Resumen de Base de Datos

Guido Tagliavini Ponce

Última modificación: 11 de enero de 2017

Cursada del 1C-2016

No-SQL

- Término para denominar aquellas bases de datos que no siguen los principios de los RDBS (Relational Database Systems).
- Problemas de RDBS:
    - Para cumplir con las propiedades ACID, necesitan de diversos subsistemas funcionando:
        - **Subsistema de recovery.** Garantiza que si una transacción es abortada en cualquier momento, deben retrotraerse los cambios realizados (atomicidad). Además, si en algún momento hay algún problema con la base de datos, debe poder recuperar las modificaciones hechas por transacciones ya instaladas (durabilidad).
        - **Subsistema de control de concurrencia.** Controla el nivel de aislamiento, administrando locks (en el caso de control pesimista) o la lógica de versionado (en el caso de control optimista).
        - **Control de restricciones de integridad.** Para asegurar consistencia.
    - Todos estos subsistemas actúan en detrimento de la performance y la velocidad de respuesta.
    - Las queries relacionales suelen involucrar joins entre tablas. En el caso de una base de datos relacional distribuida, esto implica el envío de grandes volúmenes de datos entre nodos, aumentando dramáticamente el uso de ancho de banda y deteriorando notablemente la performance.
- Las bases de datos No-SQL aseguran un set distinto de propiedades, llamado **BASE**:
    - **Basic Availability.** Toda solicitud genera una respuesta.
    - **Soft-state.** El estado del sistema puede cambiar con el tiempo, a veces sin ninguna entrada. Esto es debido a la consistencia eventual.
    - **Consistencia eventual.** La BD puede ser momentáneamente inconsistente, pero eventualmente será consistente.
- Ventajas de No-SQL:

    - Representación libre de esquemas. Pueden convivir datos con distintas estructuras.
    - Mejor desempeño en entornos distribuidos.
    - Mejor escalabilidad.
- Tipos de BD No-SQL:
    - Key-Value
    - Column family
    - Orientadas a documentos
    - Orientadas a grafos
- **Key-Value**
    - Idea: vincular claves con valores. Similar a un diccionario.
    - Operaciones básicas: put(key, value), get(key), delete(key).
    - Es posible ubicar una clave eficientemente. La operación principal que nos interesará realizar es encontrar una clave específica, y no escanear todas las claves. Esto lo hace escalable.
    - Ejemplo: Redis.
    - Algunos sistemas permiten la caducidad de claves después de un tiempo (Time To Live o TTL). Pasado ese tiempo, la entrada es desalojada de la DB.
    - No hay tablas, sino *namespaces* o *buckets*.
    - Modelado:
        - Como no hay columnas, de alguna forma hay que indicar a qué atributo nos estamos refiriendo en cada entrada. Por ejemplo, podríamos querer asociar, para cada DNI, el nombre de una persona, y su dirección.
        - Claves: **nombre_entidad + ‘:’ + identificador_entidad + ‘:’ + nombre_atributo**
        - Valor: cualquier valor de tipo string, lista, conjunto, JSON, etc.
        - Ejemplo 1. Base de datos de información de personas, con DNI, nombre y dirección. Las claves van a ser de la pinta persona:id_persona:atributo.

          namespace[persona:1:dni] = “12345678”
          namespace[persona:1:direccion] = “Av. Cabildo 1500”

          Para ubicar a una persona específica, debemos saber su id_persona. Entonces, debemos asociar cada DNI (atributo que sí conocemos) con el id_persona.

namespace[id_persona:12345678:id] = “1”

Otra opción sería dividir la información en dos namespaces distintos.

- Ejemplo 2. Siempre conviene hacer agregados atómicos. Esto es, en lugar de agregar datos en forma de objetos JSON grandes, conviene agregar cada campo por separado. Consideremos una base de datos de materias dictadas, con su nombre y profesor que la dicta.

  namespace[materia:1] = {“nombre” : “Algoritmos”, “profesor” : “Pepe”}
  namespace[materia:2] = {“nombre” : “Álgebra”, “profesor” : “Pepa”}

  En lugar de este esquema, es más conveniente este otro:

  namespace[materia:1:nombre] = “Algoritmos”
  namespace[materia:1:profesor] = “Pepe”
  namespace[materia:2:nombre] = “Álgebra”
  namespace[materia:2:profesor] = “Pepa”

- Ejemplo 3. Base de datos de personas, con pasaporte y ciudad natal. Queremos poder buscar rápido por nombre.

  namespace[persona:1:pasaporte] = ”ABC123”
  namespace[persona:1:ciudad] = “Buenos Aires”
  namespace[persona:2:pasaporte] = ”DEF456”
  namespace[persona:2:ciudad] = “Seattle”
  namespace[persona:3:pasaporte] = ”GHI789”
  namespace[persona:3:ciudad] = “Buenos Aires”

  Este esquema de por sí no nos permite buscar rápido por ciudad, *excepto que la base de datos soporte índices*. Si no soportara índices, podríamos construir nuestro propio índice invertido: para cada ciudad, la lista de ids de personas que hayan nacido en dicha ciudad.

  namespace[ciudad:Buenos Aires:ids_persona] = (1, 3)

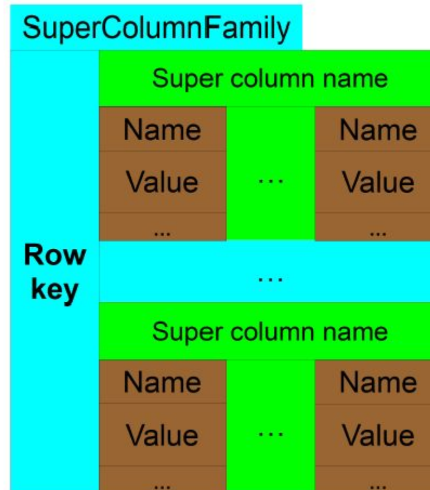
namespace[ciudad:Seattle:ids_persona] = (2)

- **Column family store**
  - Idea: En una BD relacional, cada tabla se almacena por filas. Esta organización física de los datos determina que aquellas operaciones que iteren sobre columnas, como por ejemplo las agregaciones, serán lentas. En cambio, si almacenamos los valores de las columnas en forma contigua, tales operaciones son mucho más eficientes.
  - Una BD column family divide a las columnas (de la tabla subyacente) en *familias de columnas*. Una familia de columnas es un conjunto de columnas almacenadas con cierto grado de localidad (mismo archivo, o mismo sector del disco, o misma computadora, etc.). Las familias de columnas no necesariamente son disjuntas.
  - Una BD column family se divide, desde un punto de vista lógico, en *rows*. Cada row asocia una key (la clave de la fila en la tabla subyacente), con un conjunto de familias de columnas. Las rows están ordenadas por key. Desde el punto de vista físico, esto se traduce en que cada familia de columnas está ordenada por las row keys.
  - Si queremos ubicar el valor de una columna C para una clave K, debemos ir a la familia F de la columna C, y extraer el valor de C en la row K. Como dentro de F los elementos de las columnas están ordenados por row keys, ubicar el valor asociado a K en C es rápido.
  - Si queremos realizar una agregación sobre toda una columna C, vamos a la familia F de la columna C (datos con localidad espacial), y recorremos los valores de C secuencialmente. Si la agregación depende de la row key, recorremos sólo el rango deseado.
  - Una columna C en una familia F no tiene por qué definido su valor para toda clave K. Esta representación permite que los valores vacíos de la tabla subyacente no ocupen espacio.
  - Dos tipos de column family stores:
    - Standard column family. Cada row almacena todas las columnas (la column family está formada por todas las columnas de la tabla relacional).

ColumnFamily

Row key	Name	...	Name
	Value		Value
	...		...

        - Super column family. Es el esquema descripto antes. Cada row almacena un conjunto de column families, en lugar de una sola.



    - Cada valor de una columna está acompañado de un timestamp. Esto permite almacenar todos los valores que una columna ha tenido a lo largo del tiempo.
    - Ejemplos: BigTable, Cassandra.
    - Modelado:
        - Dado que se orienta en torno a agrupar columnas en familias para mejorar la performance, el modelado está dirigido por los patrones de escritura y lectura deseados.

