Resumen Final

Bases de datos

(basado en resúmenes de CubaWiki, diapositivas y apuntes del 2do cuatrimestre de 2019,
actualizado en 2022 por otro estudiante, Leandro N. Borgnia
(este resumen no es mío pero me pareció muy completo y bien sintetizado y quería mejorarlo.))

6 de marzo de 2020

Introducción

Definiciones

- **Base de datos:** Es un conjunto de datos relacionados con un **significado inherente**. Esto hace que no consideremos a un conjunto de datos aleatorios como una, ya que se diseña y construye con un propósito específico. Surgieron en la década del '70 debido a las crecientes necesidades de almacenar y acceder a mayores cantidades de datos de forma eficiente. Suelen estar manejadas por un **DBMS** (Database Management System).
- **Dato:** Es un hecho conocido que puede ser registrado y tiene un significado implícito. Suele ser más volatil que la estructura en que está definido.
- **DML (Data Manipulation Language):** Es el usado en los DBMS para modificar las instancias: obtener, agregar, cambiar, etc. Algunas de sus funciones en SQL son SELECT, INSERT, DELETE.
- **DDL (Data Definition Language):** Se usa en los DBMS para alterar el esquema de la base: crear, agregar atributo, renombrar. Algunas de sus funciones en SQL son CREATE, ALTER, DROP, RENAME.

DBMS

Es la herramienta que utiliza cada aplicación para manejar grandes cantidades de datos de manera eficiente. Suele ser configurada por los **administradores de bases de datos**. Entre sus funciones tenemos:

1. Permitirle a los usuarios crear nuevas bases de datos y especificar sus esquemas.
2. Perimitirle a los usuarios realizar consultas a los datos y modificarlos.
3. Almacenar grandes cantidades de datos por un período largo de tiempo para perimitir consultas y modificaciones eficientes.
4. Garantizar la durabilidad de los datos al tener un sistema de recuperación en caso de fallas, errores o mal uso intencional. Este también se encarga de las copias de seguridad.
5. Controlar el acceso concurrente a los datos por parte de los usuarios, de manera de evitar interacciones inesperadas y garantizar que las acciones realizadas sean seguras y completas.

Otros componentes de la DBMS incluyen:

- **Recovery Manager:** Encargado de restaurar la base de datos a un estado consistente en caso de haber una falla. Para hacer eso hace uso del **log**, un archivo que lleva un registro de las acciones efectuadas a la DB.
- **Optimizador de consultas:** Encargado de armar un plan de ejecución eficiente en base a una consulta, basándose en la información del **system catalog**.

System catalog

Es el lugar de un **RDBMS** (Relational DBMS) en que se guardan los metadatos del esquema. Estos comprenden:

- Información sobre tablas y columnas. Esto incluye datos estadísticos como el tamaño de los archivos y el factor de bloqueo, la cantidad de tuplas de la relación, la cantidad de bloques de la relación y el rango de valores de una columna.
- Vistas, interpretadas como consultas que pueden guardarse para referenciarse nuevamente.

- Índices, utilizados para optimizar las consultas.
- Usuarios y grupos de usuarios, para controlar los accesos.
- Triggers, para actualizar automáticamente ciertos datos en respuesta a eventos o acciones.
- Funciones de agregación definidas por el usuario.

Entre sus principales usos tenemos:

I Obtener el esquema de una tabla al verificar una consulta.

II Obtener la selectividad esperada de un atributo al optimizar una consulta.

III Obtener los permisos de un usuario al verificar un acceso.

Arquitectura e independencia

En una base de datos tenemos 3 **niveles**:

- **Interno (o física):** Es el que describe el almacenamiento físico de las estructuras de la base de datos.
- **Conceptual (o lógico):** Es el que contiene el esquema conceptual que describe la estructura de la base de datos sin enfocarse en lo físico sino en entidades, tipos de datos, operaciones de usuarios y restricciones.
- **Externo (o de usuario):** Contiene los esquemas o vistas de usuarios en los que se describe la parte de la base de datos que le interese a un grupo en particular.

Entre ellos podemos tener las siguientes **independencias**:

- **Lógica:** Capacidad de poder cambiar el esquema conceptual sin cambiar los externos, ya sea para expandir o reducir la base de datos (agregar o quitar un nuevo tipo de registro), cambiar las restricciones, etc. No suele ser fácil de lograr. Esto implica que la capa Lógica o Conceptual sea independiente de la de usuario o Externa, y por tanto el usuario no necesite entender la arquitectura lógica de los datos para usar la Base de Datos.
- **Física:** Capacidad de poder cambiar el esquema interno sin cambiar el conceptual (y por ende el externo). Esto involucra cambiar la organización de los archivos o agregar algún índice para las consultas. En este caso estamos hablando de la independencia de la capa física o interna comparado con la capa conceptual o lógica. Esto permite que entender y modificar la Base de Datos en términos lógicos o conceptuales de sus estructuras, no requiera entender su almacenamiento físico.

Modelización

Los pasos para diseñar una base de datos relacional son:

$$\text{Requerimientos} \leftrightarrow \text{MER} \leftrightarrow \text{MR} \leftrightarrow \text{Normalización} \leftrightarrow \text{Diseño físico} \leftrightarrow \text{BD}$$

Se basan en la metodología de diseño lógico para bases de datos (LRDM).
MC (Modelo Conceptual): Es una concpetualización formal del mundo real (dominio específico) que modela sus objetos, características y relaciones. Se usan para comunicar ideas, buscar concensos y tienen gran importancia a la hora de analizar aplicaciones y validar acciones de los usuarios. Se representan a través de lenguajes de ontologías, generalmente expresados a través de diagramas (DERs, UML).

MER (Modelo Entidad-Relación)

Es una herramienta que permite realizar una abstracción o modelo de alguna situación de interés del mundo real. Se realiza a través de la técnica de los **DERs** (Diagramas de Entidad-Relación) conformados por los siguientes elementos.

Entidades

Objeto o concepto del que queremos registrar información en un contexto dado. Se representan a través de bloques y conforman conjuntos de instancias. Pueden ser:

- **Fuertes:** Su existencia no depende de la de otra entidad y se identifican a través de atributos propios.
- **Débiles:** Necesitan de la identificación de otra entidad para distinguirse.

Además cuando tenemos varias que comparten atributos podemos generalizarlas de forma parcial o total, tratando a cada una como una **especialización**. Estas heredan atributos de la generalización y pueden ser disjuntas (con atributo discriminante) o solapadas.

Atributos

Son las características descriptivas de las entidades relevantes al problema. Con ello constituyen la información concreta a mantener de cada elemento de una entidad. Tienen un respectivo dominio y pueden identificar a la instancia. Según cómo lo hagan tenemos:

- **Superclaves (SK):** Conjunto de atributos que identifican unívocamente las instancias.
- **Claves candidatas (CK):** SK minimales.
- **Clave primaria (PK):** Una CK elegida según un criterio para identificar a la entidad.

Además pueden ser **multivaluados** (más de un valor a la vez) y **compuestos** (se dividen en subpartes con significados diferentes), los cuales se usan poco en la realidad.

Interrelaciones

Constituyen la manera de vincular entidades en el dominio del problema. Su **cardinalidad** determina la cantidad de instancias participantes de cada entidad (1-1, 1-N, N-M) y su **grado** la cantidad de entidades (unarias, binarias y ternarias). Cada extremo de la relación se corresponde con un **rol**. Además la **participación** de las entidades puede ser total o parcial según si todas sus instancias deben formar parte de ella. Se almacenan como n-uplas ordenadas.
Cuando se desea interrelacionar una relación N-M con una entidad se puede usar una **agregación** que abstraiga a la primera como una entidad.

Reglas de dominio

Son restricciones adicionales destinadas a expresar limitaciones sobre los datos que no pueden expresarse de otra manera. Pueden escribirse a través de lenguajes formales o informales con tal de salvar la pérdida de información.

Trampas de conexión

En caso de haber una mala interpretación de las interrelaciones que genere que se pierda información y no se tenga una representación adecuada del mundo real decimos que tenemos una **trampa de conexión**. Entre ellas tenemos:

- **Trampa del abanico (Fan Traps):** El camino entre ciertas entidades es ambiguo. Sucede generalmente cuando salen dos o más interrealciones 1:N en abanico desde la misma entidad.
- **Trampa del sumidero (Chasm Traps):** La interrelación existente entre dos entidades del modelo no tiene camino. Suele aparecer cuando una entidad participa parcialmente de dos o más relaciones.

Para solucionarlas suele reestructurarse el modelo.

Otras consideraciones sobre DERs

- La cardinalidad de las interrelaciones ternarias se basa en ver la cantidad de elementos de una entidad que se relacionan con un par de las otras.
- Las interrelaciones unarias deben tener al menos un extremo de participación parcial y sus roles definidos.
- No suelen modelarse entidades de un sólo registro.
- Las interrelaciones N-M pueden tener atributos identificatorios para tener mas de un vínculo entre instancias con distinto atributo (registro, historial).
- Según el contexto pueden reemplazarse interrelaciones ternarias por pares de binarias o agregaciones.
- Para evitar proliferación de roles en las especializaciones se puede tener una entidad aparte.

MR (Modelo Relacional)

Es la herramienta que nos permite expresar el **esquema lógico** de la base de datos. Se basa en un conjunto de **relaciones** que pueden pensarse como **tablas** con columnas y filas. Las columnas se corresponden con atributos y su conjunto conforma el esquema de la tabla, tomando el dominio de cada uno. Por otra parte las filas conforman las instancias en base a un conjunto de valores (registros).
Cada entidad se verá representada por una relación y para representar las interrelaciones usamos el concepto de **clave foránea (FK)** correspondiente a la PK de de otra entidad. Sobre esto, tenemos las **restricciones adicionales de referencia e integridad** para que las relaciones representen de manera consistente al esquema lógico.

Lenguajes de consulta

Álgebra relacional (AR)

Es un lenguaje formal especificado en base a propiedades matemáticas que en el modelo relacional le permite a los usuarios especificar consultas sobre instancias de relaciones, dando como resultado una nueva relación. Permite así formalizar las operaciones asociadas al modelo relacional, da una base para implementar y optimizar consultas en RDBMS y constituye los módulos internos de las principales operaciones y funciones de muchos sistemas relacionales. Es **procedural y axiomático** con operaciones unarias y binarias.

Operaciones unarias

- **Select(σ):** Selecciona un subconjunto de tuplas de una relación en base a cierta condición. Genera una **partición horizontal**, es conmutativa, se puede agrupar en cascada y conserva la cantidad de atributos. En SQL:

  `SELECT * FROM` *Relacion* `WHERE` *Condicion*

- **Project(π):** Selecciona un subconjunto de columnas de una relación. Genera una **partición vertical**, no es conmutativa y la cantidad de tuplas se preserva si la proyección contiene alguna superclave. En SQL:

  `SELECT DISTINCT` *Columnas* `FROM` *Relacion*

- **Rename(ρ):** Le asigna un nuevo nombre a los atributos y/o a la relación. Suele usarse para resultados intermedios. En SQL:

  `SELECT` *NRelacion.Columna* `AS` *NColumna* `FROM` *Relacion* `AS` *NRelacion*

Operaciones binarias (entre R y S)

- **Union/Intersection/Minus:** Se corresponden respectivamente con las operaciones matemáticas de conjuntos $\cap$, $\cup$ y $\setminus$. Su relación resultante no contiene duplicados y requiere que R y S sean **union compatibles**, es decir, que contengan la misma cantidad de atributos y sus tipos de datos coincidan en orden. La relación resultante contendrá los nombres de R. En SQL (se puede agregar `ALL` para no eliminar duplicados):

  `SELECT` *Columnas* `FROM` *R* `UNION/INTERSECT/MINUS SELECT` *Columnas* `FROM` *S*

- **Producto Cartesiano:** Crea una nueva relación que combina cada tupla de R con una de S. Su grado es la suma de los grados de sus relaciones y no requiere que las relaciones sean union compatibles. En SQL:

  `SELECT * FROM` *R* `CROSS JOIN` *S*

- **Join:** Combina pares de tuplas relacionadas entre R y S en base a cierta condición. Esto no incluye a las tuplas NULL ni a las que no cumplan con la condición (que debe tener cierto formato para ser válida). Es conmutativa, su grado es la suma de los de sus relaciones y la cantidad de tuplas resultantes depende de la condición. Si la condición es de igualdad se denota **EquiJoin**. En SQL se usa con las cláusulas `SELECT`, `FROM` y `WHERE`.

- **Natural Join:** Similar a Join pero relacionando los campos del mismo nombre, dejando sólo uno de los duplicados. Para ello requiere de la correspondencia de nombres o que se haga un renombre previo.

- **Outer Join:** A diferencia de los anteriores clasificados como **Inner Join**, aquí se pueden incorporar al resultado las tuplas de R, S o ambas que no cumplan con la condición. Respectivamente son de tipo **Left**, **Right** y **Full**, y los atributos restantes de la tupla en el resultado se rellenan con NULL. Estas operaciones son parte del estándar SQL2.

- **Division:** Retorna los valores de R emparejados con todos los valores de S, para lo cual requiere que los de S estén en los de R. El resultado contiene los atributos en R que no están en S. Si bien no suele implementarse en SQL puede expresarse en base a otras operaciones.

CRT (Cálculo Relacional de Tuplas)

Es otro lenguaje de consultas asociado al MR. Emplea una **técnica declarativa** de consultas (no describe un orden de evaluación) con fundamentos basados en la lógica matemática y forma las bases fundacionales de SQL. Se basa en la expresión:

$$\{\texttt{t} \mid \texttt{COND(t)}\}$$

donde `t` es una tupla y la única variable de la expresión, y `COND` es una fórmula bien formada de CRT. Como resultado se obtiene un conjunto de todas las tuplas de CRT que verifican la condición.

Expresividad

La **expresividad** de un lenguaje nos determina la amplitud de las ideas que pueden ser representadas y comunicadas en él. En los lenguajes de consulta equivale al conjunto de consultas que se pueden expresar usándolos.

Expresividad de CRT

CRT es una especialización particular de la **LPO** (Lógica de Primer Orden) adaptada a bases de datos, donde a cada instancia se le asocia un valor de verdad según si cumple con el predicado de la expresión de la consulta. En su semántica sólo buscamos saber la validez de la fórmula en la DB.
Dada una expresión, su **dominio** es el conjunto de valores que aparecen en ella como constantes o existen en cualquiera de las tuplas de las relaciones a las que hace referencia. Decimos que es **segura** si garantiza producir una cantidad finita de tuplas como resultado. Ejemplo de expresión insegura:

$$\{t \mid \neg(t \in \texttt{EMPLEADO})\}$$

Alternativamente, en una expresión segura todos los valores en el resultado son parte del dominio de la expresión.

Proposición: CRT restringido a expresiones seguras tiene el mismo poder de expresividad que AR.

DRC (Cálculo Relacional de Dominio): A diferencia de CRT usa atributos en lugar de variables. Tiene el mismo poder expresivo.

Límites de la expresividad

Hay consultas sobre las DB que no son expresables a través de AR ni CRT (incluso LPO). Podemos demostrar si una lo es o no a través de ciertas herramientas matemáticas. Para LPO tenemos el **teorema de Ehrenfeucht-Fraisse**.

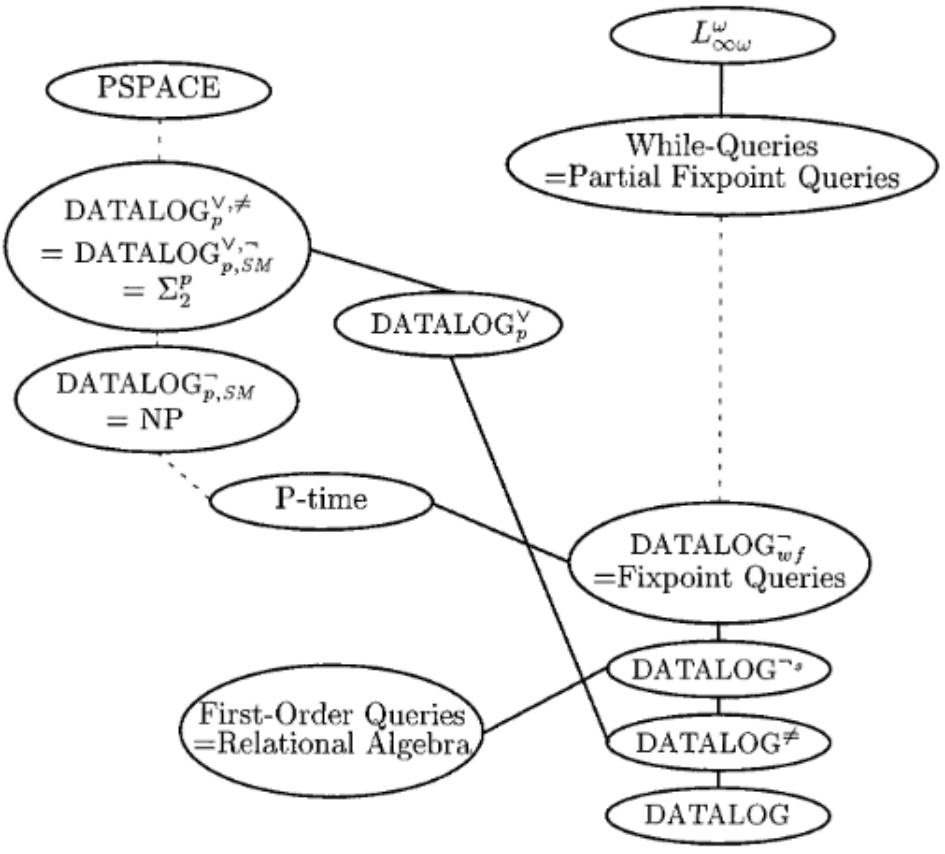


Fig. 2. Expressive powers of disjunctive Datalog and other query languages.

Teorema de Ehrenfeucht-Fraisse

Es una demostración basada en Teoría de Juegos que se usa para determinar si dos estructuras son **elementalmente equivalentes** (isomorfas). Cuenta con dos jugadores: Spoiler y Duplicator, y dos estructuras (grafos): A y B. Se juega por rondas y en cada una Spoiler elige un nodo de una estructura y en respuesta Duplicator debe elegir uno de la otra. Spoiler busca demostrar que las estructuras son **distinguibles**, mientras que Duplicator busca lo contrario. Dadas

n rondas los grafos son **indistinguibles** si a partir de los nodos elegidos de cada estructura se tiene un isomorfismo parcial (se mantiene la igualdad y adyacencia). Si para n rondas Duplicator tiene estrategia ganadora decimos que:

$$A \sim_n B$$

Ahora, las estrategias se pueden expresar a través de fórmulas de LPO donde la cantidad de rondas se corresponde con la cantidad de cuantificadores. Por ende, si $A \sim_n B$ luego A y B cumplen las mismas sentencias con n cuantificadores. Generalizado a la LPO:

Teorema: *Una propiedad P no es expresable en la LPO si para todo n natural se pueden hallar dos grafos A y B tales que P sea falsa en A, verdadera en B y* $A \sim_n B$.

