar a mandar las cosas a disco hasta que después la transacción haga commit, ese commit se escribe en el log.

9.1.3 Undo/Redo

Problemas:

- El log ocupa más lugar.
- Requiere más trabajo ante un crash.

Ventaja: el transaction manager puede hacer lo que quieras.

9.2 Checkpoint

En un momento se dejan de aceptar transacciones. Se espera a que commiteen todas las transacciones activas en este momento. Se flushea todo a disco (o sólo las commiteadas en el caso del redo) y escribimos un "¡checkpoint¿" en el log y volvemos a aceptar transacciones. De este modo no tenemos que revisar todo el log hasta arriba de todo para restaurar.

Problema: cuando se quiere hacer el checkpoint hay que dejar de aceptar transacciones y eso puede ser inaceptable.

9.3 Checkpoint no quiescente

Se escribe un "¡start ckpt$(T_1, ...T_n)$¿" donde las transacciones $(T_1, ..., T_n)$ son las activas.

9.3.1 Undo

Al momento de escribir el end checkpoint, todas las transcciones activas tienen que haber commiteado (y, consecuentemente, sus cambios tienen que estar en disco por la regla de undo).

9.3.2 Redo

Al momento de escribir el end checkpoint, todas las transacciones que hicieron commit antes del start checkpoint ya tienen sus cambios grabados en disco.

9.3.3 Undo/Redo

Al momento de escribir el end checkpoint, absolutamente todo lo que pasó antes del start checkpoint ya tiene que estar en disco.

10 Seguridad

La **seguridad integrada** es la delegación de la autenticación a la base de datos al sistema operativo.

11 NoSQL

11.1 Teorema CAP

El teorema `CAP` establece que es imposible para un sistema de cómputo distribuido garantizar simultáneamente las siguientes tres propiedades (a lo sumo se pueden 2 de 3):

- **C**onsistency: que todos los nodos vean la misma información al mismo tiempo.
- **A**vailavility: la garantía de que cada petición a un nodo reciba una confirmación por si o por no (si fue completada la petición).
- **P**artition tolerance: que el sistema siga funcionando a pesar de algunas pérdidas de información o fallos parciales del sistema.

11.2 Big table

"Distributed multidimensional sorted map". Es la implementación de almacenamiento de Google, donde la mayor parte de las celdas están sin utilizar. Se distribuye en forma paralela, tiene geo-redundancia.

Además tiene una dimensión de *timestamp*. Mapea $(rowKey, columnKey, timeStamp)$ a datos arbitrarios.

11.3 Map reduce

11.4 Consistencias eventual

Se prefiere tiempo real por sobre consistencia. No hay garantía de que los datos que obtengas sean los últimos. La propiedad de **consistencia eventual** dice que, informalmente, "si nadie lo toca eventualmente va a ser consistente".

12 Data Mining

Es la etapa de análisis de *Knowledge Discovery in Databases* (`KDD`). Extraer patrones de los datos y generar modelos predictivos sobre minerías de datos.

12.1 Reglas de asociación

Una **regla de asociación** es cuando se determina *"siempre que pasa A, pasa B"*. Para medir cuán *buena* es una regla de asociación, se utilizan tres criterios:

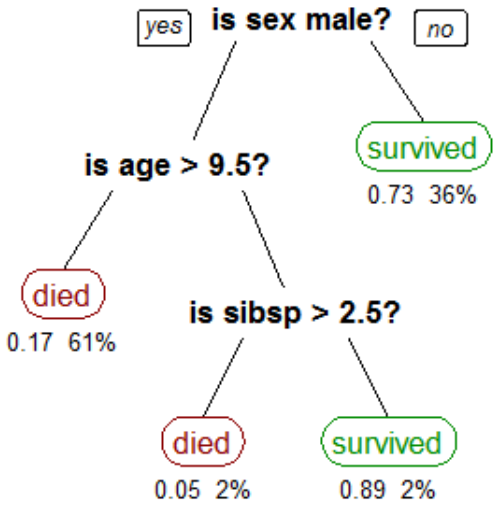
- **Soporte**: indica la **representabilidad** de la regla. Es la cantidad de veces que A y B sobre el total de transacciones.
- **Confianza**: indica cuánto trae A a B. Es la cantidad de veces que aparecen A y B sobre el total de veces que aparece A.
- **Lift**: indica el nivel de atracción de las variables. O sea compara la cantidad de veces que aparecen juntas contra la cantidad de veces que hubieran aparecido juntas si las probabilidades hubieran sido disjuntas.