- **Orientadas a documentos**
    - Los datos se agrupan en colecciones de documentos. Un documento es un registro con datos semi-estructurados (por ejemplo XML o JSON).
    - Los documentos no necesariamente tienen todos la misma estructura.
    - Se puede consultar por cualquier valor de los documentos.
    - Ejemplo:

```
{
        “id_empleado” : 1,
        “nombre” : “Pepe”,
        “dni” : “12345678”,
        “edad” : 91
}
```

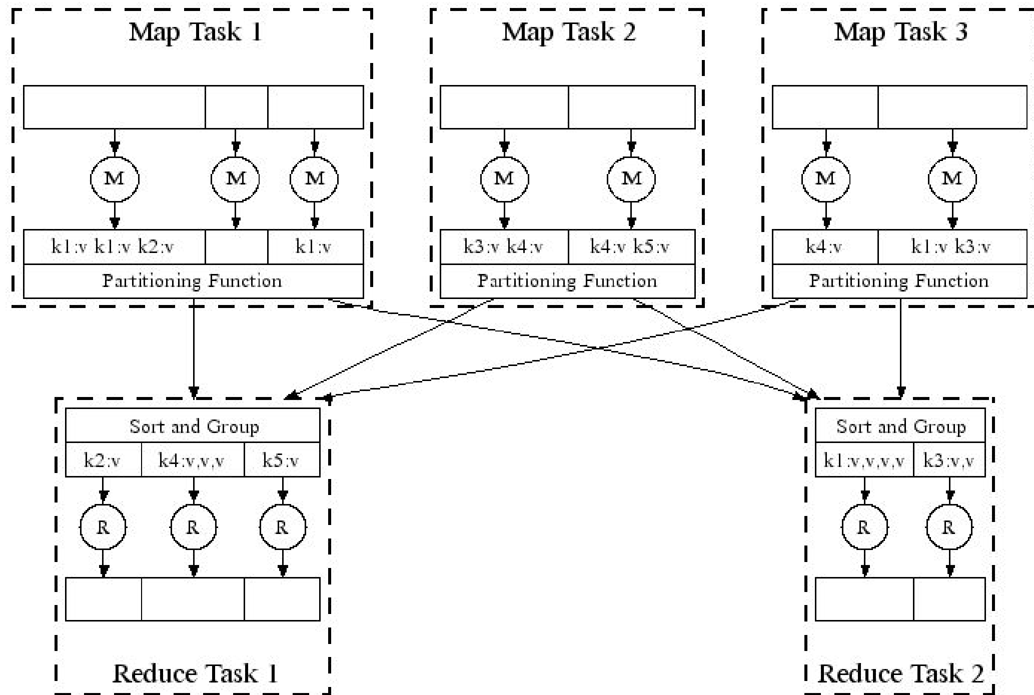
    - Se suelen utilizar índices sobre los valores.
    - Ejemplo: MongoDB.

- **Orientadas a grafos**

    - Los datos se organizan en forma de grafos. Cada nodo representa una entidad, y puede estar relacionado con otros nodos.
    - Ejemplo:
      (“Pepe”, tipo, “Persona”)
      (“Pepe”, tiene_dni, “12345678”)
      (“Pepe”, leyo_libro, “Catching Fire”)
      (“Pepe”, leyo_libro, “Sycamore Row”)
      (“Sycamore Row”, escrito_por, “John Grisham”)
      (“John Grisham”, tipo, “Persona”)

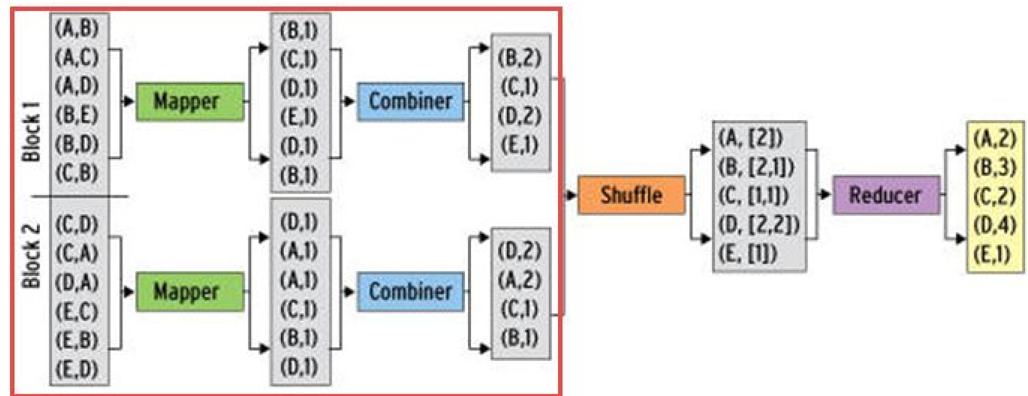
      En esta base de datos podríamos hacer la consulta “¿Qué libros de John Grisham leyó Pepe?”.
    - Resulta útil cuando el núcleo de nuestro problema son las relaciones entre los objetos del dominio.
    - Permite responder consultas sobre la forma en que están relacionados los objetos. Por ejemplo “¿Cuántos amigos tiene una persona, en promedio?” o “¿Cuántas películas tienen como protagonistas a un actor y un hijo suyo?”.
    - Son difíciles de hacer escalar, puesto que no es fácil particionar un grafo en varios servidores de modo tal de no afectar la performance general de las queries.
    - Ejemplo: Neo4j.

- **MapReduce**
    - Idea:
        - Objetivo: procesar un gran volúmen de datos. Lo hacemos en dos pasos:
            - Map: para cada elemento extraer algo que nos interesa, produciendo una tupla (k, v).
            - Agrupar los v de un mismo k. Es decir, producir una tupla (k, [v1, …, vn]).
            - Reduce: Aplicar una agregación sobre [v1, …, vn] y producir un resultado.
        - Como la cantidad de datos iniciales es enorme, debemos realizar este cómputo en forma distribuida. La arquitectura utilizada para ejecutar este procedimiento es la siguiente:



Cada uno de las tareas se ejecuta, potencialmente, en un nodo distinto. Aquellas máquinas que ejecutan el map se denominan mappers, y las que ejecutan el reduce se denominan reducers. El entorno MapReduce se encarga de todas las tareas involucradas en este procesamiento distribuido: particionar los datos de entrada, elegir los mappers y reducers, manejar las fallas de las máquinas y administrar la comunicación.

- Combinable reducer:
  - Es un reduce que se ejecuta en el mismo nodo de un map. La idea es directamente reducir los valores asociados a una clave k producido en un mapper, y transmitirle este valor al reducer encargado de k.
  - El reduce debe producir el mismo tipo de valores que los que toma. Además debe ser asociativo y conmutativo.



- Ejemplo:
  Calcular la cantidad de personas con edad entre 35 y 40 años. Los documentos que se procesan tienen el siguiente esquema:

```
{
        "id" : integer,
        "nombre" : string,
        "edad" : integer
}

map(doc) {
        if (doc["edad"] >= 35 && doc["edad"] <= 40)
                emit(doc["id"], 1);
        else
                emit(doc["id"], 0);
}

reduce(key, values) {
        return (key, sum(values));
}
```

- **Sharding**
  - Particionado de los datos. Surge a partir de la imposibilidad de almacenar todos los datos en la misma máquina.
  - Un shard es un fragmento de todos los datos. Los shards comparten esquema y su unión representa la totalidad del dataset.
  - Debemos determinar a qué shard mandamos cada dato. Para esto se utiliza un hash de algún atributo de los datos. Por ejemplo la primera letra del nombre de usuario, o la ubicación geográfica.
  - Ventajas:

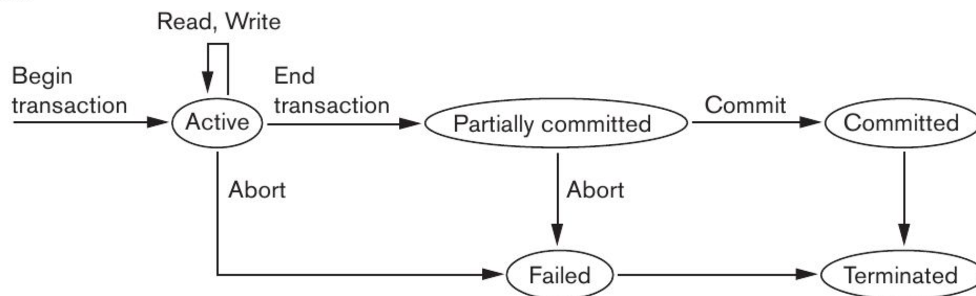
        - Permite escalar horizontalmente.
        - Provee tolerancia ante fallos, pues si un nodo falla, sólo su porción de los datos quedan fuera de servicio.
    - Desventajas:
        - Las consultas ahora requieren la comunicación entre varios shards. Gran uso del ancho de banda.
- **Replicación**
    - Queremos garantizar que el servicio siempre estará disponible. Por ende, si un nodo se cae, queremos seguir pudiendo responder a todas las consultas. Debemos replicar el servidor (o los servidores) que proveen el servicio, sobre varias máquinas.
    - Las lecturas pueden realizarse desde cualquier réplica. Las escrituras son más delicadas.
    - Modelos de replicación:
        - Master-Slave: las escrituras (insert, delete, update) se hacen desde un único nodo. Éste se encarga de transmitir el nuevo valor a las otras réplicas. Si el nodo master falla, un slave toma su lugar.
            - (+) Es simple. No hay que lidiar con problemas de orden temporal de escrituras.
            - (+) Ideal para escenarios de lectura intensiva.
            - (-) El master es un cuello de botella. Ante demasiadas escrituras podría saturarse.
            - (-) Las lecturas pueden ser inconsistentes, debido a la diferencia de tiempo para recibir las actualizaciones en cada réplica.
        - Peer-to-Peer: todas las réplicas manejan escrituras y lecturas por igual, que transmiten a los otros slaves.
            - (+) Al ser descentralizado, no hay cuellos de botella. Mayor tolerancia a fallas.
            - (+) Mayor throughput, porque todos los nodos pueden responder peticiones.
            - (-) Lecturas inconsistentes y escrituras conflictivas.
    - Solución al problema de inconsistencia por lecturas concurrentes:
        - Sistema de voting. Cada vez que llega una petición de lectura de un registro, se hace voting entre las réplicas para determinar si hay un valor mayoritario de ese registro. Se devuelve dicho valor.
        - Estrategia de concurrencia pesimista. Lock de lectura (sólo permite otras lecturas) sobre un registro a leer.
    - Solución al problema de conflictos por escrituras concurrentes:

        - Estrategia de concurrencia pesimista. Lock de escritura (no permite realizar ninguna otra operación) sobre un registro a escribir.
        - Estrategia de concurrencia optimista. No usa locks. Permite inconsistencias, a sabiendas de que eventualmente se llegará a un estado consistente.
- Sharding + Replicación
    - Por cada shard creamos varias réplicas.
    - Esto permite combinar la escalabilidad que provee el sharding, con la disponibilidad de la replicación.
    - Commodity processors: usar los mismos nodos que almacenan datos, para realizar procesamiento y responder a las queries localmente. Esto evita enviar grandes volúmenes de datos a través del cable.
- Teorema CAP vs. No-SQL
    - Típicamente, las bases de datos se utilizan en entornos distribuidos. Pensemos cómo impacta el teorema CAP en términos de un tal sistema distribuido.
    - Consistencia: en cada momento, todo nodo debe dar el mismo resultado a una query dada.
    - Disponibilidad: en cada momento, todo nodo debe dar un resultado a una query dada, aunque no sea el más actual.
    - Tolerancia a partición: si hay una falla en la red de la base de datos, el sistema no debe morir.
- Síntesis de los 4 modelos de bases No-SQL
    - Key-Value
        - Las únicas queries que permite son get(key).
        - Los valores asociados a las claves son opacos. Esto significa que la BD no conoce su estructura.
        - Su simplicidad hace que la resolución de las queries sea sumamente rápida.
    - Document oriented
        - Los documentos no son opacos.
        - Permite queries complejas, utilizando la estructura de los documentos y realizando agregaciones.
        - Muchas BD conoce su estructura.
    - Column family
        - Ideales para hacer análisis de datos (agregaciones).
    - Graph databases
        - Ideales para problemas donde el objeto central son las relaciones entre las entidades.

- Desnormalización
  - Proceso de agregar redundancia de datos, con el fin de optimizar ciertas consultas sobre la base de datos.

Control de concurrencia

- **Transacción**
  - Sucesión de operaciones que se ejecutan formando una unidad lógica de procesamiento.
  - En un modelo sin locking, una transacción se compone de cuatro tipo de acciones:
    - Leer un ítem X: r[X]
    - Escribir un ítem X: w[X]
    - Abortar: a
    - Commitear: c
  - El elemento X, que llamamos ítem, puede representar diversas cosas. Por ejemplo, un atributo, o una tupla entera, o una instancia de una relación.
- **Diagrama de estados de una transacción**



- **Propiedades ACID**
  - Cuatro propiedades que satisfacen las transacciones.
  - **Atomicity**. Una transacción se ejecuta completamente, o no se ejecuta por completo.
  - **Consistency**. Al principio y al final de la ejecución de una transacción, la BD está en un estado consistente. Esta consistencia está dada por cierto conjunto de invariantes.
  - **Isolation**. Una transacción no interfiere con otra ejecutada concurrentemente. Una transacción debe aparentar ser ejecutada como si

lo hiciera en forma aislada, incluso si hay otras ejecutándose concurrentemente.
    - **Durability**. Los cambios aplicados a la BD por una transacción commiteada deben persistir en la BD. Estos cambios no se deben perder debido a una falla del sistema.
- **Historia**
    - Ejecución concurrente de un conjunto de transacciones.
- **Operaciones conflictivas** en una historia
    - Dos operaciones de transacciones Ti y Tj (i != j), ambas commiteadas, se dicen conflictivas, si operan sobre el mismo ítem y al menos una de las dos es un write.
    - Por ejemplo, en H = r1[X] w2[X] c1 c2, las primeras dos operaciones son conflictivas.
- **Equivalencia de historias**
    - Dos historias H1 y H2 son equivalentes si:
        1. Están definidas sobre el mismo conjunto de transacciones.
        2. Las operaciones conflictivas tienen el mismo orden.
- **Historia serial**
    - Historia en la cual todas las operaciones de las transacciones no se entrelazan. Más específicamente, para todo par de transacciones Ti y Tj (i != j), todas las operaciones de Ti aparecen antes que las de Tj, o viceversa.
    - Las historias seriales nunca tienen problemas. Entonces, toda historia que sea equivalente a una serial tampoco debería tener problemas.
- **Serializabilidad**
    - Una historia se dice serializable (SR) si es equivalente a una historia serial.
    - Por ejemplo, la historia H = r1[X] w2[X] c1 c2 es serializable, pues es equivalente a H' = r1[X] c1 w2[X] c2.
    - La historia H = r1[X] w2[X] r2[Y] w1[Y] c1 c2 no es serializable.
- **Anomalías**
    - Comportamiento que no se puede manifestar en una historia **serial**.
- **Anomalías ANSI SQL**
    - **Dirty Read**. Una transacción lee un ítem escrito por otra transacción que aún no ha commiteado. Esto es: H = w1[X]...r2[X]...(c1 o a1)...
      Problema que puede generar: en la historia H, la transacción 1 aborta, lo cual deja a la transacción 2 con una lectura que era inválida.

- **Non-repeatable Read**. Una transacción hace dos lecturas de un ítem, y entre ellas otra transacción escribe ese mismo ítem. Esto es: H = r1[X]...w2[X]...r2[X]...
    - **Phantom Read**. Análogo a non-repeatable read, pero en lugar de leer un ítem, se leen una serie de ítems que satisfacen un predicado, y la escritura es sobre alguno de ellos. Esto es: H = r1[P]...w2[X in P]...r1[P]...
- Otras anomalías (no ANSI SQL)
    - **Lost Update.** Dos transacciones leen un ítem. Luego una de ellas lo escribe y la otra transacción lo sobreescribe, sin percatarse de la actualización. Esto es: H = r1[X] r2[X] w1[X] w2[X]...
    - **Write Skew.** Dos transacciones hacen lecturas concurrentes sobre ítems comunes, y luego hacen escrituras concurrentes sobre ítems distintos. Esto es: H = r1[X] r1[Y] r2[X] r2[Y] w1[X] w2[Y]...
      El problema de esto es que podemos terminar en un estado inconsistente, si había algún invariante que X e Y debían satisfacer en conjunto. Por ejemplo, si debiera ser X + Y <= 1, y empezáramos con X = 0, Y = 0, y ambos writes pusieran un 1, terminaríamos con los valores X = Y = 1, que no cumplen el invariante.
- Grafo de precedencia (Serializability Graph o SG)
    - Dada H sobre T1, …, Tn, SG(H) es un grafo dirigido con nodos T1, …, Tn, y tal que Ti -> Tj sii hay alguna operación de Ti que precede a alguna operación de Tj con la que conflictúa.
- **Teorema**. H es SR si y sólo si SG(H) es acíclico. Más aún, todo orden topológico de SG(H) es una historia serial equivalente a H.
- Un método naïve para garantizar serializabilidad sería ejecutar las transacciones, y cuando todas hayan terminado verificar si el grafo de la historia es acíclico. En caso negativo, reiniciar la ejecución.
    - Esto no es muy útil porque llegado el punto que hay que reiniciar la ejecución, ya es preferible ejecutar las transacciones serialmente en lugar de seguir perdiendo tiempo.
    - Queremos evitar anomalías de una forma más efectiva.
- Los métodos de control de concurrencia se clasifican en dos:
    - **Pesimistas**: basados en locking sobre los ítems, evitando anomalías por medio de la exclusión mutua del uso de los ítems.
    - **Optimistas**: basados en controlar las distintas versiones del valor de los ítems. Asumen que no ocurrirá un comportamiento no serializable y actúan para reparar el problema sólo cuando ocurre una violación aparente.
- **Locking**

- Un **lock** es un privilegio de acceso a un ítem de la BD. Una transacción puede adquirir o liberar un lock sobre un ítem.
- **Locking binario**
  - Modelo en el cual un lock tiene sólo dos estados:
    1. Locked X (o l[X]): privilegio para acceder (leer y escribir) a X
    2. Unlocked X (o u[X]): lock liberado
  - Una ejemplo de transacción en este modelo es T = l[X] r[X] l[Y] w[Y] u[X] u[Y] c.
  - En este modelo, no toda historia es válida, pues la adquisición de locks impone restricciones. Por ejemplo, la historia l1[X] w1[X] l2[X] r2[X] no es válida, pues la transacción 2 no puede adquirir el lock sobre X, pues lo tiene la transacción 1.
  - Decimos que una historia es **legal en el modelo de locking binario** si:
    1. Una transacción lee o escribe X sólo si posee un lock sobre X.
    2. Una transacción sólo hace lock de X si no hay otra transacción que en ese momento tenga un lock sobre X.
  - Queremos ver bajo qué condiciones una transacción legal es serializable en este modelo.
  - Consideramos el grafo de precedencia análogo para este modelo. Dada H sobre T1, …, Tn y legal, los nodos de SG(H) son T1, …, Tn, y Ti -> Tj sii Ti hace l[X] y luego Tj hace l[X].
  - Una historia es serializable sii su grafo de precedencia es acíclico.
- **Locking ternario**
  - Un lock tiene tres estados:
    1. Read locked X (o rl[X]): privilegio (compartido) para leer a X
    2. Write locked X (o wl[X]): privilegio (exclusivo) para leer y escribir a X
    3. Unlocked X
  - Las reglas de adquisición de locks en este modelo están dados por la siguiente matriz de compatibilidad de locking. En cada entrada indica si un lock puede ser concedido.