Hay predicados "extra-lógicos" en SQL que permiten expandir su poder expresivo: recursión, funciones de agregación y agregamiento, operaciones aritméticas sobre atributos numéricos, store procedures.

Normalización

Al diseñar bases de datos la normalización surge como una técnica para tener una **medida de calidad** sobre sus esquemas (aunque pueda variar según el contexto de uso). Se toma en un nivel conceptual (significado de las relaciones) y uno implementativo (cómo se almacenan físicamente las tuplas). Primordialmente, busca **preservar la información** y **minimizar la información almacenada de forma redundante**.

Pautas de diseño

El proceso de normalización se basa en 4 pautas de diseño que pueden usarse como guía sin necesariamente determinarlo (no siempre son independientes entre sí).

- **Semántica:** Diseñar los esquemas de manera que sea intuitivo entender el significado de sus atributos para evitar confusiones y malos usos. Esto implica no combinar atributos de diversos tipos de entidades y relaciones en la misma relación.
- **Almacenamiento:** Minimizar el espacio ocupado por el diseño evitando anomalías de actualización. Entre ellas tenemos:
  - **Inserción:** La incorporación de una nueva instancia de una entidad está ligada a la existencia de otra o puede producir inconsistencias con datos anteriormente cargados en la relación.
  - **Deleción:** El borrado de una instancia produce que se pierda toda la información de otra.
  - **Modificación:** Modificar una instancia de una entidad puede volver sus datos inconsistentes con los de las apariciones de una entidad en la relación.

  Al permitir alguna de estas anomalías es preferente que estén debidamente indicadas para su correcta operación. Una razón por la que se suele violar esta pauta es la mejora en el **rendimiento** subyacente dada por el caso de uso (por la frecuencia de las consultas y actualizaciones puede convenir tener un campo de más pero de rápido acceso).
- **NULL:** Evitar la presencia de valores nulos en las relaciones debido a su variedad de interpretaciones posibles (valor no aplicable, desconocido, concido y ausente). Cuando sean inevitables deben ser la excepción.
- **Tuplas espúreas:** No crear descomposiciones en las que al reconstruir la información usando juntas esta no se corresponda con la original. Esto suele ocurrir cuando en una descomposición no se mantienen las claves de las entidades de la relación. Puede verificarse a través de consultas que usen ambas entidades.

Dependencias funcionales (DF)

Son herramientas formales utilizadas para analizar esquemas que ayuden a detectar algunos de los problemas mencionados. Representadas de la forma $X \to Y$ donde X e Y son subconjuntos de atributos de la relación, nos dicen que los valores de Y dependen de los de X, imponiendo así una **restricción** sobre las posibles tuplas que se pueden formar.
Se especifican según la semántica de los atributos de la relación. Con ello, una **instancia legal** es aquella que las respeta (también llamada extensión o estado legal). Nótese que sólo con los datos no es posible determinarlas, aunque nos permiten confirmar su existencia o descartarlas.

Proceso de normalización

Partiendo de las DFs y las claves de cada relación de una DB, podemos aplicarle una serie de tests para certificar si satisface una **forma normal**. De no hacerlo, se le puede aplicar un **proceso de normalización** enfocado en minimizar la redundancia y las anomalías de inserción, deleción y modificación para que en una nueva descomposición

la DB pase estos tests. Nótese que las formas normales no garantizan un buen diseño de la DB.
Tras efectuar este proceso se buscan:

- **Nonadditive Join (SPI):** Garantía de que no se produzcan tuplas espúreas, es decir, se puede recuperar la información original.
- **Preservación de DF (SPDF):** Cada DF estará representada en algún esquema resultante de la descomposición.

Lo primero se busca a cualquier costo, mientras que lo segundo es a veces sacrificado.
Una DF $X \to Y$ es **completa** si al eliminar un atributo de X la dependencia deja de existir. Es **parcial** si sigue existiendo. Por otra parte, la **cobertura minimal** F_M de un conjunto de dependencias F es tal que el lado derecho de cada dependencia tiene un sólo atributo y las dependencias son completas. Para formarla debemos descomponer los atributos a derecha, quitar los redundantes a izquierda y eliminar las DFs redundantes (no es conmutativo el proceso).

Formas normales basadas en Clave Primaria

- **1FN:** No deben haber relaciones dentro de relaciones o como atributos dentro de las tuplas. Esto implica que el dominio de los atributos sólo puede ser de valores atómicos (o sea, no pueden ser **multivaluados**). Para alcanzar esta forma normal suele tomarse el atributo en una nueva relación que tenga como PK a ambos atributos.
  Otras opciones involucran expandir la PK (introduciendo posible redundancia) o dividir el atributo en la cantidad de valores que puede tomar (derivando en posibles valores nulos). A través de la aplicación recursiva de 1FN se pueden sacar las **relaciones anidadas** de una relación.
- **2FN:** Todos los atributos no primos deben depender completamente de la PK de la relación. Para pasar a 2FN basta con descomponer en subesquemas en los que aparezcan los atributos de la PK de los que dependen como parte de su clave.
- **3FN:** Una DF $X \to Y$ es transitiva si tenemos un conjunto de atributos no primos Z tal que $X \to Z$ y $Z \to Y$. Luego, en esta FN ningún atributo no primo debe depender transitivamente de la PK. Para satisfacerla garantizando SPI e SPDF tomamos la cobertura minimal F_M de la relación, formamos los subesquemas XA de cada DF $X \to A$, unimos los de igual lado izquierdo, agregamos uno con los atributos de una clave si ninguno de los resultantes contiene alguna y eliminamos los esquemas redundantes.
- **BCFN (Boyce-Codd Normal Form):** Toda dependencia funcional no trivial $X \to A$ de la relación debe tener a X como SK. Es más restrictiva que 3FN y al pasar a esta pueden perderse DFs ya que su proceso se basa en crear subesquemas reemplazando las DFs que violan esta FN.

Inferencia

Una DF es **inferida** por un conjunto de DFs si se cumple para toda instancia legal de la relación que cumpla con este cojunto. Con esto, la **clausura** F^+ de un conjunto de DFs F son todas las dependencias funcionales que pueden ser inferidas a partir de este. A partir de esta podemos hallar una CK X si $X \to A$ pertenece y para ningún subconjunto Y de X, $Y \to A$ peretenece (X entonces determina funcionalmente a todos los atributos de la relación). Podemos calcular F^+ aplicando las **reglas de inferencia** (o "axiomas de Armstrong"):

- **RI1 (regla reflexiva):** Si X contiene a Y entonces $X \to Y$.
- **RI2 (regla de incremento):** $X \to Y$ implica $XZ \to YZ$.
- **RI3 (regla transitiva):** $X \to Y$ e $Y \to Z$ implican $X \to Z$.

Estas crean una clausura **fiable** (toda DF implicada a partir de F es inferida) y **completa** (F^+ es determinada completamente a partir de F aplicando estos axiomas). Entre sus corolarios tenemos:

- **RI4 (regla de descomposición o proyección):** $X \to YZ$ implica $X \to Y$.
- **RI5 (regla de unión o aditiva):** $X \to Y$ e $X \to Z$ implican $X \to YZ$.
- **RI6 (regla pseudotransitiva):** $X \to Y$ e $WY \to Z$ implican $WX \to Z$.

Transacciones

Son conjuntos de instrucciones que al ejecutarse forman una **unidad lógica de procesamiento**. Pueden tener uno o más accesos a la DB a través de diversas operaciones: inserción, eliminación, modificación, etc.
Las transacciones pueden ser **read-write** ó **read-only** según si actualizan o no los ítems de la DB. Los ítems leidos conforman su **read-set** y los actualizados su **write-set**. El tamaño de cada uno viene dado por su **granularidad** (registro, bloque de disco, valor) y esta determina su identificador (número, dirección física).
Entre sus operaciones tenemos:

- **read_item(X):** Se toma la dirección del bloque de disco del ítem X, se copia su contenido al buffer de memoria principal (de no haberlo hecho) y finalmente se copia el ítem X del buffer a la variable X del programa.
- **write_item(X):** Similar al anterior con la diferencia de que se copia el contenido de la variable X del programa al ítem X del buffer y este es luego almacenado en memoria, según la **política** de copia de datos a disco (depende de la caché, SO y recovery manager).

Propiedades ACID

En los sistemas de control de concurrencia y recovery procuramos que se cumplan una serie de propiedades:

- **Atomicity:** Las transacciones son unidades atómicas, por lo que se ejecutan en su completitud o no se ejecutan. Depende del subsistema de recovery del DBMS.
- **Consistency preservation:** La transacción al ejecutarse lleva la DB de un estado consistente a otro. Depende de que los programadores cumplan con las restricciones de integridad impuestas, ya que la DB no tiene manera de verificarlo automáticamente.
- **Isolation:** La ejecución de una transacción no debe interferir con la de otra, es decir, debe simular ser aislada incluso al ejecutarse a la vez. Depende del subsistema de control de concurrencia y puede ser de varios **niveles**: 0 (sin sobreescribir dirty reads), 1 (sin lost updates), 2 (sin lost updates ni dirty reads) o 3 (nivel 2 sin repeatable reads).
- **Durability:** Los cambios aplicados a una DB por una transacción commiteada deben perdurar ante fallos. Depende del subsistema de recovery del DBMS.

Subsistema de manejo de transacciones

Este subsistema del DBMS se conforma de los siguientes módulos:

- **Transaction manager (TM):** Encargado de preprocesar y ejecutar las transacciones en coordinación con el resto. Dentro de ellas, las operaciones tienen un orden fijo y pueden terminar en commit o abort.
- **Scheduler (planificador):** Controla el orden de ejecución de las transacciones en base a una serie de reglas definidas según el paradigma.
- **Recovery manager (RM o log manager):** Encargado de asegurar la durabilidad y atomicidad de las transacciones. Realiza las siguientes operaciones:
  - **RM-Read(T_i,x):** lee x para la transacción T_i.
  - **RM-Write(T_i, x, v):** escribe v en x para la transacción T_i
  - **Commit(T_i):** Commitea T_i según la política de recovery.
  - **Abort(T_i):** Aborta T_i según la política de recovery.
  - **Restart:** Lleva a la base de datos al último estado commiteado según el log y la política de recovery.
- **Buffer manager (BM):**, tiene un protocolo de operaciones que le dan al Recovery Manager soporte para acceder y modificar la Base de Datos, siendo dichas operaciones:
  - **INPUT(X):** Copia el bloque de disco del ítem X a un buffer de memoria.
  - **READ(X, t):** Copia el contenido del ítem X del buffer de memoria a la variable temporal t en memoria local.
  - **WRITE(X, t):** Copia el valor de la variable local t al ítem X en un buffer de memoria.
  - **OUTPUT(X):** Copia el bloque conteniendo al ítem X del buffer de memoria a disco.
  - **FLUSH LOG:** Ordena al buffer manager escribir los registros del log a disco.
- **Cache manager (CM):** Encargado de manejar las transferencias entre el disco y el almancenamiento no volatil. Entre sus operaciones tenemos:
  - **Fetch(X):** Toma un slot vacío c de la caché, copia en este el valor del ítem X de disco, inicializa su dirty bit indicando que puede ser modificado con respecto a disco y actualiza el directorio de esta memoria para señalizar que este slot es ahora ocupado por el ítem X.
  - **Flush(X):** Escribe el contenido del ítem X de caché a disco. Se puede efectuar automáticamente si al hacer Fetch(Y) no hay espacio para el ítem Y en caché.
  - Pin y Unpin

- **Data manager (DM):** Es una entidad que engloba al **Recovery manager**, **Cache manager**, a la **Base de Datos** (no al DBMS, sino a la estructura donde están guardados los datos) y la **Cache** (como estructura, no como manager).

Control de concurrencia

Los sistemas que implementan transacciones suelen requerir de respuesta rápida y alta disponibilidad por ser multi-usuario con usuarios concurrentes. Algunos ejemplos son los sistemas de reserva de vuelos, transferencias bancarias, procesamiento de tarjetas de crédito y manejo de stock (otros ejemplos: sistemas de reserva de aerolíneas, cajeros electrónicos y supermercados).
En la **multiprogramación** dos o más procesos se ejecutan concurrentemente y si no lo hacen de forma paralela su ejecución es **intercalada** en una CPU. Aquí hay que atender a una serie de posibles problemas (tomamos T_1 y T_2 transacciones con $T_1 < T_2$):

- **Incorrect Summary Problem:** T_1 modifica X y Y, restándole N a X y sumándoselo a Y. T_2 lee X después de su modificación y Y antes de dicha modificación. Si T_2 retorna la suma de ambos, va a devolver N menos de lo que debería porque no llego a ver la suma de Y, resultando en una sumarización incorrecta.
- **Lost Update Problem:** T_2 lee el valor de un ítem X previo a que T_1 lo modifique o lo sobrescribe, y T_2 sobrescribe dicho valor sin haber visualizado la escritura final de T_1, faltando una actualización en X.
- **Temporary Update Problem (Dirty Read):** T_2 lee el valor de un ítem X de una T_1 que fue abortada.
- **(Non/Un)repeatable Read Problem:** T_2 lee el mismo ítem X dos veces y entre ambas lecturas es modificado por T_1, dando lugar a diferentes valores.
- **Phantom Read Problem:** T_2 realiza la misma query dos veces y entre ambas ejecuciones el conjunto de tuplas es distinto. (nótese que la diferencia con Unrepeatable reads es que no se modifica un valor, sino que aparecen nuevos valores o tuplas).

Incorrect Summary y **Lost Update** tienden a ser resueltos por cualquier nivel de aislamiento. Para que no se produzcan **dirty reads**, se debe exigir **"Read Commited"**, para evitar **Unrepeatables reads** se exige **"Repeatable Read"**, y finalmente para que no sucedan **Phantom Reads** se exige **"Serializable"** como nivel de aislamiento.

Historia

Una **historia** (plan) determina la forma en que se ejecuta un conjunto de transacciones restringiendo que las operaciones sigan su orden si pertenecen a la misma transacción. De allí decimos que que dos operaciones entran **en conflicto** cuando son de transacciones diferentes, operan sobre el mismo ítem y al menos una es una escritura (no pueden ejecutarse en simultáneo).
Además, dadas dos transacciones T_1 y T_2 decimos que T_1 **lee** X **de** T_2 si T_1 escribe sobre X y T_2 es la última transacción no abortada que había escrito sobre X.

Definiciones

- **Equivalencia:** Dos historias son **equivalentes** si se definen en base al mismo conjunto de transacciones y el orden de las operaciones en conflicto de las transacciones no abortadas es el mismo.
- **Seriabilidad (SR):** Una historia es **seriablizable** si es equivalente a una serial, lo cual permite satisfacer la propiedad de aislamiento. Podemos verificar si lo es a través de la construcción de un **grafo de precedencia** (SG(H)), donde cada nodo es una transacción y los arcos dirigidos de T_i a T_j se dan si hay operaciones en conflicto entre ellas y T_i sucede antes que T_j. Si el grafo es acíclico, la historia es serializable.
- **Recuperabilidad:** Según las restricciones impuestas, tenemos diferentes niveles:
  - **Recuperable (RC):** Se da si para todo T_i, T_j tales que T_i lee de T_j y T_i hace commit entonces $c_i > c_j$.
  - **Avoids Cascading Aborts (ACA):** Se da si toda transacción lee de aquellas que hayan hecho commit. De esa forma se pueden evitar los dirty reads sin abortar como en RC.
  - **Strict (ST):** Toda transacción lee o escribe en un ítem si la que anteriormente lo había escrito terminó (commit o abort).

  En base a ello: Seriales $\subset$ ST $\subset$ ACA $\subset$ RC.

Paradigmas de planificación

Atendiendo a los problemas anteriormente mencionados de un sistema concurrente, los schedulers pueden optar al recibir una operación por rechazarla, demorarla o procesarla. Aquellos que favorecen su procesamiento se denotan **agresivos** (mayoría), y si no, **conservadores** (minoría). Cómo actúen depende de cómo traten a los data ítems: si

los bloquean para su uso exclusivo siguen un **paradigma pesisimista**, mientras que si asumen un comportamiento serializable y actúan en caso de no suceder, **paradigma optimista**.

Paradigma pesimista - Locking

Para saber si un ítem está disponible para ser usado podemos usar el concepto de **lock**, que se representa como una variable asociada a este. En su versión **binaria** un ítem puede bloquearse o desbloquearse a través de las operaciones de lock y unlock. Luego, si una transacción desea operar sobre un ítem debe solicitar un lock sobre este y esperar a que se lo otorguen. Esto hace que sólo una transacción pueda operar sobre el ítem a la vez, incluso si las operaciones son lecturas.
Para salvar esta limitación se tiene el **shared lock (ternario)**, que permite que el lock sea **exclusivo** (write lock) o **compartido** (read lock), con sus respectivas operaciones. De ahí, si una transacción desea que pase de uno a otro deberá efectuar una operación de "upgrade".
Luego, una transacción puede leer un ítem X si solicitó un lock sobre este y no lo ha liberado, y ninguna otra tiene un write lock a la vez. Para la escritura se requiere haber solicitado y no liberado un write lock y que ninguna otra tenga un lock sobre X o haya pedido un upgrade.

Two Phase Locking (2PL)

Decimos que una transacción cumple con el mecanismo **2PL** si toda operación de lock precede a toda operación de unlock (fases de crecimiento y decrecimiento). Si en una historia legal todas sus transacciones lo cumplen podemos demostrar que la historia es **serializable** (el algoritmo es correcto).
Decimos que una transacción es **2PL Estricta (2PLE)** si es 2PL y no libera ninguno de sus locks de escritura hasta después de hacer commit o abort. Esto garantiza que una historia sea **ST** pero no libre de deadlocks. Por otra parte, que sea **2PL Rigurosa (2PLR)** implica que tampoco los de lectura sean liberados antes.