12.2 Apriori

Apriori es un algoritmo que permite encontrar conjuntos de ítems frecuentes en una base de datos transaccional. Procede identificando los items frecuentes en los conjuntos y extendiéndolos a conjuntos más grandes mientras que superen el *treshold* seteado por el usuario.

Este algoritmo puede ser usado para determinar reglas de asociación de la base de datos, muy usando en el **market basket analysis**.

12.3 Árboles de clasificación



12.4 Market Basket Analysis

El **affinity analysis** es una técnica de análisis de datos y *data mining* que descubre reglas de co-ocurrencia entre actividades realizadas por individuos y grupos específicos. En general, se puede aplicar a cualquier proceso tal que los agentes puedan ser identificados y cuyas actividades puedan ser recopiladas. Un ejemplo es **market basket analysis** en el que los vendedores como Wallmart buscan entender el comportamiento de compra de sus compradores. Esta información puede ser usada con el propósito de *cross-selling* y *up-selling*, y afectar los descuentos y programas de beneficio al cliente.

12.4.1 Buisness intelligence

Buisness inteligence: Extraer info de los datos para mejorar la toma de decisiones.

12.4.2 Tipos de métodos

- Métodos supervizado: es una técnica para deducir una función a partir de datos de entrenamiento etiquetado (que se conoce el resultado deseado). El objetivo del aprendizaje supervisado es el de crear una función capaz de predecir el valor correspondiente a cualquier objeto de entrada válida después de haber visto una serie de ejemplos, los datos de entrenamiento. Hay dos ipos
  - Clasificación: tengo un serie de etiquetas que corresponden a clases y quiero inferir la función que me clasifican. (tengo un conjunto de células. Es o no un tumor?)
  - Regresión: tengo un conjunto de valores y una función desconocida y quiero inferir el valor que va a tener los valores. (tengo todos los precios de alquileres en baires en los ultimos 5 años. cuánto va a valer el mes que viene?)
- Métodos no supervizados: Encontrar patrones ocultos en datos no etiquetados. Ejemplos:
  - Market basket analysis.
  - Clustering (k-clustering, clustering jerárquico).

Overfitting: el algoritmo está sobreentrenado. Más que aprender a predecir la función, aprende a predecir el ruido.

13 Data Warehousing

Un **data warehouse** es una collección de datos que cumple las propiedades `INTS`:

- Integrated.
- Non-Volatile.
- Time variant.
- Subject oriented.

Para acordárselo uno puede usar la regla mnemotécnica: "**No Tv** =¿ **InSo**mnio".

Se utilizan para dar soporte a decisiones empresariales y de *buisness inteligence*. Proveen acceso a los datos para análisis complejo, descubrimiento de conocimiento y toma de decisiones. Dan soporte a demandas de alta performance en los datos de la organización.

A diferencia de las bases de datos tradicionales, los *data warehouses* típicamente soportan analisis temporal y de tendencia, que necesitan almacenar información histórica. Además, los cambios en los *data warehouses* suelen ser actualizados menos frecuentemente (no se considera vital que los cambios en el minimundo sean reflejados inmediatamente).

13.1 Dimensiones

Uno de los conceptos clave en los data warehouses es el de dimensión. Una dimensión provee estructura de etiquetado a la información que en otro caso serían medidas numéricas desordenadas. Una dimensión es un conjunto de datos individuales y disjuntos. Sus propósitos principales son:

- Filtrado.
- Agrupamiento.
- Etiquetado.