		Lo tiene Tj (i != j)	
		Read lock	**Write lock**
Pedido por Ti	**Read lock**	*SI*	*NO*
	Write Lock	*NO*	*NO*

  - Por ejemplo, T = rl[X] r[X] wl[Y] u[X] w[Y] u[Y] es una transacción válida en este modelo.

        - En algunos casos, una transacción puede solicitar un upgrade de un lock de lectura sobre X a un lock de escritura.
        - Decimos que una historia es **legal en el modelo de locking ternario** si:
          1. Una transacción lee X sólo si posee un lock de lectura sobre X.
          2. Una transacción escribe X sólo si posee un lock de escritura sobre X.
          3. Una transacción no puede adquirir un lock de lectura sobre X si otra transacción posee un lock de escritura sobre X actualmente.
          4. Una transacción no puede adquirir un lock de escritura sobre X si otra transacción puse un lock (cualquiera) sobre X actualmente.
        - Grafo de precedencia para este modelo. Dada H sobre T1, …, Tn y legal, los nodos de SG(H) son T1, …, Tn, y Ti -> Tj sii vale alguna de
            - Ti hace rl[X] o wl[X] y luego Tj hace wl[X].
            - Ti hace wl[X] y luego Tj hace rl[X].
        - Una historia es serializable sii su grafo de precedencia es acíclico.
    - Los locks introducen la posibilidad de deadlocks. Por ejemplo, si se quisieran ejecutar dos transacciones T1 = l1[X] l1[Y]... y T2 = l2[Y] l2[X]..., la BD podría comenzar a ejecutar las transacciones en el orden l1[X] l2[Y] y esto generaría un deadlock en cualquiera de las dos posibles próximas operaciones.
    - El uso de locks no previene automáticamente las anomalías. Hay algunas historias con locking ternario que son legales y no serializables.
    - **Protocolo 2PL (Two-Phase Locking)**
        - Una transacción es 2PL si todos los locks preceden al primer unlock.
        - **Teorema.** Sea H una historia sobre T1, …, Tn, tal que cada Ti es 2PL. Si H es legal entonces es SR.
- Manejo de deadlocks
    - Al usar locks podemos tener deadlocks.
    - Para detectar deadlocks podemos armar un **grafo de asignacion de recursos (Wait-for graph)** que tiene un nodo por cada transaccion T1, …, Tn, y hay un eje Ti -> Tj si la transaccion Ti actualmente esta pidiendo un lock de un item que tiene asignado (tiene un lock) Tj. Entonces, hay deadlock si y solo si el grafo tiene un ciclo.
    - Para desactivar el deadlock, debemos romper el ciclo. Para esto, abortamos una transaccion.
    - Debemos elegir que transaccion abortar. Criterios:

                - La que menos tiempo haya pasado en ejecucion.
                - La que menos cambios haya hecho en la base de datos.
                - La que menos tiempo le quede antes de terminar. Esto puede ser dificil o imposible de determinar.
                - Usando timestamps. A cada transaccion T le asignamos un timestamp Ts(T). Si Ts(T1) < Ts(T2) entonces T1 es mas vieja que T2. Si T1 esta esperando un recurso que tiene T2 (T1 -> T2 en el grafo)
                    - **Wait-Die.** Si T1 es la mas vieja, se queda esperando (wait). Si es la mas nueva, se aborta y recomienza más tarde con el mismo timestamp (die).
                    - **Wound-Wait.** Si T1 es la mas vieja, se aborta T2 y recomienza más tarde con el mismo timestamp (T1 wounds T2). Si es la más nueva, se queda esperando (wait).
- **Recuperabilidad**
    - Cuando una transacción aborta, debemos deshacer todos sus cambios. Esto puede ser complicado, debido a que hay otras transacciones ejecutándose concurrentemente. Veamos algunas situaciones en las que el rollback no es posible o genera conflictos.
    - Asumimos que una DB aborta restaurando la imagen de todos los ítems que modificó, justo antes de efectuar su primera modificación.
    - Ejemplo 1 (irrecuperabilidad).
      H1 = w1[X] r2[X] w2[Y] c2...
      Si ahora la transacción 1 aborta, también deberíamos abortar a la transacción 2, pues ha leído el valor de X que escribió 1, y que pudo haber usado para escribir Y. Sin embargo, la transacción 2 ya ha commiteado y no podemos abortarla.
      Para evitar esta situación deberíamos demorar el commit de 2 hasta que las transacciones de las cuales lee hayan abortado o commiteado.
    - Ejemplo 2 (aborts en cascada).
      H2 = w1[X] r2[X] w2[Y] a1…
      Similar al anterior, pero como aquí la transacción 2 aún no ha abortado, lo hacemos. En consecuencia, el abort de 1 causó el abort de 2.
      Para evitar esta situación deberíamos demorar la lectura de X hasta que las transacciones de las cuales lee hayan abortado o commiteado.
    - Ejemplo 3.
      H3 = w1[X] w2[X] a1 a2
      El abort de la transacción 1 debería volver atrás al valor que tenía X al principio de la historia. Luego se aborta la transacción 2, lo cual debería

deshacer la escritura. Sin embargo, esta escritura ya fue automáticamente deshecha por el abort de 1.
Para evitar esta situación deberíamos demorar la escritura de X hasta que las transacciones de las cuales lee hayan abortado o commiteado.

    - Decimos que una transacción **T1 lee de T2** (T1 != T2) en una historia si T1 efectúa una operación r1[X] y T2 es la última transacción que escribió X antes de dicha operación.
    - Siguiendo los tres ejemplos anteriores, vamos a clasificar las historias según su *nivel de recuperabilidad*:
        - **Historia recuperable.** H es recuperable (RC) si cada transacción hace su commit después de que hayan hecho commit todas las transacciones de las cuales lee.
        - **Historia que evita aborts en cascada.** H evita aborts en cascada (ACA) si lee únicamente de transacciones que ya hayan commiteado.
        - **Historia estricta.** H es estricta (ST) si lee o escribe únicamente de transacciones que ya hayan commiteado o abortado.
    - Teorema (de recuperabilidad). ST => ACA => RC.
    - El concepto de recuperabilidad es ortogonal al de serializabilidad. Hay historias recuperables (RC ó ACA ó ST) que no son serializables.
- Locking y recuperabilidad
    - Hay una variante del protocolo 2PL, llamada **2PL estricto (2PLE)**, que además de serializabilidad garantiza recuperabilidad estricta.
    - Decimos que T es 2PLE si es 2PL y además no libera ninguno de sus locks de escritura hasta después de haber hecho commit o abort.
    - **Teorema.** Una historia legal sobre transacciones 2PLE es SR y ST.

- **Métodos optimistas**
    - Asumen que no ocurrira un comportamiento no serializable, y solo actuan para resolver un problema cuando ha ocurrido una violacion aparente.
    - Asignar a cada transaccion T un timestamp Ts(T). Los timestamps van en orden ascendente.
        - Para generar los timestamps podemos usar el reloj del sistema o un contador ascendente.
    - Los timestamps determinan el orden de serializacion. Es decir, si se ejecutan transacciones T1, …, Tk, con timestamps en ese orden, se espera que la historia sea equivalente a una historia serial con las transacciones en ese orden.

- A cada item X de la base de datos le asociamos dos timestamps y un bit de commit:
    - RT(X): tiempo de lectura; el timestamp mas alto de una transaccion que ha leido X.
    - WT(X): tiempo de escritura; el timestamp mas alto de una transaccion que ha escrito X.
    - C(X): bit de commit; es verdadero si y solo si la transaccion mas reciente que escribio X ha commiteado.
- Se llama scheduler al componente encargado de manejar este concurrencia. Debe actualizar RT, WT y C cada vez que ocurre una operacion.
- Idea general: cada vez que ocurre una operacion de una transaccion Ti, el scheduler verifica si dicha operacion podria haber ocurrido si la transaccion se hubiera realizado instantaneamente en el momento de su timestamp.
    - Si no pudo haber ocurrido, decimos que el comportamiento es fisicamente irrealizable.
- Comportamientos fisicamente irrealizables:
    - Read too late
        - T quiere leer X, pero se tiene $Ts(T) < WT(X)$. En otras palabras, una transaccion posterior ya escribio X. Pero en el orden teorico, T deberia haber leido el valor de X anterior.
    - Write too late
        - T quiere escribir X, pero se tiene $Ts(T) < RT(X)$. En otras palabras, una transaccion posterior ya escribio X. Pero en el orden teorico, esa otra transaccion deberia haber leido el valor de X de T.
- Otro comportamientos que el scheduler evita:
    - Dirty reads
        - T quiere leer X, y $WT(X) < Ts(T)$ (fisicamente realizable). Sin embargo el bit C(X) esta apagado, o sea que la transaccion que escribio X no commiteo. El scheduler entonces retrasa la lectura de T hasta que la otra transaccion commitee o aborte.
- Reglas
    - El scheduler recibe una peticion r[X] de T
        - Si $Ts(T) < WT(X)$
            - Ocurre un read too late. Es fisicamente irrealizable.