Deadlock

Es la situación en las historias en que en un grupo de transacciones cada una está esperando a que la otra libere un lock y no puede avanzar. Se pueden prevenir o evitar según la estrategia empleada:

- Para **evitarlos** podemos detectar si existe a través de un **grafo de espera** (Wait For Graph, WFG(H)) donde para cada transacción en un nodo creamos un arco dirigido a otro si debe esperar a que se libere un lock de su transacción. Cuando su último lock de espera se libera, el arco también.
  Si el grafo es cíclico hay deadlocks. De ahí se elige a una **transacción víctima** siguiendo algún criterio (tiempo de ejecución, variables actualizadas o por actualizar, ciclos en los que aparece, etc.) y se la aborta para reiniciarla posteriormente.
- La otra opción es **prevenirlos** implementando **timestamps (TS)**, que se le asignan a cada transacción según el momento en que comienzan. Esto hace que si una transacción T_i desea solicitar un lock sobre un ítem bloqueado por T_j se tengan dos estrategias:
  - **Wait-Die:** Si TS(T_i) < TS(T_j) se pone a T_i en espera y si no se aborta a T_i para reiniciarla más tarde con el mismo TS.
  - **Wound-Wait:** Si TS(T_i) < TS(T_j) se aborta a T_j y si no se pone a T_j en espera.

  Otras estrategias involucran establecer un **tiempo de timeout** (su definición no es trivial), **No Waiting** (NW) o **Cautious Waiting** (CW). En la última, si T_i solicita el lock de un ítem bloqueado por T_j se tiene que: si T_j no está bloqueada esperando a otra transacción, T_i se bloquea, y si no se aborta.

Paradigma optimista - Algoritmo por control de timestamp

En este mecanismo no hay locking sino que el TM le asigna a cada operación un **timestamp** (TS, el mismo para las de una misma transacción). Luego para dos operaciones en conflicto se decide su orden según este número (se puede tomar del reloj del sistema o un contador).
Cada operación se procesa enviándose al data manager (DM) y se verifica con el planificador usando tres datos de cada ítem X: su máximo TS de escritura (**WT(X)**), su máximo TS de lectura (**RT(X)**) y su bit de commit (**C(X)**) que indica si la transacción más reciente que lo actualizó ya hizo commit. Con estos, el planificador asume al verificar una operación que el orden de los timestamps es el orden serial y que cada una pudo haber ocurrido si cada transacción se hubiese realizado instantáneamente en su TS. De no darse este comportamiento, la transacción es **físicamente irrealizable**.

Reglas del planificador

Si la operación recibida es $r_T(X)$ y:

- $TS(T) >= WT(X)$: Si C(X) es True se concede la solicitud y se actualiza RT(X) con el máximo entre TS(T) y RT(X). Si es False se demora T hasta que C(X) sea True o T aborte (previniendo **dirty read**).

- $TS(T) < WT(X)$: La transacción es físicamente irrealizable (**read too late**).

Si en cambio recibe $w_T(X)$ y:

- $TS(T) >= RT(X)$ y $TS(T) >= WT(X)$: Se escribe el nuevo valor de X, se actualiza WT(X) y C(X) ← False.
- $TS(T) >= RT(X)$ y $TS(T) < WT(X)$: Si C(X) es True se aplica la **Thomas Write Rule** y se ignora la operación de escritura para evitar la sobreescritura. Si es False, se demora T hasta que la transacción anterior termine.
- $TS(T) < RT(X)$: La operación es físicamente irrealizable (**write too late**).

Si recibe $C(T)$, para cada ítem X tal que $WT(X) = TS(T)$ se pone C(X) en True y se prosigue con las transacciones atrasadas por este evento.
Finalmente, si recibe $Rollback(T)$ (R(T) o A(T)) cada transacción esperando por un ítem X tal que $WT(X) = TS(T)$ se reinicia y verifica nuevamente.
Para los casos en que la operación es físicamente irrealizable, se produce un rollback de la transacción con un nuevo TS.
Nótese cómo este planificador evita el problema de **lost update** al abortar la lectura o ignorando la sobreescritura.

Paradigma optimista - Control de concurrencia por multiversión

Este mecanismo se basa en atacar los problemas de **read too late** del algoritmo anterior al darle a cada transacción la versión vigente de los data items al momento de su inicio. Esto hace que se guarden tantas **versiones** como la más antigua vigente de las transacciones activas o commiteadas (no abortadas).
Para esto, al ocurrir una escritura $w_T(X)$ (legal) se crea una nueva versión de X como X_t con $t = TS(T)$ en la base de datos. En una lectura $r_T(X)$ se busca X_t tal que t sea el máximo TS menor o igual a TS(T). Esto hace que WT(X) se interprete como el máximo t para el que exista X_t. Para RT(X) se guarda un valor tr para cada Xt y si $w_{T'}(X)$ es tal que exsite $X_{t,tr}$ tal que $t < TS(T')$ y $tr > TS(T')$ (write too late) entonces se debe hacer rollback de la transacción. Además, se puede borrar X_t si no existe transacción activa T tal que $TS(T) < t$.

Comparación entre los paradigmas

- **Similitudes:** En ambos puede haber **starvation** de una transacción y se tiene un **registro global** para manejar el control de concurrencia de las transacciones: en el pesimista son los locks y en el optimista una serie de timestamps de las operaciones con su bit de commit (agrupados en sets en el caso de implementar validación).
- **Diferencias:** El optimista es mejor si la mayoría de las operaciones son de consulta (escritura) o raramente las transacciones decidan leer y actualizar el mismo elemento de la DB. Locking es más efectivo en situaciones de mayor conflicto. Por ende, los sistemas comerciales suelen establecer un **compromiso** según el tipo de transacción: si es read-only usan las versiones creadas por read-write; si es read-write usan locking creando versiones de los elementos.

Recuperación

El DBMS debe procurar que, al recibir una transacción todas sus operaciones sean ejecutadas exitosamente, almacenando sus resultados en la DB (commit), o que ninguna operación tenga efecto sobre ella (abort). Si ocurre una **falla**, en el primer caso el DBMS debe asegurarse que los cambios pasen a disco y en el segundo, que si ya se realizaron operaciones sus cambios se deshagan ya que pudieron haber dejado a la base de datos con valores inconsistentes.
Entre las posibles fallas de un sistema tenemos:

- **Fallo de la Computadora (system crash):** Error de HW, SW o red.
- **Falla en la transacción o sistema:** Error en la lógica de la programación (división por cero) o interrupción del usuario.
- **Errores locales o condiciones de excepción detectadas:** Los datos necesarios de la transacción no fueron encontrados.
- **Ejecución del Control de Concurrencia:** Abort por violar la serialización o producir un deadlock. Estas suelen reiniciarse posteriormente.
- **Falla de disco:** Durante una lectura o escritura.
- **Problemas físicos o catástrofes:** Suministro eléctrico, aire acondicionado, incendio, robo, sabotaje, entre otros.

De ellas, salvo las dos últimas, el sistema deberá poder mantener la suficiente información para reestablecerse rápidamente en un modo a prueba de fallas. De esto se encarga el **subsistema de recovery** del DBMS a través de los módulos cache manager (CM), recovery manager (o log manager) y un archivo **log** en disco.

Log

Mantiene un **registro** de todas las operaciones de las transacciones que afectan a la BD de manera de reestablecer el sistema ante fallas. Es incremental y secuencial, se guarda en disco pero no se salva de fallas a disco ni catástrofes, por lo que períodicamente es preferente guardar una **copia de seguridad** en otro dispositivo. Su presencia hace que para que una transacción se considere **commiteada** sus cambios debieron haber sido registrados en el log.
Entre los registros que se pueden almacenar en el log tenemos:

- **Start record (`<START` T_i`>`):** El incio de la transacción T_i.
- **Commit record (`<COMMIT` T_i`>`):** El final exitoso de la transacción T_i indicando que sus cambios deben guardarse permanentemente en la DB.
- **Abort record (`<ABORT` T_i`>`)**: El final no exitoso de una transacción T_i, indicando que falló y sus cambios deben ser deshechos de la DB.
- **Update record:** La actualización de un ítem X por parte de la transacción T_i.

Nótese que una transacción se considera **incompleta** si no contiene un commit record ni abort record, los cuales sólo pueden estar al final de ella.
El **método de logging** determina la información que guardará el update record en el log y cómo operará el RM ante una falla. Entre ellos tenemos undo logging, redo logging y undo/redo logging.

Garbage Collection Rule

Una entrada $[T_i, x, V]$ puede ser eliminada del log si pasa una de estas condiciones:

- T_i Abortó.
- T_i Comiteó, T_j es una transacción posterior a T_i, T_j comiteó, y $[T_j, x, W]$ aparece en el log. O sea, si otra transacción comiteó después y modificó dicho valor, por lo que $[T_i, x, V]$ no es **el último valor comiteado**.

Undo logging

Se basa en permitir que las **transacciones incompletas** escriban a disco y deshacer sus cambios si no llegaron a hacer commit antes de la falla. Aquí los update records son de la forma `<`T_i`, X, v>` indicando que T_i actualizó el valor de X cuya imagen anterior era v. Cada update record se baja a disco antes de bajar el nuevo valor de X y el commit record se baja después de todos los cambios de la transacción, indicando que son válidos.
Al producirse una falla:

1. Se recorre el log desde el final hacia atrás marcando las transacciones según si se encuentra un commit record o abort record.
2. Por cada update record `<`T_i`, X, v>` si se encontró un commit o abort record para T_i, saltearlo. Si no, asignar v a X.
3. Agregar un abort record al log por cada T_i incompleta y bajarlos a disco.

Este método es conveniente al tener **pocas transacciones incompletas** ya que deriva en pocos cambios a deshacer. Además, se pueden ir bajando datos a la DB durante la escritura del log para ahorrar tiempo. No obstante, requiere **leer todo el archivo de log** y que los ítems se bajen a disco justo después de que la transacción termine, aumentando la cantidad de operaciones de entrada y salida.

Redo logging

Se basa en tomar las **transacciones completas** cuyos cambios no llegaron a bajarse a disco. Sus update record son de la forma `<`T_i`, X, v>` donde la transacción T_i actualizó a X con el valor v. Estos se bajan a disco antes que el nuevo valor de X pero el commit record también, de manera de asegurar en el log los cambios a escribir en la BD.
Al producirse una falla:

1. Se recorre el log una primera vez identificando las transacciones completas e incompletas.
2. En el segundo recorrido de atrás hacia adelante, por cada update record `<`T_i`, X, v>` si T_i es incompleta no hacer nada; pero si no asignar v a X.
3. Por cada T_i incompleta agregar un abort record al log y bajarlos a disco.

Nótese que puede optarse por borrar los registros de una transacción abortada o commiteada en caso de haber una más reciente que haya actualizado sus ítems. Sin embargo, en la recuperación **el archivo de log debe leerse dos veces** y como las transacciones suelen ser completas va a haber muchos cambios a rehacer. A su vez, como los cambios no pueden bajarse a la BD hasta luego del commit es necesario mantener los bloques modificados en un **buffer**.

Undo/Redo logging

Es una combinación de los anteriores que se basa en atacar tanto **transacciones incompletas como completas**. Sus update records son de la forma `<`T_i`, X, `v_0`, `v_1`>` donde T_i actualizó el valor del ítem X de v_0 a v_1. Cada uno de ellos se baja a disco antes de acutalizar X. Luego se baja su commit record.
Con ello, si ocurre una falla:

1. Se aplica **UNDO** a las transacciones incompletas en orden inverso (usando v_0 para actualizar el valor de X por cada update record).
2. Se aplica **REDO** a las transacciones completas en orden (asignado v_1 a X por cada update record).
3. Agregar un abort record a cada transacción incompleta y bajarlos a disco.

Esto hace que el sistema sea más **flexible** a los escenarios de fallas, pero para ello debe guardar más información y el trabajo de recuperación es mayor.

Checkpoint

Considerando que regularmente es necesario bajar los datos del log al disco ya que de expandirse lo suficiente se perdería mucho tiempo en la recuperación, podemos implementar un **sistema de checkpoints**. Se basa en dos técnicas:

- Actualizar el log junto con la lista de transacciones commiteadas y abortadas hasta el momento de manera de indicar las modificaciones escritas y deshechas que no deben realizarse.
- Escribir las imágenes posteriores a las modificaciones efectuadas por las transacciones commiteadas o las imágenes previas de las abortadas a disco. Esta es opcional.

En base a ello, los mecanismos de checkpoint son:

- **Quiescente:** Este deja de aceptar nuevas transacciones mientras espera a que las activas (las empezadas sin registro de commit o abort) terminen. Luego agrega al log el registro `<CKPT>` y lo baja a disco (no hay transacciones activas). Recién en ese momento puede aceptar nuevas transacciones. En la recuperación sólo aplica el método **UNDO** desde el final del log hasta el último checkpoint (el primero encontrado).
- **No quiescente:** Este sigue aceptando transacciones pero efectúa un **flush** cuando todas las activas a su inicio terminan. Para eso usa el registro `<START CKPT (...)>` para indicar su inicio en el log con sus transacciones activas y `<END CKPT>` para indicar que terminaron.
  Sus etapas y recuperación varían según el método de logging:
  - **UNDO:** Parte de agregar el registro `<START CKPT (...)>` al log y bajarlo a disco. Seguidamente, se espera a que estas transacciones terminen, sin restringir que empiecen nuevas. Al terminar se agrega el registro `<END CKPT>`, bajándolo a disco.
    En la recuperación se empieza recorriendo desde el final. En caso de encontrar un `<END CKPT>` se procede hasta hallar un `<START CKPT (`T_1`, ..., `T_k`)>`. Seguidamente, se continua recorriendo hasta hallar el `<START `T_i`>` más antiguo entre las T_i del registro. Finalmente se procede con el método normal.
  - **REDO:** Parte similar al anterior, agregando el registro `<START CKPT (...)>` y bajándolo a disco. Ahora, espera a que se bajen a disco los cambios de todas las transacciones ya commiteadas al momento de iniciar el checkpoint. Finalmente, se agrega el registro `<END CKPT>` al log y se lo baja a disco.
    En la recuperación se indentifica al último checkpoint finalizado correctamente (el último con un `<END CKPT>` posterior). Luego, desde su registro `<START CKPT (...)>` se aplica **REDO** a todas las transacciones que iniciaron luego de su inicio o estaban activas en ese momento.
  - **UNDO/REDO:** Agrega al log los registros `<START CKPT (...)>` y `<END CKPT>` como en el caso **REDO**. La diferencia recae en que antes de aplicar **REDO** sobre las completas aplica **UNDO** desde el final hasta encontrar el `<START `T_i`>` más antiguo del último `<START CKPT (...)>` para las incompletas.

Optimización

Una **query** (consulta) tiene muchas formas de ser ejecutada para obtener el set deseado. Cada una define un **plan** (query plan) y para elegir cuál ejecutar el DBMS cuenta con un módulo llamado **query optimizer**.
Ahora, como encontrar el plan óptimo es un **problema NP-completo**, el módulo debe emplear otras técnicas para hallarlo en un tiempo razonable. Entre ellas tenemos:

- **Uso de heurísticas:** Se aplican transformaciones del álgebra relacional con la propiedad de mantener los resultados obtenidos. Estas suelen mejorar la performance pero no siempre.
- **Estimación de selectividad:** Se hace uso de la información en la base de datos para estimar el grado de selectividad de la consulta (cantidad de tuplas devueltas). Esto permite darnos una idea de su costo.

- **Índices y tipo de archivo:** El plan elegido depende fuertemente de los índices dispuestos en las tablas y cómo se ordenan físicamente los datos en disco.

Una vez obtenido el plan, el code generator se encarga de generar su código correspondiente y el runtime database processor de ejecutarlo.

Métricas de relaciones

Dada una relación R tenemos una serie de parámetros del catálogo que podemos medir y usar para optimizar una consulta:

- **Bloque:** Porción de datos levantada por cada lectura a disco.
- **LB:** Longitud de los bloques.
- **B$_R$:** Cantidad de bloques ocupados por R.
- **FB$_R$:** Factor de bloqueo de R, es decir, la cantidad de tuplas de R que entran en un bloque.
- **L$_R$:** Longitud de una tupla de R.
- **T$_R$**: Cantidad de tuplas de R.
- **I$_{R,A}$**: Imagen del atributo A de R, es decir, cantidad de valores distintos de la columna A en R.
- **X$_I$**: Altura del árbol de búsqueda (B+) con índice I.
- **FB$_I$**: Factor de bloqueo de I, es decir, cantidad de tuplas del índice I que entran en un bloque.
- **BH$_I$**: Cantidad de bloques que ocupan todas las hojas del índice I.
- **MBxB$_I$**: Cantidad máxima de bloques en un bucket del índice hash I.
- **CBu$_I$:** Cantidad de buckets del índice hash I.
- **B**: Cantidad de bloques que entran en memoria principal.

Almacenamiento físico

Las relaciones se almacenan en **archivos** y cada tupla tiene asociado un identificador llamado **rid** (register id), que no es un atributo de la relación. Estos archivos pueden ser:

- **Heap files:** Son el tipo de archivo más simple consistente en una **colección desordenada** de registros agrupados en bloques. Aquí su **costo** de exploración completa (scan), búsqueda por igualdad ($A = c$) y búsqueda por rango ($c <= A <= d$) es BR (lineal en la cantidad de bloques). Puede tener páginas para datos y otras para el **mapa de reserva de índices** (IAM, Index Allocation Table) que sirve para saber dónde están guardados los datos en disco.
- **Sorted file:** Estos mantienen sus **registros ordenados** según el valor de determinados campos. Si bien su **costo** de exploración completa sigue siendo BR, el de búsqueda por igualdad y búsqueda por rango es $log_2(BR)+[T'/FB_R]$, donde T' es la cantidad de tuplas que cumplen con el criterio de búsqueda (y FB_R la cantidad de tuplas por bloque de R).

Índices

Son **diccionarios de claves** (no necesariamente únicas) asociados a la relación. Cada una se corresponde con una o más columnas (atributos) de ella y su valor asociado es el de sus registros (o tupla). Puede ser el registro completo, su rid a una lista con los rid de todos los registros asociados a ella.
De estos se tienen tres tipos:

- **Clustered:** Dicta el **orden físico** de los datos del archivo, que sólo puede tener uno definido. Se almacenan en forma de **árbol de búsqueda balanceado** de páginas (B+) de hasta 3 niveles donde sus hojas se corresponden a los datos y cada nodo intermedio referencia en sus filas la dirección física y el valor mínimo de la clave de la página (salvo en la primera que guarda `NULL` para insertar una fila con clave más baja en la tabla de forma óptima).
- **Non-clustered**: No determinan el orden de los datos y se almacenan en una **estructura por fuera** de ellos. Se parecen a los anteriores en estructura con la salvedad de que en sus hojas almacenan el valor de la clave y un **rowid**, que según si se combinan con una heap table o clustered index, es su dirección física o la clave de su fila (respectivamente). En los nodos intermedios se guarda la dirección física de la página con el valor mínimo de clave. Puede guardarse el rowid en caso de haber más de un índice definido.