Informalmente, podemos ver a las dimensiones un data warehouse como los distintos ángulos sobre los que estudiamos la información almacenada en el mismo.

13.1.1 Modelos multidimensionales

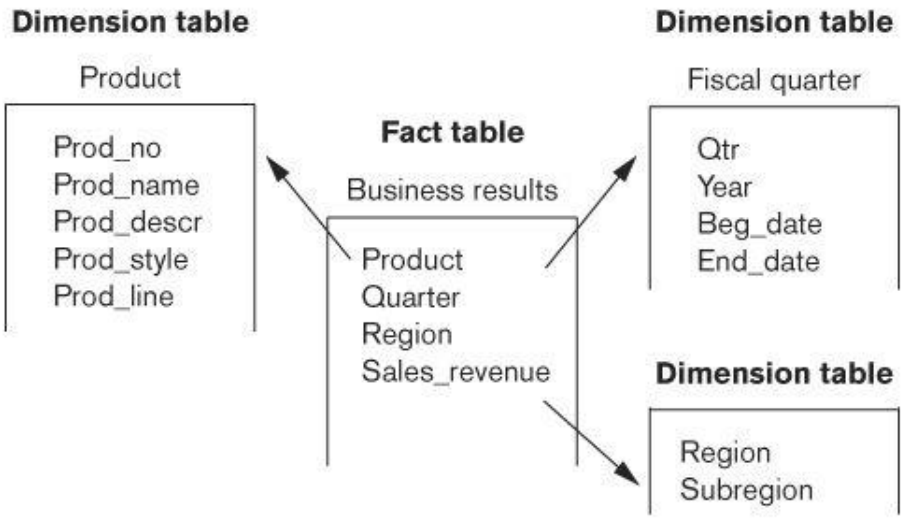
Los modelos multidimensionales permiten agruparse en vistas:

- **Roll-up**: agrupa operaciones en unidades más grandes en una dimensión.
- **Drill-down**: permite desagregar en unidades más pequeñas en una dimensión.

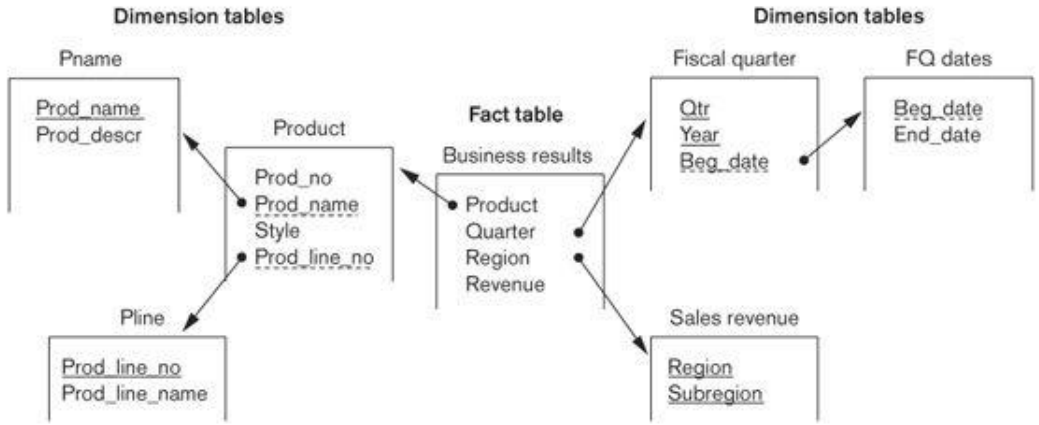
13.2 Esquemas multidimensionales

Los datawarehouses generalmente tienen dos posibles esquemas para sus dimensiones:

- **Star**: consiste en una tabla de hechos central y n tablas individuales para cada dimensión.



- **Snowflake**: nuevamente hay una tabla central de hechos, pero las dimensiones están organizadas en forma jerárquica.



14 Fuentes

- Fundamentals of Database Systems (6th Edition) - Elmasri & Navathe.
- Slides de la cátedra de Bases de Datos de Cecilia Ruz.
- Apuntes de clase de Julián Sackmann.