      - Rollback de T (abortar y reiniciar con nuevo timestamp).
    - Si no, WT(X) <= Ts(T)
      - Si C(X) es true, realizar la lectura. Si es RT(X) < Ts(T), actualizar RT(X).
      - Si C(X) es false, hay que evitar dirty read. Demorar la lectura hasta que C(X) sea verdadero o la transaccion que escribio a X aborte.
  - El scheduler recibe una peticion w[X] de T
    - Si Ts(T) < RT(X)
      - Ocurre un write too late. Es fisicamente irrealizable.
      - Rollback de T.
    - Si RT(X) <= Ts(T)
      - Si Ts(T) < WT(X)
        - No deberiamos necesitar escribir X, pero hay que esperar a que la transaccion que escribio X la ultima vez commitee o aborte.
        - Si C(X) es true, ignorar la escritura.
        - Si C(X) es false, demorar T hasta que C(X) sea verdadero o la transaccion que escribio a X aborte.
      - Si WT(X) <= Ts(T)
        - Escribir X.
        - Asignar Ts(T) = WT(X).
        - Poner C(X) = false.
  - El scheduler recibe una peticion de c de T
    - Para cada item X escrito por T, hacer:
      - C(X) = true
      - Proseguir con las transacciones que esperan que X sea commiteado.
  - El scheduler recibe una solicitud de a (o rollback) de T
    - Restaurar valores previos de los elementos escritos por T.
    - Para cada transaccion esperando sobre un item X que T escribio, repetir el intento de lectura o escritura y verificar si ahora es legal.
- **Timestamping con multiversion**
  - Podemos evitar el problema de read too late, guardando multiples versiones de cada item.

        - Cada vez que ocurre un w[X] legal de una transaccion T, se crea una nueva version del item X. Su tiempo de escritura es t = Ts(T), y nos referimos a el como Xt.
        - Cuando una transaccion T hace r[X], el scheduler busca la maxima version Xt tal que t <= Ts(T).
        - Tambien guardamos cada momento de lectura (no solo el ultimo) de cada item Xt. Cuando una transaccion T hace r[X] y lee la version Xt, guardamos una marca Xt,tr, con tr = Ts(T).
        - Cuando una transaccion T hace w[X], el scheduler debe verificar que al agregar Xt, no haya ninguna lectura de X que debio haber sido Xt. Especificamente, revisa que no exista Xt’,tr con t’ < t (o sea, la version es mas vieja que Xt) y Ts(T) < tr (o sea, la transaccion tr efectivamente debio haber leido al menos la version de T).
        - Si una version Xt tiene tiempo de escritura t tal que no existe una transaccion activa T tal que Ts(T) < t, se pueden borrar todas las versiones de X previas a Xt.
    - Timestamping vs. locking
        - Timestamping es mejor cuando casi todas las operaciones son de lectura, o cuando no hay mucha contencion. Si hay mucha contencion, timestamping va a producir muchos rollbacks y demoras.
        - Locking es mejor cuando hay mucha contencion. Si hay poca contencion, el overhead del locking no es despreciable.
        - Se clasifica a las transacciones:
            - **Read-write**. Se manejan con locking pero crean versiones de los elementos.
            - **Read-only**. Se manejan con timestamping, usando los timestamps creados por las transacciones read-write.
- **Transacciones**
    - BEGIN TRANSACTION;
      …
      COMMIT; / ROLLBACK;
    - **Autocommit**. Modo en el que todo **comando** enviado a la BD es ejecutado como una transacción aparte, y commiteado implícitamente.
    - **Transacciones implícitas y explícitas.**
        - Explícita: la transacción comienza con un START y concluye con COMMIT o ROLLBACK.

        - Implícita: transacción cuyo principio y fin no están explícitos. Es todo comando al comienzo de una sesión, o posterior a un COMMIT o ROLLBACK de una transacción.
- Niveles de control de concurrencia típicos
    - El estándar ANSI SQL establece cuatro niveles de control de concurrencia (o sea, aislamiento). El más bajo de todos permite la ocurrencia de todas las anomalías. El más alto garantiza serializabilidad.

Isolation level	Dirty read	Nonrepeatable read	Phantom
Read uncommitted	Yes	Yes	Yes
Read committed *RCSI (SQLServer)*	No	Yes	Yes
Repeatable read	No	No	Yes
Serializable *Snapshot (SQLServer)*	No	No	No

- PostgreSQL usa control de concurrencia con multiversión.
    - Utiliza dos variantes de Snapshot Isolation.
    - **Snapshot Isolation (SI)**
        - La idea es que cada transacción toma una snapshot de la base al principio de su ejecución. Durante toda la transacción usará esos valores de la base.
        - Al comenzar una transacción, se genera un Snapshot Timestamp (ST).
        - Para cada ítem en la BD, se almacenan todas sus versiones, cada una con un timestamp distinto.
        - Cada vez que la transacción pide el valor de un ítem X, la BD le devuelve la última versión de X con timestamp <= ST.
        - Las escrituras de una transacción se hacen localmente (las otras transacciones no pueden ver sus cambios hasta que haga commit).
        - Al commitear, se genera un Commit Timestamp (CT). A continuación, se chequea que no haya conflictos entre las versiones de los ítems que modificó la transacción, con los ítems modificados por otras transacciones entre ST y CT.
        - Supongamos que X1, …, Xn son los ítems modificados. Si timestamp(Xi) < ST para todo i, se instalan los cambios de

la transacción. Si no, se aborta, y decimos que se produjo un **write-write conflict**.
        - SI no asegura serializabilidad. Asegura que no hay Dirty Read, Non-Repeatable Read ni Phantom Read.
    - **Read Commited Snapshot Isolation (RCSI)**
        - Análogo a SI, pero sólo se almacena la última versión de cada ítem, y no se chequean write-write conflicts al commitear.
        - Las escrituras se siguen haciendo localmente, y sólo se instalan al commitear. Esto asegura que sólo se leen versiones commiteadas, y por ende no hay Dirty Read.
        - Pueden haber Non-Repeatable Reads, pues durante una transacción podemos pasar a leer una nueva versión de un ítem, instalado por otra transacción que commiteó.
    - Postgre implementa Read Uncommited y Read Commited mediante RCSI, y Repeatable Read y Serializable con SI.
- SQL Server implementa los siguientes niveles de aislamiento:

Isolation level	Dirty read	Nonrepeatable read	Phantom	Concurrency control
Read Uncommitted	Yes	Yes	Yes	Pessimistic
Read Committed (locking)	No	Yes	Yes	Pessimistic
Read Committed (snapshot)	No	Yes	Yes	Optimistic
Repeatable Read	No	No	Yes	Pessimistic
Snapshot	No	No	No	Optimistic
Serializable	No	No	No	Pessimistic

    - Las implementaciones pesimistas las hace usando locks de transacción.
        - Un lock de transacción es aquel que se toma y no se lo libera hasta el final de la transacción.
        - Este concepto se aplica a tres tipos de locks:
            - Escritura (exclusive).
            - Lectura (shared).
            - Lectura en rango (shared).
        - Las reglas de compatibilidad de estos locks son las mismas que vimos antes, al hablar de concurrencia.
    - Read Commited: implementado con locks de escritura de transacción.

        - Repeatable Read: locks de escritura y lectura de transacción.
        - Serializable: locks de escritura y lectura en rango de transacción.

Recuperacion ante fallas (logging)

- **Objetivo**. Llevar a la base de datos a un estado consistente, luego de que ocurra una falla en el sistema.
- Las tecnicas de recuperacion se basan en mantener un **log**, en el cual se deja registro de los cambios sobre la BD. Ante una falla, se lee el log y se realizan las operaciones necesarias para volver a un estado consistente (todos los cambios son de transacciones que terminaron completamente, y no que quedaron a medias).
- Idempotencia de la recuperacion. Los pasos de recuperacion son idempotentes, i. e. ejecutarlos muchas veces tiene exactamente el mismo efecto que ejecutarlos una vez. Esta propiedad es necesaria en caso que crasheemos durante la recuperacion.
- **Log (o journal)**
    - Archivo append-only al que se le van agregando registros con los eventos ocurridos en la BD.
    - Tipos de registros:
        - <START T>. La transaccion T ha comenzado.
        - <COMMIT T>. La transaccion T commiteo.
        - <ABORT T>. La transaccion T aborto.
        - Registros de operaciones de read y write de una transaccion.
- Las transacciones pueden estar completas o no. Vamos a querer recuperar las transacciones completas, y borrar todo rastro de las incompletas.
    - **Transacciones completas.** Aquellas transacciones con registro de COMMIT o ABORT.
    - **Transacciones incompletas.** Aquellas con registro de START pero sin registro de COMMIT ni ABORT.
- Metodos de recuperacion
    - **Undo logging.** Deshacer las transacciones incompletas. Asumir que las transacciones completas fueron escritas en disco.
    - **Redo logging.** Rehacer las transacciones completas. Asumir que las transacciones incompletas no fueron escritas en disco (ni un poquito).
    - **Undo/Redo logging.** Deshacer las transacciones incompletas y rehacer las completas. No se asume nada sobre que cosas se escribieron en disco.