- **Hash estático:** Se define como una **tabla de hash** con una cantidad estática de buckets. Es muy útil para realizar búsquedas por igualdad ya que su costo es $MBxB_I$ (cantidad máxima de bloques por bucket). No obstante, para el resto de las operaciones requieren de un barrido lineal (file scan).

Los índices pueden ser **densos** o no según si almacenan una entrada por cada registro en la base de datos o sólo algunos. Además son **primarios** si guardan registros completos de archivos (sólo uno por tabla) o **secundarios** si sus valores son rids (puede haber más de uno por tabla).

Árbol de búsqueda B+

Es el **árbol balanceado** en el que se basan los primeros índices. Siempre tienen una página hoja y otra raíz a menos que la tabla entre toda en una sola. Se busca que **los datos no se solapen** entre páginas y de sobrar lugar se pueden agregar tablas de índices.
Cada nodo interno (y la raíz) tiene una cantidad de hijos y claves dada por un parámetro d (entre $d/2$ y d). Las hojas tienen información asociada a la clave de los registros del archivo y se pueden recorrer como *listasdoblementeenlazadas*. Esto hace que la estructura sea recomendada para acceder a rangos de claves.
Si bien para su exploración completa es necesario un **file scan**, para las búsquedas cada índice tiene cierta optimización:

- **Clustered:** En la **búsqueda por igualdad** se recorre el árbol en busca de la primera ocurrencia de la clave. Con ella se busca su registro asociado en el archivo y partir de este punto se lo recorre mientras se hallen ocurrencias de la clave. Esto hace que su costo sea $X_I + [T'/FB_R]$.
  Para la **búsqueda por rango** el costo es el mismo ya que se emplea el msimo método para hallar el primer elemento del rango y recorrer el archivo desde ahí.
- **Non-clustered:** La **búsqueda por igualdad** es similar al caso clustered pero al llegar a las hojas del árbol se las recorre secuencialmente mientras haya ocurrencias de la clave. Luego, por cada una se leen sus registros asociados. Esto lleva su costo a $X_I - 1 + [T'/FB_I] + T'$.
  En la **búsqueda por rango** el costo es el mismo ya que se sigue el mismo procedimiento para hallar el primer elemento del rango y a partir de este todas sus ocurrencias (leyendo sus registros asociados en el archivo).

Operadores del plan de ejecución

Las consultas se procesan por etapas:

- **Parse:** Validación de la sintaxis de la sentencia SQL con su transformación inicial a árbol de ejecución.
- **Bind:** Vínculo entre los objetos y la carga de metadata.
- **Optimize:** Generación del plan de ejecución.
- **Execute:** Ejecución del plan.

En un plan de ejecución, cada nodo es un operador que implementa al menos los métodos:

- **Open():** Se inicializa el operador y se configuran las estructuras de datos requeridas.
- **GetRow():** Se toma una fila del operador.
- **Close():** Se finaliza el operador limpiando las estructuras y datos necesarios.

En base a ellos, tenemos dos operadores de acceso a datos:

- **scan:** Barre una estructura. Se denota **file scan** si recorre todas las entradas de un archivo, o **index scan**, las de un índice.
- **seek:** Recorre una estructura en base a un índice. Tenemos dos casos:
  - Un índice de un **árbol B+** con un predicado como una conjunción de términos con atributos de un prefijo de su clave (su tupla).
  - Un índice **hash** con un predicado como una conjunción de términos con todos los atributos del índice.

  En el caso de un índice non-clustered, al pasar de una hoja a otra estructura se efectúa un **bookmark lookup** (por rid o clave).

Juntas:

Tenemos 3 tipos de operadores de juntas entre relaciones:

- **Nested Loops Join:** Son las más básicas de implementar ya que buscan los elementos recorriendo una tabla a partir de la otra. Son **efectivas** cuando el inner input es más grande que el outer input y el primero está indexado. Según si el índice forma parte del atributo del join se tiene el **Index Nested Loop Join** (INLJ) o

el **Block Nested Loop Join** (BNLJ). El costo del primero depende del tipo de índice, siendo el segundo de fuerza bruta. Además el predicado puede no ser de igualdad.

- **Merge Join:** Recorre ordenamente las tablas a través de un scan. Su **costo** se basa en el algoritmo de ordenamiento (de necesitarse a través de un operador) y el merge (lineal en el tamaño de ambas tablas por condición de igualdad).
- **Hash Join:** Se usa para procesar entradas largas, no ordenadas y sin índices eficientes. Su predicado es de **igualdad**.

Otros operadores

- **Filtro:** Se usan para las condiciones impuestas por la cláusula `HAVING`, las cuales son eficientes para operar sobre la memoria.
- **Agrupamiento (agregación):** Tenemos **stream** y **hash**, donde el primero se usa siempre que no se tenga la cláusula `GROUP BY` y en caso de hacerlo, los datos deben estar ordenados.
- **Compute Scalar:** Realizan operaciones de conversión, cálculo de datos y otras de cómputo matemático.
- **Union:** Permiten agrupar relaciones y se dividen en **merge**, **hash** y **concat**.

Optimizaciones y heurísticas

- **Algebráicas:** Estas buscan mejorar la performance de la consulta más allá de su representación física aprovechando las propiedades algebráicas del AR:
  - Cascada de select/project por conjunción.
  - Conmutatividad de select/select con respecto a project/producto cartesiano (y la junta)/unión e intersección/select y project con respecto al producto cartesiano (y la junta).
  - Asociatividad del producto cartesiano, junta, unión e intersección.
- La **materialización** es la escritura de un resultado en disco. Por otra parte, el **pipelining** se da entre dos operaciones O_1 y O_2 si las tuplas resultantes de la primera pasan a través de un **stream** en memoria a la segunda. Esto hace que se ahorre el costo de output de $O1$ y de input de O_2. No es aplicable a operaciones que necesitan todas las relaciones de antemano, como las juntas.
- La **integridad referencial** permite evitar consultar a una tabla a través de juntas.
- Considerar sólo **árboles sesgados a izquierda** permite limitar los árboles candidatos a analizar.
- Al descomponer las **selecciones conjuntivas** podemos aprovechar los índices de cada relación por separado.
- Tomar las selecciones lo más cercano posible a las hojas del árbol reduce la cantidad de datos a materializar al seleccionar prematuramente las tuplas que nos interesan.
- Reemplazar **productos cartesianos** seguidos de selecciones por juntas.
- Descomponer las listas de atributos de proyecciones y llevarlas lo más cerca posible de las hojas.
- Los **hints** son instrucciones para el motor de búsqueda para cambiar su operación. Se especifican a través de la cláusula `OPTION` y sólo cambian su procesamiento, no su semántica (retornar rápidamente una serie de filas, buscar usando un índice en específico, recompilar una consulta, optimizar en base a un parámetro dado o *parameter sniffing*, entre otras).
- Para comparar y analizar las implementaciones de una consulta tenemos las **estadísticas**. Entre sus métricas, muestran la distribución de los datos en un histograma contando límites por paso, rango de valores, cantidad estimada promedio, valores distintivos y cantidad de duplicados promedio.
- Para optimizar lecturas se pueden utilizar los atributos buscables a través de índices (**SARGable**: search-ARGument-table). Estos se aplican sólo cuando los predicados tienen condiciones de inclusión. En los índices compuestos depende del predicado de su primera columna.
- La **cobertura de un índice** permite guardar partes selectivas en base a un criterio dado y con ello retornar la información necesaria o que se pueda llegar a usar en el futuro. También se puede filtrar el indexado para, por ejemplo, tomar todos los valores no nulos y así recorrer la tabla más rápidamente.

Long duration transacions

Son aquellas transacciones que caen dentro de dos escenarios:

- Modifican una **cantidad muy grande de registros** de la base de datos. Si se ejecutasen como una sola transacción tardarían mucho tiempo en ejecutarse y requerirían almacenar mucha información para hacer rollback.
- Deben **acceder a diferentes bases de datos** para ejecutarse.

Para tratarlas definimos una **saga** como una colección de acciones que juntas conforman a la transacción. Esta consiste además de un **grafo dirigido** donde cada nodo es una **acción**, se tienen los nodos terminales Abort y Complete (no salen ejes de ellos), y una marca del nodo inicial.
Con esto, cada camino del grafo representa un **curso de acción** (secuencia de acciones) que puede llevar a *Abort* ó *Complete*. Cada acción puede verse como una transacción y para cada una se define su "transacción de compensación" como aquella que al ejecutarse permite revertir la base al estado anterior a ejecutarse su transacción correspondiente (para la acción A se define como A^{-1}).
Entonces, si al ejecutar un curso se llega a *Abort* se usan las transacciones de compensación de sus acciones en el orden inverso para deshacer sus cambios de la base de datos. Si en cambio se llega a *Complete* los cambios se pueden mantener en la base.

Bases de datos distribuidas (DDB)

Se definen como colecciones de BDs **lógicamente relacionadas** y **distribuidas** a través de una red de computadoras. Son manejadas por un **DDBMS** que busca hacer transparente la distribución al usuario y que sus nodos cumplan con una serie de características.
Dada una consulta su estrategia de ejecución depende del **tipo y la topología de la interconexión** entre los nodos de la red (LAN, WAN, conexión directa, conectividad). También depende de la **forma** en que esté distribuida la información (dada una junta debemos atender a la cantidad de datos que deben ser transeridos).

Fragmentación

Los **fragmentos** son las partes resultantes de dividir la DB en unidades lógicas. La fragmentación puede ser:

- **Horizontal (sharding):** Subdivisión en **subconjuntos de tuplas** de la relación original. Es análoga al operador `SELECT` y de comprender a todas las tuplas se denomina **completa** (puede ser disjunta). Su reconstrucción viene dada por `UNION`. Aumenta el performance de las escrituras y lecturas distribuyendo la carga en distintos nodos, pero reduce el performance de operaciones analíticas al requerir `UNIONS` cuando se quiere todos los datos de la tabla (ver OLAP y OLTP).
- **Vertical:** Subdivisión en subrelaciones dadas por un **subconjunto de columnas** de la original. Es análoga al **proyector** y si cada atributo está en al menos una proyección y todas comparten sólo la PK se denomina **completa**. La reconstrucción viene dada por `OUTER JOIN`. Aumenta el performance de las lecturas y escrituras ya que distribuye la carga en distintos nodos cuando se trata de accesos a pocas columnas, pero a costa que las operaciones que requieran el uso de todas las columnas se vean ralentizadas por la necesidad de realizar `JOINS`.
- **Mixta:** Combinación de las dos anteriores que puede reconstruir la relación con las operaciones en el orden apropiado.

Un **esquema de fragmentación** de una DB es un conjunto de fragmentos que incluye a todos sus atributos y tuplas, y permite reconstruirla con la secuencia de operaciones apropiada. Por otra parte, un **esquema de asignación** describe la ubiación de los fragmentos en los nodos de la DDB y de estar uno en más de un lugar se denomina **replicado**.

Replicación

Las **replicas** son copias de los datos que permiten aumentar su disponibilidad y confiabilidad. Según su nivel, la replicación es:

- **Total:** Todos los nodos tienen copias de la DB.
  Ventajas: El sistema puede continuar operando si al menos uno está disponible y mejora el rendimiento de lecturas.
  Desventajas: El rendimiento de escrituras empeora y las técnicas de control de concurrencia y recuperación son más costosas.
- **Nula:** Caso adverso al anterior en que ningún elemento está replicado.
  Ventajas: Las escrituras son menos costosas y no requieren de tanto control de concurrencia ni recuperación.
  Desventajas: Tiene múltiples puntos de falla y cuello de botella para lecturas.

- **Parcial:** Algunos elementos se replican y otros no, distribuyéndose a sitios particulares. Esta depende de las metas de disponibilidad, performance y el tipo de transacciones de cada sitio del sistema. La descripción de la replicación viene dada por el esquema de replicación.

Transparencia

Se basa en **ocultar los detalles** de implementación a los usuarios finales. Si es **total**, la visión del usuario es la de una DB centralizada donde se centra en la **independencia** entre datos lógicos y físicos (con el compromiso del sobrecosto de proveerla). Más específicamente, podemos ver distintos **Tipos de Transparencia** para las DBs distribuidas:

- **Organización de los datos:** Cómo están distribuidos los datos a través de la red. Se divide en transparencia de **ubicación** (ejecutar una tarea independientemente de dónde estén los datos) y de **nombres** (tras asociar un nombre a un objeto no hacen falta datos adicionales para ubicarlo).
- **Fragmentación:** El usuario **se libera de conocer detalles sobre la fragmentación** de los datos. Esa transparencia puede ser para la fragmentación **horizontal**, **vertical**, o **ambas**.
- **Replicación:** El usuario **se libera de conocer detalles sobre la replicación** de los datos, **de su ubicación** y la razón por la que fueron replicados, sin embargo **puede** llegar a **ser configurable**.
- **Otras Transparencias:** Como de **Diseño o Ejecución**, que **lo liberan al usuario de entender donde o cómo se procesan las queries**, y como esta diseñada la misma en y entre los nodos.

La **transparencia** tiene la ventaja de proveer una visión de la Base de Datos como si fuera **centralizada**. Pero tiene el problema de no permitir reflejar características deseables del **DDBMS**. Por tanto la transparencia **debe poder ser configurable**, o **debe haber un compromiso entre la facilidad de uso** y **el sobrecosto de proveer transparencia**.

Características esperables

- **Disponibilidad y confiabilidad:** Se refieren a la **probabilidad** de que un sistema se encuentre continuamente disponible u operando en un determinado intervalo de tiempo (respectivamente, aunque los términos se usen de forma indistinta). Ante las fallas del sistema, un sistema **confiable** puede:
  - **Enfatizar su tolerancia a las fallas**, reconociendo que pueden ocurrir y diseñar mecanismos que las detecten y las remuevan antes de que ocurran.
  - **Asegurar la ausencia de fallas** en el sistema final mediante procesos de desarrollo que incluyen control de calidad y testing.

  En el caso de los DDBMSs, estos deben **tolerar fallos de sus componentes subyacentes** sin alterar los pedidos de sus usuarios ni la consistencia de la base. El RM se encarga de tratar fallas en las transacciones, hardware y comunicaciones en las redes.
- **Escalabilidad y tolerancia a la partición:** La primera se refiere a la medida en que un sistema puede **expandirse** mientras continúe operando ininterrumpidamente. Puede ser **horizontal** (cantidad de nodos del sistema distribuido) o **vertical** (capacidad de un nodo individual).
  La segunda implica que el sistema pueda seguir operando aún cuando la red se particione.
- **Autonomía:** La medida en que los nodos individuales de una DDB puedan operar **independientemente**. Se busca que tenga alto grado y hay de varios tipos:
  - **Diseño:** Independencia del uso del modelo de datos/técnicas de gestión de transacciones entre los nodos.
  - **Comunicación:** Medida en la que los nodos deciden compartir o no información con otros.
  - **Ejecución:** Acciones "a gusto" del usuario.

Teorema CAP

Publicado en 1998 y presentrado por Eric Brewen en el año 2000, establece que cualquier sistema que comparta datos a través de la red puede cumplir con a lo sumo dos de las siguientes propiedades:

- **Consistency:** Toda lectura devuelve la escritura más reciente del ítem.
- **Availability:** Todo pedido recibe una respuesta, pero sin el anterior ítem, no se garantiza que esta respuesta sea la de la última escritura. Pero sí garantiza que no es errónea o un código de error.
- **Partition tolerance:** El sistema sigue funcionando a pesar de la perdida de una cantidad arbitraria de paquetes (mensajes) o nodos en la red (incluso aunque los pierda todos). La idea es que si decide seguir con la operación, puede perder consistencia por no disponer de la información que el nodo caído o incomunicable dispone, o

puede arriesgar disponibilidad si decide cancelar la operación o retrasarla (porque puede terminar retrasándola infinitamente).

Según Brewen, el objetivo del teorema debe enfocarse en encontrar el balance entre consistencia y disponbilidad mientras se encuentre la forma de manejar las particiones de la red, considerando cómo recuperarse ante eventuales fallas. Para lo segundo, el sistema debe detectar la partición, entrar en modo "particionado" y luego recuperar el modo anterior.

Propiedades BASE

Son las propiedades que cumplen las DBs NoSQL (de las cuales muchas incorporan funcionalidades de DDBs) y son mucho más laxas, o débiles, que las propiedades **ACID** (las cuáles también son deseables en una DDB). Estas son:

- **Basic Availability:** El sistema debe garantizar disponibilidad en términos del teorema **CAP** incluso cuando ocurren fallas (si se produce una falla, en el peor de los casos la Base de Datos se reserva la posibilidad de particionarse). No obstante, esta disponibilidad no garantiza consistencia, las lecturas puede no leer la última escritura, y la escritura puede no persistir si se encuentran conflictos no reconciliables. Esto puede lograrlo a través de una base de datos **altamente distribuida**, en donde no se guarde una sola copia con tolerancia a fallas sino múltiples réplicas en discos. De esa forma, una falla que inhabilita el acceso a un grupo de datos no necesariamente lo hace a toda la DB.
- **Soft-state:** El estado de la DB puede cambiar a lo largo del tiempo incluso sin intervención externa. Esto significa que en un momento dado no necesariamente habrá una única "versión" de cada dato y la **consistencia** debe ser manejada por los desarrolladores, no el DBMS.
- **Eventual consistency:** El sistema garantiza que posterior a la ejecución de escrituras, el sistema converge después de un tiempo no determinado, a un estado donde cualquier lectura de ese ítem de datos siempre retorne el mismo valor, recalcando de vuelta, que los conflictos de concurrencia quedan en manos de los desarrolladores y no del DBMS.

Ventajas de las DDBs

- Permiten mejorar sencilla y flexiblemente el desarrollo de aplicaciones, debido a la transparencia de control y distribución de los datos.
- Aumentan la disponibilidad al aislar sus fallas a su sitio de origen.
- Mejoran la performance al fragmentar y replicar los datos, dando nuevas oportunidades de conseguir escalabilidad horizontal (distribuyendo la carga entre nodos) y mantener cerca los datos necesitados (en nodos geográficamente cercanos a dónde se realice la consulta).

Control de concurrencia (CC) en DDBs

Este subsistema se encarga de **mantener la consistencia** entre las copias de un data ítem en la DB (mientras intenta mantener el resto de las propiedades de las bases centralizadas). Para ello se puede usar la técnica de **locking** en donde para cada ítem se designa una copia como **distinguida** y los pedidos de lock y unlock son enviados a ella.
Según su ubicación, se tienen diferentes variantes:

- **Sitio primario:** Un único sitio es **distinguido** y encargado de procesar las transacciones. Si le otorga el lock a un nodo, puede acceder a cualquier copia de dicho valor. El DDBMS debe asegurarse que al modificar un valor, este se refleje en todas las copias de dicho valor.
  Ventajas: Es una extensión del esquema centralizado y si las transacciones cumplen con el enfoque 2PC la seriabilidad está garantizada.
  Desventajas: Hay cuello de botella en el único sitio así como un único punto de falla que puede paralizar al sistema.
- **Sitio primario + backup:** Similar al anterior pero con un **sitio adicional de backup** para reemplazar al primario en caso de fallar.
  Ventajas: Simplifica el proceso de recovery del sitio primario.
  Desventajas: La performance de locking disminuye y persiste el problema del cuello de botella.
- **Copia primaria:** Permite que los sitios que almacenan las copias **sean encargados de procesar las Transacciones y locks**, en vez de tener un Sitio Primario que se encargue de todo. Entre los que tienen la copia de un data item, **uno** es seleccionado como **distinguido**, y se encarga de los locks **de dicho data item**.
  Ventajas: Ataca el problema del cuello de botella y el único punto de falla del sitio primario. De combinarse con backups, se puede mejorar su disponibilidad y confiabilidad (aunque con cierto costo adicional). Además, la caída de un nodo solo afecta a las transacciones que pidieron locks en data items de dicho nodo.
  Desventajas: La complejidad del control de concurrencia es muchísimo mayor al no parecerse a sus contrapartes

centralizadas, y el uso de la red para algoritmos de votación y decisión va a ser mayor que en casos de Sitio Primario.

Elección de nuevo coordinador

Al suceder una **falla**, en los casos sin backup las transacciones que acceden a los datos del sitio afectado deben ser abortadas y reiniciadas, mientras que de haber backup estas son suspendidas hasta ser designados estos sitios como primarios. Si ambos fallan puede iniciarse un **proceso de elección** de un nuevo coordinador desde un sitio Y:

1. El coordinador se considera **fallido** si el sitio Y intenta reiteradamente comunicarse con él y no lo logra.
2. Luego, este sitio envía a todos los nodos activos que será el nuevo coordinador.
3. De recibir la mayoría de los votos positivos, se lo declara como tal. Esto resuelve los casos en que más de un sitio desea ser coordinador a la vez.

Votación

Alternativamente a la copia distinguida, se puede hacer un **pedido de lock(X)** a todos los sitios con el ítem X. De allí, cada copia puede aceptarlo o rechazarlo. Si recibe la mayoría de aprobaciones en un cierto período de tiempo, les avisa a todas las copias que tomó el ítem. Si no, cancela el pedido y les informa de la cancelación.
Ventajas: Es un método de CC **realmente distribuido** por compartir la decisión entre sus nodos. **Desventajas:** Genera mucho tráfico de mensajes y de caerse nodos durante la votación el algoritmo puede acomplejizarse.

Recuperación en DDBs

Este subsistema debe atender a dos principales problemas:

- **Fallos en sitios y comunicaciones:** A partir de un sitio X es difícil determinar si un sitio Y se encuentra **caido** sin intercambiar una gran cantidad de mensajes. En efecto, la falta de respuesta de Y podría deberse además a una falla en la comunicación que hace que no haya recibido el mensaje o no haya podido enviarlo.
- **Commit distribuido:** Cuando una transacción actualiza un valor en varios sitios, no puede commitearlo hasta asegurarse que sus efectos no se perderán en ninguno de ellos (log). Para asegurar la correctitud frente a este problema se suele usar el **protocolo 2PC**.

Gestión de transacciones distribuidas

En las DDBs tenemos varios **módulos** encargados de garantizar las propiedades **ACID**: gestor global de transacciones, local de transacciones, de CC y de recovery. El primero actúa en el sitio de origen de la transacción y le pasa la operación al de CC. Este se encarga de manejar los locks y hasta adquirirlo bloquea a la transacción. Tras obtenerlo, el runtime processor ejecuta la operación y luego libera el lock y le avisa al gestor de transacciones del resultado.

Protocolo 2PC:

En este actúan los gestores de recovery (global y local con su información correspondiente) y el TM del nodo coordinador de la operación a realizar. Se divide en dos fases:

- **Primera fase:** El coordinador le **avisa a todos** los nodos que estén listos para realizar el commit (o abort) y espera recibir la mayoría de aprobaciones para pasar a la siguiente fase (dentro de un tiempo de timeout). Si no lo hace, aborta la operación.
- **Segunda fase:** El coordinador les dice a todos los nodos que va a **ejecutarse el commit (o abort)** y espera su respuesta positiva dentro de un tiempo de timeout (si no cancela la operación). Sin embargo, si el **coordinador falla** tras solicitar el lock, el resto de los nodos pueden quedarse permanentemente bloqueados.

La respuesta a esto es el **protocolo 3PC** en el que se agrega una **fase intermedia** de reconocimiento con tiempo de timeout que causa abort entre la primera y la segunda, y commit entre la segunda y la tercera para estos sitios y evita que se bloqueen.

Gestión de consultas distribuidas

El procesamiento de las consultas se divide en etapas:

1. **Mapeo:** Se pasa la consulta SQL a AR basándose en el esquema conceptual global y se normaliza y reestructura de manera similar al caso centralizado.
2. **Localización:** Se mapea la consulta resultante a múltiples fragmentos individuales en base a la información de distribución y replicación de datos.

3. **Optimización de la consulta global:** Se selecciona una estrategia de una lista de candidatas cercanas a la óptima en base a algún criterio: tiempo, costo total, comunicación, entre otros.
4. **Optimización de la consulta local:** Se ejecuta en todos los nodos de la DDB con técnicas de optimización centralizadas.

Semijoin

En la tercera etapa, un gran costo a considerar es el de la cantidad de datos a transferir. Una manera de mitigarlo es intentar reducir la cantidad de tuplas de una relación antes de la transferencia. Así, el semijoin consiste en **enviar sólo las columnas de la junta**, realizarla y de su resultado retornarla junto con los atributos necesarios para completar la consulta.
Ventaja: Si sólo una pequeña parte de la relación participa en la junta se puede minimizar la transferencia de datos.
Desventaja: Es una heurística y puede fallar.

Tipos de DDBs

Los sistemas de DDBs pueden diferir según el grado de:

- **Homogeneidad:** Si todos los servidores y usuarios hacen uso del mismo software. Aquí la **heterogeneidad** puede surgir por diferencias en:
  - **Modelo de datos:** Pueden diferir en su modelo relacional, objetos o archivos (por ejemplo, la misma información puede ser tratada como un atributo o una relación). Sobre esto es necesario un mecanismo inteligente de procesamiento de consultas que relacione la información basándose en metadata.
  - **Restricciones:** Varían de sistema a sistema y deben ser tratadas por el esquema global.
  - **Lenguaje de consultas:** Dentro del mismo modelo de datos puede tener diferentes versiones.

  También puede haber **heterogeneidad semántica** si hay diferencias en el significado, interpretación y uso de los datos, gracias a la autonomía de diseño. Se puede tratar con **software variado** que administre las consultas y transacciones desde la aplicación global a cada DB y visceversa: middleware, application servers, enterprise resourse planning, modelos y herramientas para la **integración e intercambio de datos** y **acceso/consulta a datos basado en ontologías**.
- **Autonomía local:** Cuánto depende cada sitio del resto para funcionar como un DBMS por separado. Tenemos de varios tipos:
  - **Comunicación:** Capacidad de decidir cuándo comunicarse con el resto.
  - **Ejecución:** Capacidad de un componente de la DB de ejecutar operaciones sin interferir con las del resto y decidir su orden.
  - **Asociación:** Capacidad de decidir si compartir y cuánto su funcionalidad y recursos.

De aquí se derivan **FDBS** y **p2pDBS**, en los que cada servidor es independiente y autónomo con usuarios locales propios, transacciones locales y un DBA (alto grado de autonomía). En el primero hay cierto **esquema global de la federación** de bases de datos, mientras que en el segundo cada nodo se construye **a medida que se lo necesita**. Como sus nodos son heterogéneos, es necesario un **lenguaje canónico** para traducir las consultas.

Arquitecturas paralelas y distribuidas

Dentro de las arquitecturas de **sistemas multiprocesador** tenemos las de:

- **Memoria compartida:** Múltiples procesadores que comparten memoria primaria y tienen almacenamiento secundario (disco).
- **Disco compartido:** Múltiples procesadores que comparten almacenamiento secundario pero comparten su memoria primaria.
- **Shared-nothing:** Cada procesador tiene su propia memoria y disco. Se comunican a través de redes de alta velocidad. Se busca cierta simetría y homogeneidad entre los nodos.

De estas se derivan tipos particulares de arquitecturas:

- **Parallel DBMS (PDBMS):** Son un caso particular de las distribuidas usadas en la industria donde los nodos se encuentran **físicamente cerca** y se conectan entre sí a través de una **red local** de alta velocidad. Esto hace que su costo de comunicación sea bajo y puedan implementarse con los 3 tipos mencionados de arquitectura multiprocesador. Su principal diferencia con las distribuidas reside en su **modo de operación**.

- **Distribuidas puras:** Cada nodo está en un **sitio diferente** y se comunican a través de una **red** con un costo no despreciable. Se basan en la arquitectura **shared-nothing** y tienen un esquema conceptual global y uno interno a cada nodo que junto con el catálogo permiten imponer restricciones y optimizar las consultas locales y globales.
- **3-Tier:** Se desarrollan en el contexto de arquitecturas cliente-servidor y se separan en 3 capas:
  - **Presentation:** La interfaz que interactúa con el usuario.
  - **Application:** La encargada de manejar la lógica de negocios de la aplicación.
  - **Database Server:** La encargada de manejar, procesar y retornar los pedidos de consulta y actualizaciones de la capa de aplicación.

  Aún así, las funcionalidades DDBMS puede dividirse de diferentes maneras. Más en detalle, la capa de aplicación debe primordialmente encargarse de:
  - Generar un **plan de ejecución distribuido** para las consultas y transacciones y supervisar su ejecución.
  - Asegurar la **consistencia** entre réplicas con técnicas de control de concurrencia distribuida.
  - Asegurar la **atomicidad** de las transacciones globales, ejecutando un recovery global en caso de producirse una falla.

  Esta capa no siempre puede operar con transparencia de distribución, es decir, sin especificar los sitios en que residen los datos de las consultas o transacciones.

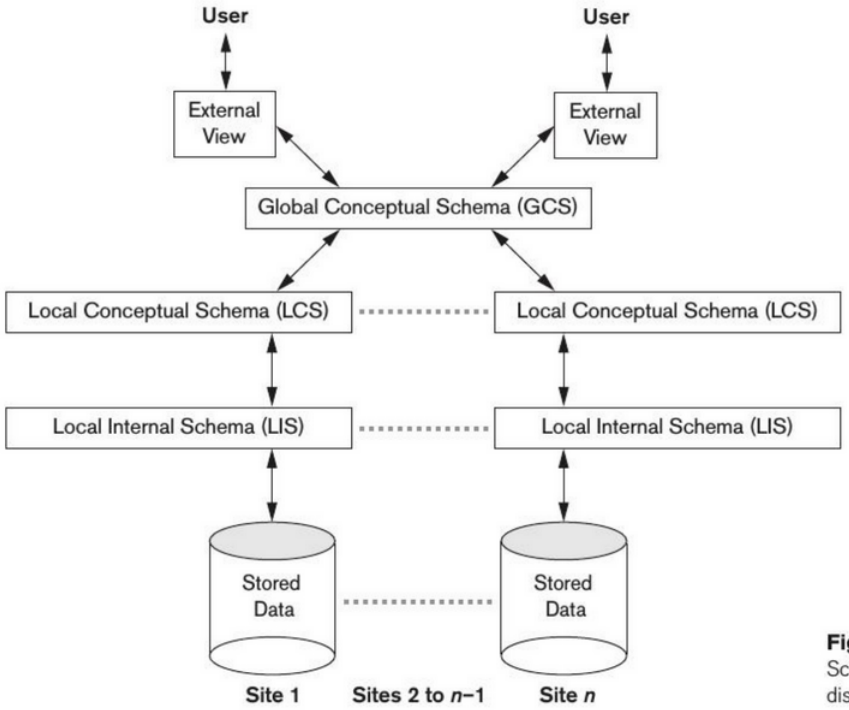


Figure 23.8
Schema architecture of distributed databases.

Esquema Arquitectura Base de Datos Distribuída Pura

Figura de Elmasri/Navathe-Fundamentals of DB Systems, 7th Ed., Pearson, 2015

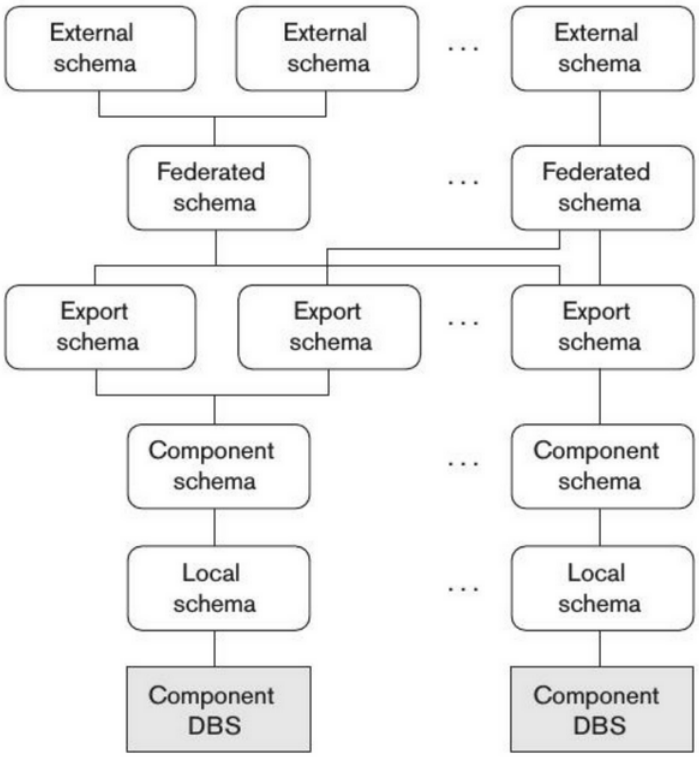


Figure 23.9
The five-level schema architecture in a federated database system (FDBS).

Source: Adapted from Sheth and Larson, "Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases." *ACM Computing Surveys* (Vol. 22: No. 3, September 1990).

Esquema de Arquitectura de un Sistema de Base de Datos Federada

Figura de Elmasri/Navathe-Fundamentals of DB Systems, 7th Ed., Pearson, 2015

Catálogo

Es una DB que contiene la **metadata** del DDBMS, particularmente la información de la **fragmentación, asignación de fragmentos y replicación de los datos**. Su administración debe tener autonomía de sitio, vistas y distribución y replicación de los datos. Su **esquema** puede ser:

- **Centralizado:** Se almacena por completo en un sitio.
  Ventajas: Su implementación es sencilla.

Desventajas:

- Tiene poca confiabilidad, disponbilidad y autonomía.
- Centraliza la distribución de la carga de procesamiento.
- Los locks pueden producir un cuello de botella en caso de haber muchas escrituras.

- **Totalmente replicado:** Cada sitio tiene una copia completa del catálogo.
  Ventajas: Puede responder a las consultas localmente.
  Desventajas: Las actualizaciones deben ser transmitidas a todos los sitios, por lo que se requiere de un esquema centralizado 2PC para mantener la consistencia y las aplicaciones de muchas escrituras, incrementando así el tráfico de la red.
- **Parcialmente replicado:** Cada sitio tiene la información completa del catálogo de los datos almacenados localmente en el sitio. Además pueden almacenar en caché copias de entradas de otros sitios (no necesariamente actualizadas). Luego, el sistema debe registrar para cada entrada del catálogo los sitios en que se creó el objeto y en donde están sus copias para propagar los cambios de ellas al original.

Datos semi-estructurados y XML

Los datos estructurados no fueron muy compatibles con el alza de la web, **HTML**, y otros.

Los datos estructurados tienen un formato estricto, fijo y predefinido pueden relacionarse con otros datos en registros de la misma estructura.

Hacer traducciones constantes de **HTML** a lenguajes estructurados tiene el problema de volver **frágil** y propenso a que **deje de funcionar** los programas, y requerir HTTP requests incluso cuando solo se requiere un valor de una tabla.

Los **datos semi-estructurados** tienen cierta estructura pero **no garantizan la predictibilidad y organización** de los **estructurados**. Aún así, **los semi-estructurados**, tienen marcas y otros elementos que les permite jerarquizarlos, pero por tanto **la definición de la estructura esta mezclada con los datos**.

`XML, eXtensible Markup Language` es un lenguaje parecido a **HTML** pero no igual, que permite intercambiar datos con cierta estructura **(semi-estructurado)** como **JSON** o los **csv**. **Permite representar datos más amigables** para el formato Humano-Computadora, con **la desventaja** que **hay mucha redundancia** y **repetición de los datos a diferencia de los estructurados**.

DTD (Document Type Definition) y **XML Schema** permiten definir estructuras en **XML** con el objetivo de asegurar que los archivos **XML** recibidos o enviados conforman un cierto esquema, **permiten además chequeos automáticos que verifican que el archivo cumple con dicho esquema**.

XPath es un lenguaje de recuperación y consulta de archivos **XML**, utiliza jerarquización en base a directorios como una estructura de carpetas.

XQuery es un lenguaje puramente de consultas, que toma conceptos de **SQL**, **XQL** y otros lenguajes de consulta para **XML**.

Además, existen extensiones para **SQL** que permiten tratar con archivos **XML** como por ejemplo la extensión de **Oracle** (y así poder usar **datos semi-estructurados** en **entornos estructurados**).