- **Politica Undo**
    - Deshacer las transacciones incompletas.
    - Usa registros de la forma <T, X, ValorAnterior>.
    - Reglas de logging:
        - Escribe cada registro <T, X, v> en disco, **antes** que escribir el valor v de X en la BD.
        - Escribe el <COMMIT T> **despues** de escribir todas las escrituras de T en la BD.
    - Recuperacion:
        - Recorrer el log de abajo hacia arriba, recordando las transacciones T para las cuales ha visto un <ABORT T> o <COMMIT T>.
        - Para cada <T, X, v> encontrado:
            - Si ya vimos un ABORT o COMIT de T (o sea, esta completa), no hacer nada.
            - Si no, T esta incompleta. Cambiar el valor de X a v en la BD.
        - Escribir un registro <ABORT T> por cada T incompleta que no fue previamente abortada. Flushear el log.
    - Ventajas:
        - En general, hay pocas transacciones incompletas. Tendremos que deshacer pocos cambios.
        - Podemos flushear datos a la BD durante la escritura del log. No hace falta demorar la escritura en BD hasta el final.
    - Desventajas:
        - Para recuperar, hay que leer todo el log.
        - No podemos demorar la escritura en la BD mas alla del fin de la transaccion. Esto es una desventaja, pues a veces es mejor acumular una gran cantidad de cambios de transacciones y flushearlos todos juntos.
- **Politica Redo**
    - Rehacer las transacciones completas.
    - Usa registros de la forma <T, X, ValorNuevo>
    - Reglas de logging:
        - El registro <COMMIT T> se escribe en disco **antes** que cualquier elemento modificado por T.
    - Recuperacion:
        - Escanear el log, identificando las transacciones commiteadas y abortadas.
        - Recorrer el log de arriba hacia abajo.
        - Para cada <T, X, v> encontrado:

                - Si hay un <COMMIT T> posterior, escribir el valor v para X en la BD.
                - Si no, no hacer nada.
            - Para cada transaccion incompleta T, escribir un registro <ABORT T> en el log. Flushear el log.
    - Ventajas:
        - ?
    - Desventajas:
        - Para recuperar, hay que leer todo el log. Mas aun, hay que escanearlo dos veces.
        - En general, hay muchas transacciones completas. Tendremos que rehacer muchos cambios.
        - Debemos demorar la escritura en la BD, hasta luego de escribir el COMMIT en disco. Esto implica mantener todos los bloques de la BD modificados en el buffer, hasta el COMMIT.
- **Politica Undo/Redo**
    - Provee flexibilidad para flushear los cambios de la BD a disco. Podemos flushear durante la transaccion, o demorar el flush hasta pasado el COMMIT.
    - Deshace todas las transacciones incompletas, y rehace las completas.
    - Escribe registros de la forma <T, X, ValorAnterior, ValorNuevo>.
    - Reglas de logging:
        - Los registros <T, X, v, w> se escriben en disco antes que las modificaciones a la BD.
    - Recuperacion:
        - Aplicar Redo a todas las transacciones commiteadas, de arriba hacia abajo.
        - Aplicar Undo a todas las transacciones incompletas, de abajo hacia arriba.
- Checkpointing
    - Todas las políticas de recovery que vimos tienen que escanear todo el log durante la recuperación.
    - Checkpointing consiste en introducir marcas en el log, que dan garantías de cierta información ya está escrita en disco, en ese punto.
    - Tipos
        - Quiscente. No aceptan nuevas transacciones durante el checkpoint.
        - No Quiescente. Aceptan nuevas transacciones durante el checkpoint.

- **Undo quiescente**
  - Reglas de checkpointing:
    - Se decide comenzar el checkpoint. Se dejan de aceptar nuevas transacciones.
    - Esperar a que todas las transacciones activas (aquellas con START pero sin COMMIT o ABORT) terminen.
    - Escribir un <CKPT> en el log y flushear.
    - Aceptar nuevas transacciones.
  - Semántica del <CKPT>
    - Al momento del CKPT, todas las transacciones están terminadas y escritas en disco.
  - Recuperación:
    - Análoga a la recuperación Undo tradicional, pero ahora sólo hace falta leer hasta el punto de checkpoint.
- **Undo no quiescente**
  - Reglas de checkpointing:
    - Se decide comenzar el checkpoint. Si las transacciones activas en ese momento son T1, …, Tk, se escribe un registro <START CKPT(T1, …, Tk)> y efectuar un flush.
    - Esperar a que todas las transacciones T1, …, Tk terminen, pero sin prohibir que empiecen otras transacciones.
    - Escribir <END CKPT> en el log y flushear.
  - Semántica del <START CKPT(T1, …, Tk)> <END CKPT>
    - Al momento del START CKPT, todas las transacciones terminadas están escritas en disco, y las únicas activas son T1, …, Tk.
    - Al momento del END CKPT, las transacciones T1, …, Tk terminaron y están escritas en disco. Se pueden haber iniciado otras transacciones entre el START CKPT y END CKPT.
  - Recuperación:
    - Análoga a Undo tradicional, pero ahora al encontrar un <End CKPT> debemos leer no más allá de su <START CKPT(...)>.
    - Si encuentro un <START CKPT(T1, …, Tk)> sin un <End CKPT>, debo leer hasta el <START Ti> (i = 1, …, k) más antiguo de una Ti que esté incompleta.
- **Redo no quiescente**
  - Reglas de checkpointing:

            - Se decide comenzar el checkpoint. Si las transacciones activas son T1, …, Tk, se escribe un registro <START CKPT(T1, …, Tk)> y efectuar un flush.
            - Esperar a que todas las modificaciones realizadas por transacciones ya commiteadas al momento del START CKPT sean escritas en disco.
            - Escribir <END CKPT> en el log y flushear.
        - Semántica del <START CKPT(T1, …, Tk)> <END CKPT>
            - Al momento del START CKPT, puede haber algunas transacciones terminadas que no terminaron de escribirse en disco, y las únicas activas son T1, …, Tk.
            - Al momento del END CKPT, todas las transacciones terminadas que no habían terminado de escribirse en disco en el momento del START CKPT están efectivamente flusheadas. Notar que no sabemos nada del estado de las activas al momento del START CKPT.
        - Recuperación:
            - Leer el log de abajo hacia arriba, buscando un <END CKPT>. Si lo encuentro, leemos hasta el <START CKPT(T1, …, Tk)> asociado. Luego, hacemos una recuperación Redo tradicional, desde el <START Ti>, i = 1, …, k, más lejano tal que Ti haya hecho COMMIT.
            - Si encontramos un START CKPT sin un END CKPT, lo ignoramos.
    - **Undo/Redo no quiescente**
        - Reglas del checkpointing:
            - Se decide comenzar el checkpoint. Si las transacciones activas son T1, …, Tk, escribir un <START CKPT(T1, …, Tk)> en el log y efectuar un flush.
            - Esperar a que todos los buffers sucios (cambios que no fueron escritos en disco) hasta el START CKPT sean flusheados.
            - Escribir <END CKPT> en el log y flushear.
        - Semántica del <START CKPT(T1, …, Tk)> <END CKPT>
            - Al momento del START CKPT, puede tanto transacciones terminadas como activas (T1, …, Tk) que no terminaron de escribirse en la BD.
            - Al momento del END CKPT, todas las transacciones terminadas al momento del START CKPT han sido

flusheadas. Las T1, …, Tk no necesariamente fueron flusheadas completamente.

- Recuperación:
    - Leer el log de abajo hacia arriba, buscando un <END CKPT>. Si lo encuentro, busco el <START CKPT(T1, …, Tk)> asociado. Hacemos una recuperación Undo desde el <START Ti>, i = 1,..., k, más viejo tal que Ti. Hacemos una recuperación Redo desde el START CKPT.
    - Si encontramos un START CKPT sin END CKPT, lo ignoramos.

Optimización de consultas

- Una query es procesada en varias partes por el motor de base de datos:
    - Parsing y traducción. Descompone la query en partes. Construye una expresión algebraica equivalente.
    - Optimizador de Consultas. Analiza distintas formas de resolver la consulta, evaluando sus costos, y seleccionando la más eficiente. Elabora un plan de ejecución.
    - Procesador de Consultas. Lleva adelante el plan de ejecución elaborado por el optimizador.
- Nos vamos a concentrar en el Optimizador de Consultas.
- Hay muchos planes de ejecución posibles, cada uno con un rendimiento asociado muy diferente. El optimizador utiliza heurísticas para encontrar un buen plan.
- En el contexto de la ejecución de una query, la operación que domina el costo es el acceso a disco. Por esto, queremos minimizar la cantidad de lecturas de bloques de disco.
- Idea general de la optimización:
    - Tomar la query SELECT-FROM-WHERE, y armar una expresión en álgebra relacional del siguiente modo:
        - Hacemos el producto cartesiano de las tablas del FROM.
        - Seleccionamos con las condiciones del WHERE.
        - Proyectamos las columnas del SELECT.
    - Construímos un árbol binario, donde las hojas son relaciones, y los nodos internos operaciones que se aplican a su/s hijo/s. Este árbol representa un plan de ejecución.