Mapeo Objeto Relacional (ORM)

Estos se enfocan en combinar las ventajas de los **modelos relacionales** (consultas de alto nivel) con los **orientados a objetos** (tipos de datos complejos como mapas o multimedia). Soportan:

- **Tipos de datos complejos:** Permiten atributos con dominio no atómico para tener un modelo más intuitivo para aplicaciones los manejen. Esto hace que violen **1FN** aunque mantienen los **fundamentos matemáticos** de la relación. Algunos de ellos son:
  - **Colecciones o conjuntos:** Particularmente permiten definir **relaciones anidadas** a través de collect o subconsultas. Soportan "desanidación".
  - **Estructuras:** Particularmente las definidas por el usuario.
- **Funcionalidades orientadas a objetos:**
  - **Herencia:** Permiten definir tipos en base a otros y los subtipos pueden redefinir métodos de su supertipo (o supertipos con cuidado de no generar conflictos) en su declaración.

- **Referencias:** Apuntan a identificadores de otros objetos y permiten evitar consultas adicionales y juntas a través de expresiones (**path expressions**).

Hay extensiones de SQL que comprenden estas funcionalidades pero no están totalmente implementadas en sistemas actuales (algunas son de uso comercial).

NoSQL

Este tipo de bases de datos **no relacionales** fueron diseñadas a partir de la existencia de datos no estructurados. Se caracterizan por ser **distribuidas**, de código abierto, y al no tener un esquema definido se basan en el concepto de **clave/valor**. Aquí almacenan los valores provistos para cada clave y los distribuyen a lo largo de la base.
Ventajas:

- Suelen tener una interfaz sencilla.
- Pueden albergar grandes cantidades de datos con escalamiento horizontal (no tanto vertical).
- Pueden recuperar su datos fácilmente gracias a su distribución.
- Sus datos pueden ir evolucionando a través del tiempo y convivir con datos de diferentes estructuras.
- Consultas que anteriormente eran complejas a través de juntas u otros operadores pueden responderse rápidamente.

Desventajas:

- No permiten realizar consultas muy complejas.
- Su variedad de opciones dificulta saber la más adecuada para cada caso.
- Carecen de operaciones de juntas, agrupamiento y ordenamiento.
- Su soporte es limitado.
- No tienen un lenguaje de consultas definido.
- Su administración de transacciones tiene poca garantía.

Herramientas y operaciones

- **MapReduce:** Es un marco de trabajo que permite **procesar paralelamente** grandes volúmenes de datos en varios equipos. Parte de la operación **Map** que toma los datos de la DB y los transforma en una colección de operaciones que se pueden ejecutar independientemente en diferentes procesadores. A la salida retorna pares `<clave, valor>` a los que **Reduce** les aplica la operación asignada y retorna su resultado.
- **Sharding:** Se basa en **particionar** a los datos de la DB en fragmentos (shards) que se distribuyen a través de servidores. Estos comparten su esquema haciendo que su unión represente la totalidad del dataset. Nótese que esta fragmentación no es trivial y suele hacerse en base al hash de alguno de sus aributos.
  Ventajas:
  - Permite salvar la imposibilidad de guardar todos los datos en la misma máquina.
  - Escala horizontalmente.
  - Brinda tolerancia ante fallas, ya que sólo una porción de los datos quedaría fuera de servicio.

  Desventajas: Ante frecuentes consultas que involucren más de un nodo, su rendimiento decrece.
- **Replicación:** Es el almacenamiento de **múltiples copias** de la base de datos en diferentes nodos de la red. Puede combinarse con sharding para tener escalabilidad y disponbilidad. Se puede implementar en modo Master-Slave o P2P (Peer to Peer):
  - **Master-Slave:** Este designa a un nodo como **master** y sobre él efectúa las operaciones de actualización (insert, delete, update), mientras que el resto (**slaves**) se usan mayormente para lecturas (cada tanto se actualizan).
    Ventajas:
    - Son ideales para escenarios de muchas lecturas.
    - En caso de fallar el nodo master, las lecturas pueden continuar en los slaves y se los puede configurar para que lo reemplacen.

    Desventajas:

- El nodo master puede representar un cuello de botella para las escrituras.
- Las lecturas pueden ser inconsistentes según la frecuencia en que se actualice cada réplica. Para ello puede implementarse un sistema en el que un valor se considere consistente al estar en la mayoría de los nodos, pero este requiere de una comunicación más rápida y confiable entre ellos.

- **Peer to Peer (P2P):** Aquí todos los nodos (**peers**) poseen el mismo nivel de jerarquía y pueden manejar tanto lecturas como escrituras.
  Ventajas:
  - Evitan los cuellos de botella al tener un sistema descentralizado y tienen mayor tolerancia a las fallas.
  - El rendimiento es potencialmente mayor ya que todos los nodos son capaces de responder a las peticiones.

  Desventajas: Las lecturas pueden ser inconsistentes y las escrituras, conflictivas. Tenemos dos posibles estrategias para tratar estos problemas:
  - **Concurrencia pesimista:** Uso de locks que asegure la consistencia y prevenga conflictos a pesar de ir en contra de la disponibilidad.
  - **Concurrencia optimista:** No uso de locks, lo cual puede generar inconstencias basándose en la propagación de los valores hasta llegar a un estado consistente. Para asegurarse de que no sucedan suele combinarse con un sistema de votación parecido a master-slave.

OLTP

Es un tipo de procesamiento de Datos. En la mayoría de los casos que vimos (**Relacionales**) y verémos a continuación de tipos de DBMS, se piensa en modelos de procesamiento de **On Line Transactional Processing** (no obstante, algunos DBMS **NoSQL**, están pensados para un uso **OLAP** o mixto **OLTP/OLAP**). Están pensadas para las bases de datos que deben soportar la "operación", el "día a día". Buscan poder resolver muchas transacciones que puedan seleccionar, insertar y modificar datos eficientemente (a grandes rasgos).

OLAP

On Line Analitical Processing es otro tipo de procesamiento de datos, dónde las Bases de Datos que cumplen con tener un tipo de procesamiento de datos **OLAP**, buscan hacer análisis estadístico, con datos sumarizados, de acceso poco frecuente, desnormalizados, por columna para cálculos estadísticos rápidos y eficientes. Ejemplos de esto son las Bases de Datos **Column Store** (notar que no son lo mismo que las **Column Family**).

Tipos de bases NoSQL

Los tipos de bases NoSQL más comunmente usados son los basados en: key-value pair (pares clave-valor), document (documentos), column family (grupos de columnas) e graph (grafos).

Se estima que las bases de datos Key-value son las que son capaces de almacenar los volúmenes mayores de datos, seguidas de las ColumnFamily, las de Documentos, las de grafos, y finalmente las SQL.

No obstante la complejidad de los datos modelables por bases de datos de Grafos, es mayor a la de documentos, seguida de las de Column Family, y las Clave-valor tienen mucha dificultad para modelar relaciones complejas de los datos.

Key-value pair databases

Es la más sencilla ya que guarda cada ítem como una **clave asociada a un valor** (como un diccionario). Las claves deben ser **únicas** dentro del dominio manejado para identificar al ítem unívocamente (para esto se pueden tener namespaces y/o buckets).
Los datos guardados constituyen su información y no tienen un esquema definido, por lo que **no tienen restricciones** en cuanto a tamaño ni tipo: texto plano, XML, JSON, imagen, etc. Esto no salva que pueda haber datos redundantes y no implementan integridad referencial.
Ventajas:

- Su escalabilidad hace que se adecúen a la nube, servicios en la web y aplicaciones móviles, ya que no es necesario consultar todas sus entradas para obtener un valor.
- Permiten almacenar objetos variados al asociarlos a claves. Esto las hace útiles para el código orientado a objetos.

Desventajas:

- La integridad de los datos y sus restricciones deben ser manejadas por los programadores.

- La falta de estructura limita el análisis posible que pueda efectuarse sobre los datos.
- Su enfoque en la escalabilidad limita su funcionalidad y complejidad.

Document databases

Se asemejan a las anteriores pero guardan sus datos en base a **documentos** (strings o representaciones binarias de strings). Estos pueden diferir en su estructura (semi-estrcuturados, JSON, XML) y pueden contener listas de atributos e incluso **documentos embebidos**. Entre sus datos pueden contener metadata con índices para referirse a sus campos. Suelen manejarse a través de motores de DB como **MongoDB**. Este tiene escala horizontal gracias al sharding y replicas. Además en su API tiene incorporadas las funcionalidades de MapReduce para procesamiento batch y las de agregaciones. Permite además implementar búsquedas ad-hoc por campos y rangos particulares. Otros motores incluyen RavenDB, RethinkDB y CouchDB.
Ventajas:

- Los documentos pueden representar flexiblemente entidades, ya que no requieren la definición de sus atributos previamente.
- Su información es accesible a través de lenguajes de consultas y APIs dedicados, a diferencia de key-value donde es necesario acceder a la clave previamente.
- La incrustación de documentos evita implementar juntas en las consultas, mejorando el rendimiento. Esta pueden especificarse a través de su esquema lógico dado por su **diagrama de interrelación de documentos (DID)**.

Desventajas: La flexibilidad de la información que maneja puede acomplejizar la interpretación de los datos en un programa.

Column family databases

Almacenan sus datos en **columnas** que asocian un nombre con un valor. Estas se agrupan en **filas** (row) y las que suelen accederse simultáneamente en las consultas se agrupan en **familias** (column families). No requieren predefinir un esquema para agregar columnas nuevas. Además, tienen mecanismos naturales de partición vertical y su almacenamiento es multi-dimensional (mapas o vectores asociativos).
Superficialmente se asemejan a las **bases de datos relacionales**, pero con algunas diferencias:

- No implementan juntas ya que su set de columnas no está predefinido.
- Suelen estar denormalizadas de manera de poder guardar en una fila toda la información relacionada con una entidad particular.

Ventajas:

- Permiten replicar y distribuir sus datos fácilmente en múltiples nodos de la red.
- Tienen un modelo de fácil acceso a través de lenguajes de consulta parecidos a SQL.
- Las column families favorecen el rendimiento de las consultas de agrupamiento (máximo, mínimo, promedio, etc.).
- Su escalabilidad ser ve favorecida al permitir que no todas las columnas tengan valores asociados.
- Pueden usar timestamps para almacenar diferentes versiones de un valor en el tiempo.

Desventajas:

- No son adecuadas para datos relacionales.
- No proveen consistencia inmediata (son BASE).

Graph databases

Estas almacenan sus datos como **nodos** conectados a través de **relaciones dirigidas**. Ambas estructuras pueden contener información compleja. SQL Server y Oracle agregaron funcionalidad basada en ellas.
Ventajas:

- Ejemplos como **NEO4J** cumplen con garantías **ACID**, a diferencia de muchas otras opciones NoSQL.
- Permiten almacenar relaciones y manejarlas eficientemente al pensar a cada nodo como una entidad.
- Se adecúan a los problemas enfocados en relaciones y basados en **grafos** (redes sociales, conexiones, recorridos, etc.) que en las bases relacionales tendrían el costo de muchas juntas y consultas.

- En el caso de **NEO4J**, este incluye capacidades para sharding (fragmentación) y replicación. La replicación usa un sistema de nodos **"Core"** y nodos **"Read Replica"**. Una escritura es confirmada cuando todos los servidores Core la confirman. Por tanto da posibilidades de escalabilidad y alta disponibilidad.

Otras bases de datos

Stream databases

Son bases utilizadas para situaciones en que **los datos se cargan continuamente** (páginas web, sensores, trayectorias de paquetes). Se manejan a través de un **DSMS** que al recibir las consultas responde respecto de los datos que van ingresando y están disponibles por un período corto de tiempo. Aquí el **stream** es un conjunto de pares `<s, t>` donde s es una tupla y t un timestamp con su tiempo de llegada.
Para contemplar la **validez** de los datos en las consultas se impone una restricción sobre la parte del stream a procesar. Esta es la **ventana** y se puede expresar diferente según la implementación del lenguaje:

- **Landmark:** Mantiene fija la parte más antigua y la más nueva avanza conforme llegan tuplas al stream.
- **Sliding:** Ambos extremos de la tupla van avanzando a medida que llegan tuplas nuevas.

Spatial databases

Se enfocan en los **Tipos de Datos Espaciales (SDT: Spatial Data Types)**, como puntos, líneas, polígonos, etc. Proporcionan indexación espacial y algoritmos eficientes de unión espacial y distancias entre cualquier tipo de par de **SDTs** usados. Comprenden a las **bases de datos geográficas**. Sus consultas suelen incluir:

- Objetos de un cierto tipo en cierto rango.
- Vecino más cercano de un cierto tipo de objeto.
- Intersección o superposición entre 2 objetos.

PostGIS es un ejemplo de una extensión, que convierte a **PostgreSQL** en una Base de Datos Espacial. Por tanto responde queries **SQL**, es **ACID**, y resuelve los problemas antes mencionados.

Cloud DataBases (CDBs)

Las Cloud Data Base Management Systems (CDBMSs) son DBMSs distribuidas que proveen todo el paquete incluyendo el poder de computo accesible como un servicio desde internet, es por esto y dado que no se almacenan in situ como otras DBMSs que los datos se guardan encriptados. Se paga por tiempo, espacio físico y poder de computo. Hay versiones relacionales y no relacionales.

Ventajas:

- Muy fáciles de escalar, generalmente solo implica solicitar una mejora del servicio obtenido.
- Reduce el costo de tener un DBMS, especialmente para pequeños y medianos usuarios. Para grandes usuarios a largo plazo dependerá de más factores.
- La tolerancia a fallas tiende a ser de muy alta calidad y a veces personalizable.

Desventajas:

- La seguridad puede ser buena pero fuera de nuestro alcance garantizarla.
- Hay dependencias legales en los datos que pueden impedirnos usar CDBMSs para ciertos datos.

Multi Tenant DataBases

Si bien no es un tipo de Base de Datos por si misma, es un concepto que puede definir algunas Bases de Datos, especialmente en los modelos de Bases de Datos como un Servicio (DBaaS).

Puede tratarse de una Base de Datos por cliente, con algún recurso compartido (Database-per-tenant/Multi-Database). Una Base de Datos para varios o todos los clientes, pero separadas por schemas distintos para cada uno (Multi-Schema). O un único schema global que es usado por todos los clientes en la misma Base de Datos compartida (Shared-Schema).

In-memory databases

Son bases de datos **cargadas completamente en memoria**. Existen de dos tipos:

- **Sin persistencia:** Un corte en el suministro de energía hace que se pierda toda la base (ej: MemBase).
- **Con persistencia** (ej: Redis, Sap Hanna).

Aquí todas la interacciones con la base se resuelven en memoria y se **escribe continuamente al log**, aprovechando que es secuencial para que las escrituras físicas sean rápidas.
Pueden almacenarse por **filas** o por **columnas**. Las primeras son útiles para operaciones de selección mientras que las otras para proyección y ocupan menos espacio.

Volt DB es un ejemplo de **IMDB** que cumple **ACID**, responde a queries escritas en **SQL**, y asegura durabilidad a través de command logging y replicas.

Redis también es una **IMDB** con servicio de servidor (Master) y cliente (Slave). Es útil porque funciona también como Base de Datos de Streams y de datos Geoespaciales, y puede observar cambios de un valor por medio de un comando `WATCH`.

Column Store databases

No confundir con las **Bases de Datos Column Family**. Las **Bases de Datos Column Store** pueden ser Bases de Datos Relacionales como las SQL convencionales, con la diferencia que guardan los contenidos de las tablas por columna y no por fila. Esto las hace mucho más adecuadas para procesamientos analíticos como los procesamientos **OLAP** (ya que operaciones analíticas como agregaciones que requieren pocas columnas son mucho más eficientes (proyecciones mucho más baratas), pero a costa de que los procesos **OLTP** sean más costosos (selecciones, y en particular modificaciones e inserciones en tablas son muchísimo más costosas).

Vertica Database es un ejemplo de CSDB que cumple **ACID** y responde a queries escritas en **SQL**.

Resorce Description Format (RDF) management systems

Comprende los datos **explícitamente declarados** y los **implícitos** generados a través de **restricciones semánticas** (reglas de entailment). Las consultas entonces se pueden resolver convirtiendo todos los datos en explícitos o reescribiendo la consulta de manera que contenga las expansiones (a través de reglas). Ejemplo:

Explícito	Implícito
a) Sócrates es humano	Sócrates es mortal
b) Los humanos son mortales	

Mobile databases (Opcional, no evaluado en 2022)

Son las enfocadas en el creciente uso de **dispositivos móviles**, ya que pueden instalarse en uno a través de una red móvil. Aquí el cliente y el servidor se conectan de forma **inalámbrica** y la memoria caché se usa para almacenar datos frecuentes y transacciones para que no se pierdan por un fallo de conexión.
Sus aplicaciones se clasifican en:

- **Verticales:** Los usuarios pueden acceden a los datos en una celda específica y fuera de ella la información no está disponible (sistema de plazas libres en un estacionamiento).
- **Horiontales:** Los datos se distribuyen por todo el sistema y los usuarios pueden acceder a ellos desde cualquier celda (acceso al correo electrónico).

Los DBMS móviles deben además proveer la capacidad de:

- Comunicarse con el servidor centralizado de la DB a través de comunicación inalámbrica o acceso a Internet.
- Replicar y sincronizar los datos en el servidor centralizado y el dispositivo móvil.
- Responder a una consulta de acuerdo con la **localización** del dispositivo móvil (ya sea, el lugar del que se efectuó la consulta, el lugar donde terminó o el indicado por ella). La movilidad de las estaciones de acceso puede dificultar la optimización por el coste de comunicación involucrado (más aún si los datos residen en una estación móvil). A esto se le suma la variabilidad del ancho de banda.

Si bien la tecnología móvil puede ser de gran provecho para negocios y empresas por optimizar la productividad, racionalizar las operaciones y crear nuevas fuentes de ingreso, los datos accesibles de manera remota pueden ser críticos a la corporación. Por ende, su **estrategia de seguridad** debe contemplar las maneras de gestionar y garantizar su seguridad en cualquier sitio y hora. Para ello se aplica la **autentificación de usuarios** cada vez que se quiera acceder a una nueva capa de confidencialidad y funcionalidad de los sistemas corporativos. De no estar autorizado, el acceso debe ser denegado.

Big Data

Big Data se los define a los datos que cumplen con las **5 V's** en mayor o menor medida.

- **Volumen:** presentan un gran **volúmen** de datos.

- **Velocidad:** accesibles a **velocidades** que permiten el acceso en cuestión de milisegundos (cómo es el caso de los mercados bursátiles).
- **Variedad:** con datos de gran **variedad**, de muchos entornos distintos y diversos, que pueden estar estructurados o no, venir de paradigmas varios, e integrar muchos modelos.
- **Veracidad:** donde los datos pueden ser poco confiables, ruidosos, mezclar orígenes de datos que al combinarlos, su output puede generar incertidumbre, y que por tanto es necesario tratarlos, analizarlos, categorizarlos según fiabilidad, con el fin de poder corroborar con algún criterio su **Veracidad**.
- **Valor:** teniendo en cuenta que los datos proveen valor intrínseco al permitir tomar decisiones eficientes y precisas que le otorgan **Valor**.

OpenData

Es un **movimiento** que apoya la **difusión de los datos** por parte de los Estados, organismos internacionales y empresas para promover su análisis y utilización en **formatos abiertos** para crear visualizaciones, aplicaciones y herramientas.
Los datos son publicados con dos motivos principales:

- Transparencia y participación ciudadana (open goverment, repercutió en América Latina)
- Generación de servicios para empresas y de valor para iniciativas privadas (repercutió en Europa)

Las publicaciones son públicas o semi-públicas y son accedidas por parte del periodismo, ongs, emprendedores y entidades académicas o estudiantiles. El acceso se considera **abierto** si al estar a disposición del público los datos cumplen con los siguientes principios:

I **Completo:** Todos están disponibles, sin limitaciones de seguridad o privilegio.

II **De primera fuente:** Son coleccionados desde la fuente, con el mayor grado posible de granularidad, sin ningún tipo de modificación o agregación.

III **En tiempo:** Están disponibles tan pronto como sea necesario para preservar su valor.

IV **Accesible:** Están disponibles para el rango más amplio de usuarios y propósitos.

V **Procesables por computadoras:** Están estructurados razonablemente como para permitir su procesamiento automático.

VI **No discriminatorios:** Están disponibles para todos, sin necesidad de registro (pueden ser accedidos de forma anónima, incluso a través de proxies anónimos).

VII **No propietario:** Tienen que tener un formato del que ninguna entidad tenga control exclusivo.

VIII **Sin licencia:** No deben estar sujetos a derechos de autor, patente, etc.

Administración de datos

Calidad de datos

Los **datos registrados** en una base pueden ser de varios tipos:

- **Registros:** Cada valor es un registro y se puede almacenar en una **matriz** (como conjunto fijo de atributos), **documento** (vector de términos de significado ámplio) o ser parte de una **transacción** (involucrando a un conjunto de ítems).
- **Semi-estructurados:** Representados a través de **XML** (lenguaje de marcación jeráquico muy utilizado para el intercambio de información) o **JSON** (similar al anterior pero con otras funcionalidades).
- **Grafos:** Enfocados en problemas basados en grafos o redes.
- **Ordenados:** Su criterio de orden puede depender de una secuencia, el espacio o el tiempo (**stream:** cuando estos fluyen continuamente con diferentes velocidades).

Ahora, teniendo los datos guardados suele presuponerse que estos son **correctos** o que su **validez** es eterna. La **calidad de datos** nos dice que es necesario **monitorearlos permanentemente**, con dinero y/o esfuerzo para que eso pase. Sobre esto es que residen una serie de posibles errores en los datos:

- Fuera de rango
- Falta de estándar

- Invalidez
- Diferencias culturales
- Discrepancias en el formato
- Cosméticas
- Inconsistencias provenientes de metadata
- ...

Para analizar la calidad tenemos varias opciones:

- **Análisis univariado:** Obtener el valor mínimo, máximo, media, mediana, moda, histogramas y demás datos estadísticos de cada variable.
- **Análisis bivariado:** Obtener el coeficiente de correlación, tablas de contingencia, diagramas de dispersión y demás entre pares de variables.
- **Perfilado de los datos:** Analizar la información en cada sitio y buscar inconsistencias.

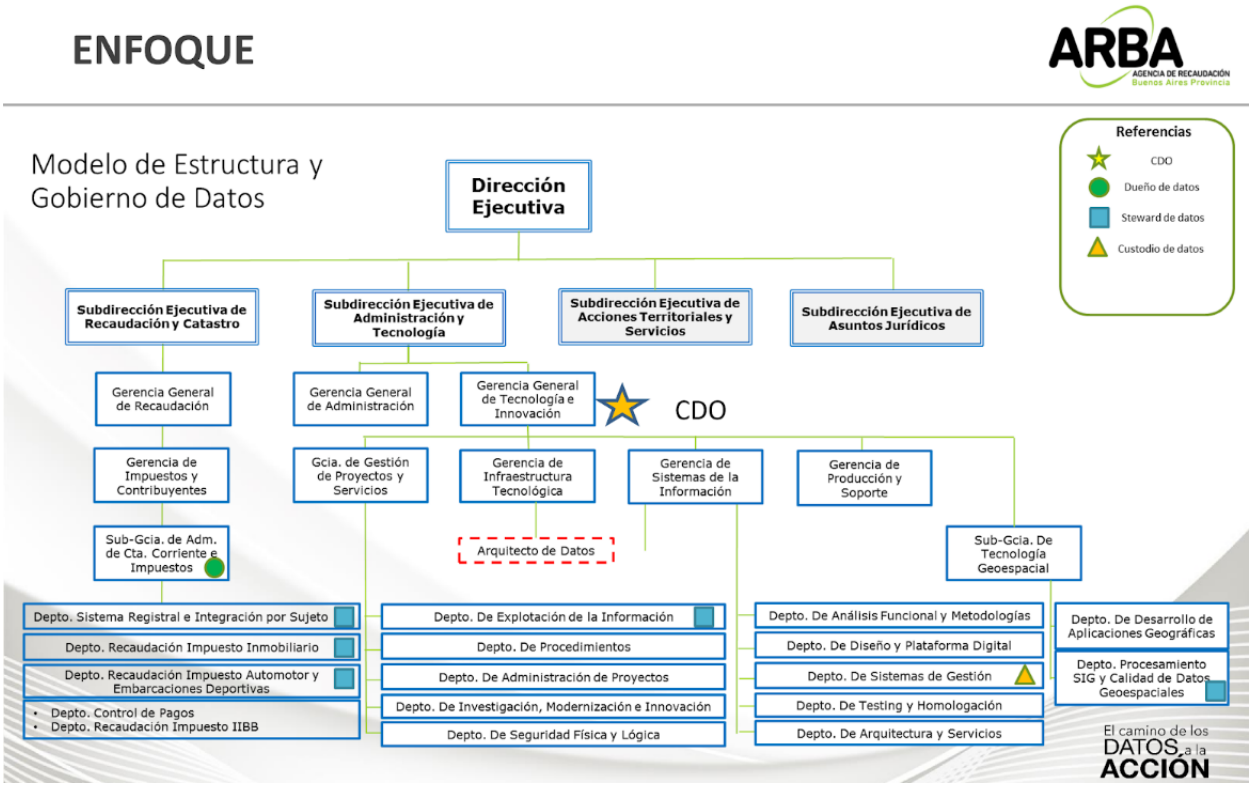
Gobierno de datos

Definición: Según la DAMA (Data Management Association) y la Data Resource Management, es el "*desarrollo y ejecución de arquitecturas, prácticas y procedimientos que manejan adecuadamente las necesidades del ciclo de vida de datos de una empresa*". Esto incluye aspectos de calidad, arquitectura, seguridad y metadata de los datos y comprende a **toda la organización**, no sólo al sistema en sí por ser los datos un **activo** de ella.
En su implementación conviene empezar con un objetivo poco ambicioso para mostrar su utilidad y luego ir incrementando su **nivel de madurez**. Nótese que puede necesitarse apoyo económico.
Nivel de madurez:
El nivel de madurez del gobierno de datos se divide en:

- **Indisciplinado:** Las decisiones de negocio dependen de la tecnología, los datos pueden ser inconsistentes o duplicados y hay poca flexibilidad para mantener los cambios de negocio.
- **Reactivo:** El negocio influye sobre las decisiones de tecnología, la información es redundante y poco controlada y hay un alto costo en mantener múltiples aplicaciones.
- **Proactivo:** Los equipos de negocio y tecnología trabajan de manera colaborativa y los datos son un activo de la compañía.
- **Gobernado:** Los modelos de negocio definen las decisiones tecnológicas, hay procesos estandarizados para definir la gestión de los activos de los datos, las decisiones corporativas se toman con datos certeros y se obtienen beneficios por la aplicación del programa de gobierno.

Roles principales:
Hay una serie de roles involucrados en el gobierno de datos:

- **Chief Data Officer:** Máximo responsable del programa y lider del equipo. Encargado de definir y/o colaborar en las iniciativas del gobierno de datos, promoviendo, negociando y justificando cambios en la estrategia de datos corporativa.
- **Arquitecto de datos:** Desarrollador de la arquitectura de datos para atender a los requerimientos de negocio. Encargado de desarrollar estándares y procedimientos de diseño y modelado, supervisarlos para cada componente y aprobar las características de desarrollo de aplicaciones e interfaces que afecten la arquitectura.
- **Data owner:** Máxima autoridad de aprobación respecto de los riesgos de gobierno en su dominio. Gestiona el ciclo de vida de los datos con sus permisos de acceso, calidad y riesgos, colabora en el gobierno de datos y conoce su significado.
- **Data steward:** Quienes apoyan a los anteriores al comprender sus procesos de negocio y datos producidos. Responsables de escribir e implementar reglas de calidad de datos, atender a sus problemas y escalarlos de darse el caso. Tiene responsabilidades concretas y puede efectuar acciones en nombre del dueño para liberar el flujo.
- **Custodio de datos:** Soporte en las áreas de bajo nivel y de telecomunicaciones de las plataformas, sistemas y aplicaciones en que residen los datos de los anteriores. Pueden tener cierta responsabilidad operativa y se encargan de mantener la integridad y seguridad de los datos, cumpliendo con las políticas del programa.



Administrador de datos

Es una persona (o conjunto de ellas) responsable de **administrar** los datos de manera **funcional o lógica**. Se diferencia del **DBA** en que el segundo es **especialista** en el motor de la DB.
Sus tareas principales son:

- Recolectar y analizar los requerimientos, modelando el negocio en base a ellos (conceptual y lógico).
- Definir estándares sobre los datos y asegurar su cumplimiento.
- Conducir sesiones de definición de datos.
- Manejar y administrar repositorios de metadata y herramientas de modelado.
- Asistir al DBA en la creación de modelos físicos a partir de los lógicos.

Nótese que la **definición de los datos** suele encontrarse en dos lugares desde el punto de vista del negocio:

- **La mente de las personas:** Si son reglas no escritas existentes en todas las áreas que interactúan con ellos. Estas son vulnerables a baja calidad por falta de consistencia o confianza.
- Los modelos de los datos: Se representan a través de las herramientas de modelado pero suelen reflejar sólo el estado final y no los cambios.

Privacidad

Es una preocupación creciente con numerosas **regulaciones** locales e internacionales al respecto. Se centra en garantizar la **protección de los datos** para todos sus usuarios más allá de dónde estén. En el ámbito local (Argentina) existen numerosos secretos estadísticos, fiscales y educativos, más allá de haber una agencia de acceso a la información pública y una dirección nacional de protección de datos personales.

Arquitectura de Datos

La arquitectura de datos es un conjunto de especificaciones que ayudan en la estandarización de como una organización recibe, almacena, transforma, distribuye y usa los datos. Esto ayuda en las ïnversiones en datos".

Características de una Buena Arquitectura de Datos

Una buena arquitectura de datos debe ser:

- **Colaborativa:** todas las áreas deben colaborar.

- **Administrada:** Buena gobernanza.
- **Simple:** Disminuir variedad de herramientas usadas, duplicación de datos, y facilitar su modos de uso.
- **Elástica:** Poder manejar demandas crecientes y cambiantes de su uso.
- **Segura:** Buenas políticas de seguridad.
- **Resiliente:** que tenga alta disponibilidad y capacidad de recuperación ante fallas.

Data as a Service (DaaS)

Data as a Service (DaaS) es una estrategia de administración de datos en la que una compañía renta su capacidad de almacenamiento, integración, procesamiento y servicios de analítica a otras compañías o terceros a través de un servicio en la nube.

Integraciones de Bases de Datos

Las bases de datos pueden ser integradas unas a otras por razones como fusiones de compañías o federaciones. Herramientas para facilitar el proceso pueden seguir estrategias:

ETL: Extraction - Transform - Load

ELT: Extraction - Load - Transform

La fase que se haga segunda tiene prioridad por sobre la tercera.

ETL transforma los datos en una staging area antes de cargarlo a la Data Warehouse donde se almacenará para su análisis.

ELT en cambio, lo carga y transforma en la misma Data Warehouse, haciéndose allí su futuro análisis, permitiendo usar el resultado y transformarlo nuevamente en el futuro sin requerir nuevas cargas, pero cualquier carga de datos nuevos va a requerir una nueva transformación.

Federaciones y Fusiones como arquitectura de Datos

Las Federaciones construyen un esquema virtual global que sabe consultar con mapeos predefinidos a cada base de datos integrada a la Federación, y con ello responde las queries que se le indiquen. No obstante no guarda ningún dato. Es útil para consultas a Bases de Datos diversas que fueron desarrolladas de formas distintas y que necesitan cooperar para un mismo fin, como por ejemplo, una federación de aerolíneas que quiere publicar los vuelos de muchas aerolíneas (despegar.com), o una federación de hoteles que desea publicar la disponibilidad de distintos hoteles con distintas Bases de Datos (hoteles.com).

En cambio **las fusiones o adquisiciones** de compañías tienden a definir mapeos que transforman los datos de la compañía adquirida generalmente a la estructura de la compañía que la adquirió, la cual además alojará copia de dicho datos transformados a la estructura de la compañía que realizó la adquisición.

Data Mining

Es la **extracción** de patrones o **información interesante**, es decir, no trivial, implícita, previamente desconocida y potencialmente útil, de grandes bases de datos (como data warehouses). Su resultado es **el KDD** (Knowledge Discovery in Databases o Descubrimiento de Conocimiento en Bases de Datos).

Outliers

Es importante tener datos de Calidad para estos algoritmos, y el tratamiento de outliers es muy importante. Dentro del conjunto de datos analizado los outliers siguen un comportamiento diferente al resto en una o más variables. Si bien se los puede usar para **detectar anormalidades**, generalmente pueden ensuciar o influir en nuestros resultados y conviene descartarlos. Esta **variación** puede provenir de:

- **La fuente:** Surge de las observaciones y se la considera un **comportamiento natural** en relación a cierta variable de estudio.
- **El medio:** Surge del mal uso de la técnica para medir una variable o cuando no exista una valoración exacta de ella. Comprende los **redondeos forzosos** en variables continuas.
- **El experimentador:** Se clasifican en:
  - **Error de planificación:** Cuando no se delimita correctamente la población o se realizan observaciones de otra.

- **Error de realización:** Se valora incorrectamente a los elementos (transcripciones erróneas, falsas lecturas con los instrumentos de medición, etc.).

De allí se dice que una **observación atípica** surge de primer tipo de variación, mientras que una **errónea** de los otros dos. Ambas pueden ser outliers y es conveniente estudiarlas antes de eliminarlas.

Otro concepto importante es la estimación de la precisión del modelo. Para ello se debe construir un conjunto de datos de entrenamiento, y otro distinto e independiente de testing. La independencia evita potenciales overfittings.

La precisión del modelo dependerá de los resultados en el conjunto de test, con el cual se puede construir una matriz de confusión, dividiendo los resultados en correctos (la diagonal de la matriz), falsos positivos, y falsos negativos para problemas de decisión binaria.

Funcionalidades del Data Mining

Entre sus funcionalidades tenemos:

- **Descripción de conceptos:** Caracterizar y discriminar a los datos a través de sus características (generalizar, resumir, contrastar).
- **Asociación:** Establecer una relación de correlación y causalidad.
- **Clasificación y predicción:** Encontrar modelos o funciones que describan y distingan clases para futuras predicciones. Estos se pueden presentar de varias maneras (árboles de clasificación, reglas, redes neuronales) y nos permiten predecir valores faltantes.
- **Cluster analisis:** Agrupar los datos en clases maximizando su similitud y minimizándola entre clases.
- **Análisis de outliers:** Detectar y comprender aquellos datos que no respetan el comportamiento general.
- **Análisis de tendencias y evolución:** Comprender la regresión, patrones secuenciales y similitudes a través del tiempo.

Para ello se tiene una serie de técnicas que pueden ser:

- **Supervisadas:** Se basan en un conjunto de entrenamiento con sus respuestas anotadas (redes neuronales, árboles de decisión, regresión).
- **No supervisadas:** Deben inferir una función para describir una estructura oculta a partir de datos no etiquetados (clustering, reglas de asociación).

Redes neuronales artificiales (RNA)

Definición: Son sistemas capaces de **aprender de sus propios errores** y adaptarse a condiciones variantes y ruido para predecir un estado futuro al asociar entradas a respuestas. Se usan para resolver problemas a **gran escala** (asociación, evaluación y reconocimiento de patrones) o **difíciles de calcular** (aproximadamente con respuestas rápidas y buenas).
Características:

- Aunque no se propagan siguiendo una secuencia predefinida de instrucciones, sólo resuelven **problemas resolvibles** por el cerebro humano.
- Se procesan paralelamente a través de un gran número de elementos **altamente interconectados** entre sí.
- Pueden mejorar su rendimiento al combinarse con otras herramientas (lógica difusa, algoritmos genéticos, sistemas expertos, estadísticas, transformada de Fourier, wavelets).
- No son útiles para cálculos precisos, procesamiento serie ni reconocer algo que no siga algún tipo de **patrón**.
- Se basan en modelos simplificados de neuronas reales (modelan el axón, las dendritas, la sinápsis y el cuerpo de la célula).

Entrenamiento:
El entrenamiento de una RNA sigue una **regla delta generalizada** consistente en un proceso con todos los datos de entrenamiento que puede repetirse varias veces:

1. Calcular la diferencia entre la salida resultante y la esperada.
2. Corregir los valores de las entradas para achicar las diferencias en base a una constante **delta** muy pequeña.

De esa forma se busca que la diferencia se vaya minimizando **de a poco**, ya que de hacerlo de golpe se puede modificar demasiado lo aprendido anteriormente.
Tipos:
Entre los tipos de RNAs más utilizados tenemos:

- Redes Neuronales Profundas es un termino paraguas para muchos tipos de Redes Neuronales pero que poseen muchas capas, dándole gran profundidad a muchas de ellas. Su principal problema es el gran costo computacional que implica entrenarlas debido a la gran cantidad de nodos que requieren entrenarse/optimizarse, y todas las conexiones que poseen.
- Las Redes Neuronales de Convoluciones son más sencillas, basadas en la operación matemática de la convolución. Buscan percibir detalles de más alto nivel, y generalmente se usan para detección de patrones en imágenes como por ejemplo bordes.