- Efectuar modificaciones al árbol (y por ende, en el plan de ejecución) de modo de minimizar la cantidad de accesos disco. Estas modificaciones se basan en heurísticas, que pueden ser de varios tipos:
  - Heurísticas basadas en propiedades del álgebra relacional. Por ejemplo, *bajar* las selecciones y proyecciones hacia las hojas (conmutatividad de la selección o proyección con el producto cartesiano o junta).
  - Heurísticas basadas en estadísticas del catálogo.
    - Tamaño de archivos y *factor de bloqueo*.
    - Cantidad de tuplas de una relación.
    - Cantidad de bloques de una relación.
    - Cantidad de valores distintos de una columna.
  - Elección del método para realizar, a nivel físico, una operación de álgebra relacional.
    - Cómo efectuar un join, proyección, selección, etc.
    - Puede haber varias formas de realizar una operación, lo cual depende de ciertos factores:
      - Qué *índices* tenemos. Una relación puede tener varios índices y de distinto tipo.
      - Qué operaciones soporta el Procesador de Consultas.

- Parámetros para medir (R es una relación):
  - *Bloque*: porción de datos que se levanta en cada lectura de disco.
  - LB: longitud de bloque.
  - B_R: cantidad de bloques que ocupa R.
  - FB_R: factor de bloqueo de R. Es la cantidad de tuplas de R que entran en un bloque.
  - L_R: longitud de una tupla de R.
  - T_R: cantidad total de tuplas de R.
  - $I_{R,A}$: imágen del atributo A de R. Es la cantidad de valores distintos de la columna A en R.
  - X_I: altura del árbol de búsqueda I (I un índice árbol B+)
  - FB_I: factor de bloqueo de I. Es la cantidad de tuplas de I que entran en un bloque.
  - BH_I: cantidad de bloques que ocupan todas las hojas de I (I un índice árbol B+).
  - $MBxB_I$: cantidad máxima de bloques en un bucket de I (I un índice hash).
  - CBu_I: cantidad de buckets de I (I un índice hash).
  - B: cantidad de bloques que entran en memoria principal.

- Se asume que:
  - En cada bloque de R sólo hay tuplas de R.
  - La longitud de las tuplas es fija para cada relación.
- Archivos
  - Una tabla se almacena en un archivo.
  - El archivo está formado por las tuplas de la relación, en algún orden.
  - Cada tupla tiene asociado un identificador, llamado rid (register id). Este rid no es un atributo de la relación.
- Índice
  - Es un diccionario (con claves no necesariamente únicas) asociado a una relación.
    - La clave es una o más columnas de la relación.
    - El valor asociado a una clave es el registro (la tupla) asociado a esa clave. El valor puede tener varias formas:
      - El registro completo.
      - El rid del registro.
      - Una lista de los rid de todos los registros con esa clave.
  - Representaciones:
    - Árbol B+
      - Árbol de búsqueda balanceado, con una cantidad de hijos y claves por nodo dada por un parámetro d (cada nodo interno, excepto la raíz, tiene entre d/2 y d claves e hijos).
      - Las claves y valores están en las hojas. Las hojas se pueden recorrer como si formaran una lista doblemente enlazada.
      - Ideales para responder consultas en rango, i. e. encontrar todas las claves k tal que a <= k <= b.
    - Hash
      - Tabla de hash, formada por una cantidad estática de buckets.
      - Ideales para búsqueda por igualdad, i. e. encontrar todas las claves k tal que k = a.
  - Propiedades:
    - Índices clustered vs. unclustered
      - Clustered: los datos del archivo están ordenados físicamente en el mismo orden que el índice.
      - Unclustered: no clustered.
      - Ejemplo:
        - Archivo ordenado por DNI.

            - Índice por DNI es clustered.
            - Índice por nombre es unclustered.
        - A lo sumo puede haber un índice clustered, pues el orden de los datos en el archivo es único.
    - Índices densos vs. no densos
        - Denso: tienen una entrada por cada posible clave de búsqueda del archivo de datos.
        - No denso: algunas claves posibles del archivo no están en el índice.
        - Ejemplo:
            - Tabla de estudiantes de computación del DC.
            - Índice de estudiantes mujeres es no denso.
            - Índice de estudiantes por DNI es denso.
    - Índices primarios vs. secundarios
        - Primario: los valores que almacena son los registros completos de los archivos.
        - Secundario: los valores son rids.
- Tipos de operaciones físicas que nos interesa conocer:
    - Escaneo completo de un archivo.
        - Cuando hacemos una proyección o una selección, e iteramos sobre cada uno de los registros.
    - Búsqueda de una clave por igualdad.
        - Cuando hacemos una selección, y esa selección tiene una condición de la pinta (k = a and …), podemos buscar los registros con clave k = a.
        - Cuando hacemos un join entre R y S sobre una clave k, recorremos los registros de R y, suponiendo que en un registro vale k = a, debemos buscar todos los registros de S con k = a.
    - Búsqueda de una clave por rango.
        - Ejemplos análogos.
- **Tipos de archivos**
    - Heap files
        - Colección desordenada de registros, agrupados en bloques.
        - Operaciones:
            - Escaneo
            - Búsqueda por igualdad
                - Escanear todo.
            - Búsqueda por rango
                - Escanear todo.

- Sorted files
    - Colección de registros, ordenados de acuerdo a los valores de ciertos campos.
    - Operaciones:
        - Escaneo
        - Búsqueda por igualdad
            - Búsqueda binaria de la primer ocurrencia.
        - Búsqueda por rango
            - Búsqueda binaria del primer elemento en el rango.
- **Tipos de índices**
    - Siempre usaremos índices densos y primarios.
    - Árbol B+ clustered
        - Escaneo (del archivo)
            - El índice no ayuda.
        - Búsqueda por igualdad
            - Bajar en el árbol, buscando la primer ocurrencia de la clave. Ir a buscar el registro asociado en el archivo. Recorrer el archivo a partir de ese punto, mientras encuentre ocurrencias de la clave.
        - Búsqueda en rango
            - Buscar el primer elemento del rango y luego recorrer el archivo a partir de ahí.
    - Árbol B+ unclustered
        - Escaneo (del archivo)
            - El índice no ayuda.
        - Búsqueda por igualdad
            - Bajar en el árbol, buscando la primer ocurrencia de la clave. A partir de esa clave en esa hoja, recorrer secuencialmente las hojas, mientras encuentre ocurrencias de la clave. Por cada clave encontrada, leer el registro asociado en el archivo.
        - Búsqueda en rango
            - Buscar el primer elemento del rango, y a partir de ahí todas las ocurrencias. Por cada una, leer el registro asociado en el archivo.
    - Hash
        - Escaneo (del archivo)
            - El índice no ayuda.
        - Búsqueda por igualdad

            - Hashear la clave y recorrer linealmente el bucket. Cada vez que encontramos una ocurrencia de la clave, ir a leer el registro asociado en el archivo.
        - Búsqueda en rango
            - No sirve.
- Tipos de escaneos
    - **File scan.** Recorrido sobre las entradas de un archivo.
    - **Index scan.** Recorrido sobre las entradas de un índice.
- Index matching
    - Tenemos una selección y queremos usar un índice.
    - Podemos hacerlo en dos casos:
        - Tenemos un índice árbol B+, y el predicado es una conjunción de términos (atrib op valor) (op es =, < ó >)que involucra a atributos de un prefijo de la clave del índice (o sea, si la clave es una tupla, involucra a todos los atributos que forman un prefijo de la tupla).
        - Tenemos un índice hash, y el predicado es una conjunción de términos que involucra a todos los atributos del índice.
- Selectividad
    - Será de nuestro interés tener una estimación de cuántas tuplas de una relación satisfacen un predicado dado.
    - Llamamos selectividad de un predicado P (y lo notamos sel(P)) a la proporción de tuplas de una relación que satisfacen P. Esto es, sel(P) = (tuplas que cumplen P) / (todas las tuplas).
    - Si X es clave candidata, entonces sel(X = a) = $1/T_R$.
- Costo de input y output
    - Para calcular el costo de un árbol de ejecución, sumamos todos los costos de input y output de cada una de las operaciones.
    - El costo de input de una operación (nodo interno) es la cantidad de accesos a disco que hace al leer el input.
    - El costo de output se define análogamente.
    - Las operaciones que vamos a analizar aquí son únicamente selección, proyección y join. El costo de cada una dependerá de cómo sea implementada físicamente. Analizaremos el costo de input y output para las tres.
- **Proyección**
    - Input
        - Escaneo de archivo
        - Búsqueda lineal en árbol B+