Redes neuronales que no aparecen en las diapositivas de 2022:

- Perceptrón multicapa
- Red de Hopfield (mapas asociativos)
- **Red de Konohen (SOM, mapas auto-organizativos):** Se basan en evidencias de cómo las neuronas del cerebro organizan su información, y en ellas la actualización delta sólo se realiza en la neurona cuyos pesos tengan la distancia mínimo con el valor a entrenar, afectando en menor medida a sus vecinas.

Árboles de decisión

Definición: Son modelos en forma de árbol que se utilizan para **clasificar** una entrada desconocida según sus **atributos**. Se componen de nodos internos con preguntas condicionales y entendibles sobre ellos, y hojas con su etiqueta o clase a predecir.
Construcción:
Para construirlo se parte de todos los ejemplos de la raíz del árbol y se los va dividiendo recursivamente a través de los atributos elegidos. Seguidamente, se podan las ramas con outliers o ruido (**prunning**).
Considerando que el modelo se construye en base a clases existentes de entrenamiento, de este se obtienen las **reglas de clasificación**. De allí, para estimar su **precisión** se debe aplicarlo sobre un conjunto de prueba y comparar sus resultados con los reales, tomando el porcentaje correctamente clasificado.
Prunning:
En el prunning entra en juego el **overfitting** basado en adaptar el árbol demasiado al conjunto de entrenamiento y puede mediarse a través de:

- **Preprunning:** Interrumpir la construcción de un nuevo nodo si la mejora está por debajo de cierto umbral (difícil de definir).
- **Postprunning:** Quitar ramas de un árbol ya construido (usando otro conjunto de entrenamiento, por ejemplo).

Ventajas:

- Son fáciles de entender por humanos porque manejan datos categóricos y numéricos.
- Requieren menos preparación que otros modelos, y usan menos suposiciones.
- Usan reglas de asociación vinculadas a modelos de razonamiento más complejos.

Regresión lineal

Definición: Es una técnica estadística que nos permite modelar e investigar la **relación entre dos o más variables** de un esquema. De allí, si se manejan sólo dos variables independientes es **simple** y si no, **múltiple**.
Requisitos para crear el modelo:

- La relación entre las variables debe ser lineal.
- Los errores deben ser independientes entre sí.
- La varianza de los errores deben ser constante y su esperanza matemática, nula.
- El error total debe ser la suma de cada uno.

Clasificación bayesiana:
La **regresión logística** se aplica cuando tenemos una variable dependiente dicotómica o politómica y no numérica. De allí, asociamos la variable con su **probabilidad de ocurrencia** e intentamos probar una hipótesis a través de la clasificación bayesiana.
Esta nos permite aproximar las probabilidades de la hipótesis, verificar cómo sube o baja con cada ejemplo de entrenamiento y realizar múltiples predicciones. Su aplicación parte del **teorema de Bayes** de probabilidad condicional, sólo que en su versión "naive" que asume que los atributos son independientes para reducir el costo de cálculo.
Entonces, las probabilidades "a-posteriori" se calculan en base a los atributos según si son categóricos o no continuos. Podemos superar la hipótesis de independencia usando redes bayesianas o árboles de decisión.

Clustering

Definición: Técnica basada en agrupar objetos dentro de colecciones llamadas **clusters**, de manera de que en cada uno, sus objetos sean similares entre sí y diferentes de los que están por fuera de ellos. Se usan para tener una idea de la **distribución de los datos** como paso previo a la aplicación de otros algoritmos.
Calidad: La calidad de un cluster viene dada por la **función de similitud** utilizada por el método (que depende del tipo de datos) y la manera en que está implementada. Entre las posibles funciones de distancia tenemos la euclídea, Manhattan, Minkowski (generalización de la anterior), etc., cada cual aplicable según el problema a resolver.
El clustering puede tener **agrupamiento**:

- **Jerárquico:** Puede hacerse a través de métodos aglomerativos o divisivos. No tiene número de clusters definido, no actúa bien cuando los datos tienen alto nivel de error y puede ser lento.
- **No jerárquico:** Rápido y fiable pero requiere especificar el número de clusters y la semilla inicial (arbitrarios).

Reglas de asociación

Definición: Se basa en hallar automáticamente patrones comunes, asociaciones, correlaciones o estructuras de causalidad entre los ítems u objetos en bases de datos transaccionales, relacionales y otros repositorios de información. Para eso forma **reglas** del estilo `IF condición THEN resultado`.
Reglas:
Las reglas pueden ser:

- **Úitles o aplicables:** Si contienen una buena cantidad de información y son traducibles a acciones de negocio.
- **Triviales:** Si ya se conocen por su frecuente ocurrencia.
- **Inexplicables:** Si se corresponden con curiosidades arbitrarias.

Su calidad se puede medir a través de su:

- **Soporte:** Proporción de transacciones en la que se encuentra.
- **Confianza:** Proporción de transacciones que la contienen respecto de la proporción que contienen a la cláusula condicional.
- **Mejora:** Capcidad predictiva de la regla.

Pueden además ser booleanas o cuantitativas, tener una o varias dimensiones y manejar elementos simples o jerárquicos.

Seguridad en Bases de Datos

Las BDs sostienen muchos datos críticos del funcionamiento del negocio. Ventas, datos personales de clientes empleados y otros, e incluso propiedad intelectual.

Objetos que se deben resguardar son las tablas, vistas, stored procedures (funciones precompicaladas de lógica de negocio en formato SQL) y triggers (queries de SQL que se causan por eventos predeterminados, útiles para auditorías).

Tipos de Amenazas a la Seguridad

- **Perdida de Integridad:** Los datos se editan inapropiadamente de forma no autorizada con fines fraudulentos.
- **Perdida de Disponibilidad:** prohíbe el acceso legítimo a la BD.
- **Perdida de Confidencialidad:** divulgación intencional o no, no autorizada de datos protegidos o confidenciales.

Ataques comunes

- Abuso de privilegios legítimos para uso no autorizado.
- Autenticaciones débiles por robo de credenciales o políticas débiles de elección de contraseñas. (se debe asegurar políticas que impidan el uso de contraseñas default, caducar contraseñas cada cierto tiempo, evitar el uso de cuentas genéricas o compartidas, controlar el uso de guests o anónimos, y analizar logins fallidos, longitud de passwords y su reuso. Las versiones nuevas de SQL permiten este tipo de controles automáticos).
- Ataques a través de configuraciones débiles de los sistemas con herramientas vulnerables o configuraciones no seguras. por defecto.
- SQL Injections.
- Cross site Scriptings.

- Root Kits
- Ataques a protocolos de comunicación débiles.
- Abusos de vulnerabilidades en el Front-End. Son muy comunes vulnerabilidades donde las aplicaciones (por ejemplo las del Front-End) realizan todas las operaciones con un usuario propio, muchas veces hard-coded de forma que si la aplicación tiene una vulnerabilidad, un usuario puede tener acceso directo a la Base de Datos, pudiendo alterar sus tablas accederlas sin autorización, o incluso eliminarlas (por ejemplo con SQL Injections). Un equipo de gestión de riesgo debe evaluar si su protección es adecuada.
- Back Ups (último recurso) incompletos, fallados, inadecuados, poco usables o incluso ataques de robo de Back Ups. Es necesario tener Back Ups frecuentes y adecuados, además de buenas políticas de recovery (una no excluye a la otra).

Regla de Anderson (Anderson's Rule)

Una base de datos es menos segura mientras más accesible lo sea. El ideal que esto busca remarcar es conseguir una base de datos con seguridad perfecta (sin vulnerabilidades (suponiendo que esto fuese posible)), y una vez obtenida, maximizar la accesibilidad, manteniendo la seguridad.

Mecanismos de protección

Es necesario evaluar riesgos en todos los componentes conectados a una Base de Datos, entre ellos:

- Sistemas Operativos usados.
- Privilegios otorgados en el sistema de archivos del Sistema Operativo, especialmente a los archivos de la Base de Datos, pero también al resto de archivos críticos del sistema.
- Componentes de Red.
- Sistemas de Aplicación.
- Seguridad Física misma, y todo elemento del sistema. (Hardware, cables, redes inalámbricas, etc)
- Identificación y autenticación de usuarios y roles.
- Sistemas de privilegios y control de acceso a objetos.
- Mecanismos de trazas para auditorias, con documentaciones exhaustivas de dichas auditorías.
- Encripción de datos.
- Seguridad de las redes.

Reglas de privilegios

Los privilegios se otorgan con reglas que pueden ser:

Privilegios de Sistema: permitiendo acceso a comandos u operaciones específicas de la BD.

Privilegios de objetos: permitiéndole acceder al usuario de una forma especifica al objeto.

Controles de Acceso

Hay varios tipos de formas de definir políticas de acceso llamadas, Controles de Acceso:

Discretionary Access Control: privilegios otorgados según el usuario y el objeto accedido. El "dueño" del objeto se responsabiliza de definirlo.

Role-based Access Control: privilegios otorgados según roles, que un usuario puede tener o no, y un objeto puede exigir.

Rule-based Access Control: privilegios en función de un conjunto de reglas.

Mandatory Access Control: privilegios concedidos según rangos jerárquicos de roles. Si un campo de una tabla exige un rango no poseído, el campo sensible puede retornarse nulo (indicando que es información clasificada), y para verlo requerirá poseer un rango de rol adecuado.

Monitoreo de comportamiento sospechoso

Se puede hacer logging y monitoreo de todo el comportamiento de los usuarios (recordemos que esto va a tener un costo en espacio), o exclusivamente del comportamiento sospechoso, el cual se debe definir. Como por ejemplo:

- Logins exitosos/fallidos.
- Accesos exitosos/fallidos a la Base de Datos.
- Horarios no habituales de uso de la Base de Datos.
- Múltiples intentos de acceso de distintos usuarios, desde una misma terminal.

Auditorías y DBA

El DBA (Data Base Administrator) es responsable de hacer auditorías de la seguridad de toda la Base de Datos, sus protocolos, sus usuarios, sus mecanismos de seguridad, sus accesos, etc.

Historia de las Bases de Datos (posiblemente no evaluado 2022)

Este tema no parece ser evaluado en los finales (no lo vi preguntado en los de entre 2018-2022), pero quizá sirve.

- Origen 1960, Network Data Model (Integrated Data Store y CODASYL)
- En la misma época, Modelos Jerárquicos (IBM Information Management System)
- Ted Codd 1970s, Modelo Relacional
- 1980s Boom Relacional (System R, Ingres, Oracle) y Relational-Object Mismatch Problem.
- 1990s PostgreSQL, MySQL, Microsoft SQL Server (Para usuarios más pequeños de Bases de Datos).
- 2000s NoSQL boom por el crecimiento de la web y su demanda, middleware (Facebook Google), Datawarehousing y OLAP (a diferencia de OLTP, ambos vistos en su sección respectiva).
- 2010s NewSQL Boom de 2 tipos: Por un lado DBMS OLTP ACID distribuidos, por otro lado Hibridos OLTP y OLAP distribuidos mezcla open y closed source.
- 2010s Tambien aparece las DBaas (DataBase as a Service), Bases de Grafos y de datos temporales.

Data Warehousing (Opcional, tema no evaluado en 2022)

Business intelligence

Comprende el conjunto de **conceptos y metodologías** que buscan mejorar el proceso de **toma de decisiones en los negocios** al basarse en hechos y sistemas que trabajan con ellos. Sus datos provienen de fuentes como **warehouses** y comprenden una serie de **herramientas** para administrarlos, extraerlos, consultarlos y modelizarlos.
La **evolución** de los datos de negocios a la información de negocios pasó por varias etapas:

- **Data collection (1960):** Acceso retrospectivo y estático
- **Data access (1980):** Acceso retrospectivo y dinámico
- **Data navigation (1990):** Acceso retrospectivo y dinámico con niveles múltiples
- **Data mining (2000):** Acceso prospectivo y proactivo

Data Warehouse (DW)

Surge como una colección de muchos datos que asisten a la **toma de decisiones de negocio**. Sus datos cumplen con ser:

- **Subject-oriented:** Tratan sobre de un tema en particular en lugar de la operatoria de la compañía.
- **Integrated:** Provienen de diferentes fuentes y son integrados consistentemente.
- **Time-variant:** Se refieren a un momento en particular en el tiempo (**snapshot**).
- **Non-volatile:** Son estables, es decir, suelen agregarse pero no quitarse para permitir una **análisis retrospectivo** de la marcha del negocio.

Puede decirse entonces que los datos están estructurados para responder a **transacciones y consultas complejas** (lectura y sumarización heurística).
Dentro de los componentes del DW, además de los datos, están:

- Los procedimientos de extracción, transformación y carga (**ETL**)
- Soporte físico de los datos (**DBMS**)
- Herramientas de explotación (**OLAP**, reporting, **data mining**, etc.)

Metadata

Describe los datos a los usuarios de manera de que puedan interpretarlos. Esta descripción abarca, entre otras cosas:

- Sus modelos lógicos
- Su mapeo a sistemas transaccionales
- Su esquema físico
- Su información de carga

La metadata se encarga de describir no sólo a la fuente de datos sino a las operaciones de transformación, estructuras de datos, reglas de limpieza y referencias históricas y temporales. De allí podemos hacer la siguiente distinción:

- **Datos:** Son los que se cargan a una base en crudo.
- **Información:** Surge de los anteriores al incorporarles una definición, un formato, un intervalo de tiempo y relevancia (es decir, datos de contexto).
- **Conocimiento:** Surge de la anterior a partir del análisis de patrones y tendencias, las relaciones y las asumciones tomadas. Para obtenerlo tenemos diversas técnicas como data mining.

Adquisición y limpieza

Como se tienen muchos datos en el DW pueden haber algunos corruptos, redundantes, irrelevantes o excesivos. Necesitamos que estos sean **correctos y completos** para satisfacer las necesidades del usuarios. Para ello podemos imponer una serie de **condiciones de integridad** para que los datos se ajusten a los estandartes de valor y completitud. El proceso encargado de tratar estos problemas es el **ETL**.

ETL

Es el proceso que se usa para obtener los datos, limpiarlos y convertirlos a un formato con el que se puedan utilizar. Este comprende varias etapas:

1. **Migración:** Se toman los datos de sistemas operacionales por fuera de las áreas de trabajo del DW a ellas, evitando traer datos innecesarios (**control preventivo de integridad**).
2. **Limpieza (data cleaning):** Se corrigen, estandarizan y completan los datos, identificando **redundantes**, atípicos (**outliers**) y perdidos (**missings**). Esto comprende la **normalización** de datos a una denominación uniforme para poder ser referenciados por el sistema de negocios. Además se deben **simplificar** los esquemas de codificación, particionando datos complejos.
3. **Transformación:** Comprende una serie de **procesos** para adpatar los datos al **modelo lógico** del DW. De allí se genera una serie de reglas de transformación que deben validarse con los usuarios. Entre ellos tenemos:
   - **Snapshots:** Para tratar entidades con cambios frecuentes.
   - **Denormalización:** Para mejorar el rendimiento de las consultas y reflejar relaciones estáticas (que no cambian en un perspectiva histórica).
   - **Sumarización:** Para acelerar los tiempos de análisis y ocultar la complejidad de los datos. Pueden incluir múltiples juntas y vistas que deben ser mantenidas a medida que se cargan nuevos datos, para lo cual resulta necesario navegar los datos hasta su mínimo **nivel de granularidad**.
4. **Carga de los datos al DW físico:** Se puede hacer de dos maneras:
   - **Full refresh:** Cargando todos los datos nuevamente.
   - **Incremental:** Cargando sólo los últimos datos.
5. **Conciliación y validación:** Comprende los controles de **detección de integridad** que verifican que los datos sean correctos y completos. Se puede hacer de manera:
   - **Completa:** Al final de todo el proceso.
   - **Por etapas:** A medida que los datos son cargados.

Modelado

El modelo de la DB que conforma un DW debe poder soportar eficientemente los requerimientos de los usuarios, ya que les da una **visualización del universo de negocio**, abstrae sus preguntas y determina cómo será implementado. De allí tenemos dos técnicas:

- **Modelo Entidad-Relación:** Entidades, relaciones y atributos.
- **Modelo Dimensional:** Se basa en 3 elementos:
  - **Hecho:** Colección de ítems de datos que con su contexto representan a un ítem de negocio, una transacción o un evento. Estos se registran en las **tablas centrales**.
  - **Dimensión:** Colección de miembros, unidades o individuos del mismo tipo (relacionados). Estas determinan el contexto de los hechos y se usan como parámetros para el análisis **OLAP**. Suelen ser: tiempo, geografía, cliente y vendedor, y pueden agrupar sus ítems jerárquicamente.
  - **Medida:** Atributo numérico de un hecho que representa su rendimiento o comportamiento relativo a su dimensión (ventas en pesos, cantidad de productos, total de transacciones).

  Puede verse este modelo como un caso particular del de E-R y se puede modelar de dos maneras:
  - **Star:** Una tabla central de hechos y una serie de tablas individuales para cada dimensión.
  - **Snowflake:** Una tabla central de hechos pero con sus dimensiones organizadas jerárquicamente.

OLAP

(no soy el autor original de esto, pero sinceramente no entendí de donde salió esta definición de OLAP, recomendaría ver la nueva definición que puse más arriba (después de Column Store Databases), pero como no lo entendí decidí no borrarlo por si le sirve a alguién)

Es una **herramienta de explotación de datos** aplicable al modelo dimensional, específicamente al cubo de medidas en base a sus dimensiones. Sus operaciones se agrupan según:

- **Nivel de Granularidad:** Se basan en el tamaño de la unidad mínima de tratamiento cuya medida resulta importante en el diseño. Aquí tenemos dos:
  - **Roll Up:** Generalización de datos de una vista al reducir dimensiones o escalar en su jerarquía.
  - **Drill Down:** Especialización de datos al introducir una nueva dimensión en una vista o bajar en la jerarquía.
- **Navegación por las dimensiones:** Toman un subconjunto de los datos según la cantidad de dimensiones. Aquí también tenemos dos:
  - **Slice:** Toma una "rebanada" del cubo dimensional, es decir, los ítems con un mismo valor en alguna de las dimensiones.
  - **Dice:** Similar a la anterior pero comprende 2 o más dimensiones.