            - Todos los atributos a proyectar deben formar parte de la clave del índice.
            - Acceder al primer nodo hoja, y recorrer todas las hojas del índice, quedándonos con los atributos de cada clave del índice.
        - Búsqueda lineal en hash
            - Todos los atributos a proyectar deben formar parte de la clave del índice.
            - Recorrer todos los bloques de todos los buckets del índice, quedándonos con los atributos de cada clave del índice.
        - CI = costo del procedimiento utilizado
    - Output
        - Llamemos Q a la relación output.
        - La cantidad de registros de Q es la misma que tenía la relación original. El costo del output está dado por la longitud de las nuevas tuplas.
        - La longitud de una tupla de la proyección, es la suma de las longitudes de los atributos proyectados.
        - Calcular el FB_Q con esta nueva longitud de tupla, y con ésto la cantidad de bloques B_Q.
        - CO = B_Q.
- **Selección**
    - Llamemos P al predicado de selección, que selecciona a través de un atributo k (entre otras cosas).
    - Input
        - Escaneo de archivo
        - Búsqueda binaria en archivo ordenado
            - El archivo debe estar ordenado según el atributo k.
        - Búsqueda en un índice B+ clustered
            - El índice debe tener como clave a k.
        - Búsqueda en un índice B+ unclustered
            - El índice debe tener como clave a k.
        - Búsqueda en un índice hash
            - El índice debe tener como clave a k.
        - Intersección de rids basado en hash
            - P debe ser una conjunción (P1 y P2), tal que P1 coincide con un índice I1, y P2 con un índice I2.
            - Buscar en I1 las tuplas que cumplan P1. Los rids encontrados se escriben en un archivo hash intermedio.

            - Lo mismo se hace sobre I2, verificando para cada rid encontrado, si está en el archivo hash intermedio. En caso afirmativo, escribirlo en un segundo archivo hash.
            - Recorremos el segundo archivo hash, y para cada rid obtener la tupla asociada.
        - Unión de rids basado en hash
            - Análogo, pero para cada rid encontrado en I2, agregar al primer (único) archivo hash intermedio, si no lo encontramos en tal archivo.
    - Output
        - Llamemos Q a la relación output.
        - La cantidad de tuplas de Q depende de la selectividad del predicado de selección.
        - Asumimos que todos los valores de una columna son equiprobables. Esto es, si hay $I_{R, A}$ valores distintos para A, todos tienen probabilidad $1 / I_{R, A}$ de ocurrir.
        - Si la selección es sobre una tabla R, con el predicado P, se tiene $T_Q = T_R * sel(P)$.
            - Caso P = (A = a)
                - $sel(P) = 1 / I_{R, A}$
            - Caso P = (A > a)
                - sel(P) = dist(a, A_max) / dist(A_min, A_max)
                - Si no hay noción de dist, se estima sel(A > a) = 1/2.
            - Caso P = (a < A < b)
                - sel(P) = dist(a, b) / dist(A_min, A_max)
            - Caso P = (A in {a1, …, an})
                - $sel(P) = n / I_{R, A}$
            - Caso P = (A = a and B = b)
                - $sel(P) = P(A = a) P(B = b) = 1 / (I_{R, A} * I_{R, B})$
            - Caso P = (A = a or A = b)
                - sel(P) = P(A = a) + P(B = b) - P(A = a and B = b)
        - La longitud de las tuplas de Q es la misma que la de las tuplas originales.
        - Con todo esto ya podemos calcular B_Q, y éste es el CO.
- **Join**
    - Queremos computar R JOIN S.
    - Input
        - Block Nested Loops Join (BNLJ)

            - Suponemos que se tienen B bloques de memoria disponibles.
            - Usamos B - 2 bloques para leer tuplas de R, 1 bloque para leer tuplas de S y 1 bloque para escribir tuplas resultantes del join.
            - Para cada segmento de B - 2 bloques de R, para cada bloque de S, joineamos el segmento de R contra el bloque de S.
        - Index Nested Loops Join (INLJ)
            - Suponemos que S tiene un índice I sobre una clave k, y que el predicado del join coincide con I.
            - Para cada bloque de R, para cada tupla t en dicho bloque, buscar el valor de k en t, en el índice I de S.
        - Sort Merge Join (SMJ)
            - Ordenar las relaciones R y S, usando un multi-way merge.
            - Luego hacemos un merge entre ambas relaciones.
    - Output
        - Suponemos que la condición del join es R.ri = S.sj.
        - Llamemos Q a la relación output.
        - La cantidad de tuplas de Q se estima como $T_Q = (T_R * T_S) / \max(I_{R, ri}, I_{S, sj})$. La idea de la estimación es dividir R en $I_{R, ri}$ grupos de R / $I_{R, ri}$ tuplas, y cada grupo se joinea contra un grupo de S / $I_{S, sj}$ tuplas de S. Notar que para que esto tenga sentido, debe haber menos grupos de tuplas de R que de S, o sea $I_{R, ri} <= I_{S, sj}$, pues no hay dos grupos de R que matcheen contra el mismo grupo de S. Es por esto que aparece el max en el denominador.
        - La longitud de las tuplas es $L_Q = L_R + L_S$. Si el join es natural, hay que restar la longitud de los campos comunes entre R y S.
        - Con todo esto ya podemos calcular B_Q, y éste es el CO.
- Materialización y pipelining
    - **Materialización**. Es escribir el resultado de una operación en disco.
    - **Pipelining**. Si entre dos operaciones tenemos un pipeline, la primer operación O1 va pasandole las tuplas resultado en forma de stream, a la segunda operación O2. Este pasaje se hace completamente en memoria, lo cual evita escribir el resultado de la primer operación en disco. Por ende, el costo de output de O1 es 0, al igual que el costo de input de O2.
    - No toda operación se puede pipelinizar. Hay operaciones que necesitan tener todas las tuplas de una relación de antemano, para poder empezar a realizar el cómputo. Por ejemplo, los joins (**siempre hay que**

materializar el input de los joins). En general, la capacidad para pipelinizar depende de las implementaciones de las operaciones disponibles en el Procesador de Consultas.

- Órdenes interesantes
    - Decimos que un resultado intermedio está en un orden interesante, si su orden coincide con el orden de un ORDER BY, o GROUP BY, o DISTINCT o JOIN (es decir, si se puede usar para hacer join, por ejemplo usando un SMJ) de nivel superior.
- Optimizaciones algebraicas
    - Buscan mejorar la performance de la consulta independientemente de la representación física. Se basan en propiedades algebraicas que transforman la consulta en AR en una equivalente.
    - Propiedades:
        - Cascada de σ
            - $\sigma_{C1 \text{ and } C2}(R) = \sigma_{C1}(\sigma_{C2}(R))$
        - Conmutatividad de σ
            - $\sigma_{C1}(\sigma_{C2}(R)) = \sigma_{C2}(\sigma_{C1}(R))$
        - Cascada de π
            - $\pi_{L1 \cap L2}(R) = \pi_{L1}(\pi_{L2}(R))$
        - Conmutatividad de σ con respecto a π
            - $\sigma_{C}(\pi_{A1, \ldots, An}(R)) = \pi_{A1, \ldots, An}(\sigma_{C}(R))$
              siempre que C sólo use atributos de A1, …, An.
        - Conmutatividad del producto cartesiano (o junta)
            - R x S = S x R
        - Conmutatividad de la unión e intersección
            - R op S = S op R
              donde op es $\cup$ o $\cap$.
        - Asociatividad de producto cartesiano, junta, unión e intersección
            - (R op S) op T = R op (S op T)
              donde op es x, join, $\cup$ o $\cap$.
        - Conmutatividad de σ con respecto al producto cartesiano (o junta)
            - $\sigma_{C}(R \times S) = \sigma_{C1}(R) \times \sigma_{C2}(S)$
              donde C1 $\cup$ C2 = C, y cada Ci son las condiciones que tienen sentido en cada relación.
        - Conmutatividad de π con respecto al producto cartesiano (o junta)
            - $\pi_{L}(R \times S) = \pi_{L1}(R) \times \pi_{L2}(S)$
              donde L1 $\cup$ L2 = L, y cada Li son los atributos que tienen sentido en cada relación.

- Heurísticas varias
    - Considerar sólo árboles sesgados a izquierda. Ésto limita los árboles candidatos a analizar.
    - Descomponer las selecciones conjuntivas. Esto puede ayudar, por ejemplo, si tenemos que seleccionar con una condición sobre dos atributos A y B, pero sólo tenemos un índice sobre A, podemos separar las selecciones y usar el índice.
    - Llevar las selecciones lo más cercano posible a las hojas del árbol. Ésto ayuda a reducir la cantidad de datos que hay que materializar hacia arriba, seleccionando prematuramente las tuplas que nos interesan.
    - Reemplazar productos cartesianos seguidos de selecciones por joins. Evitar productos cartesianos.
    - Descomponer las listas de atributos de proyecciones y llevarlas lo más cerca de las hojas posibles.
    - Realizar primero los joins más selectivos.
    - Usar pipeline entre operaciones siempre que sea posible.

Teóricas del final (resumen^N)

Mapeo Objeto Relacional

- Modelo relacional: soporta queries sofisticadas.
- Modelo orientado a objetos: permite crear tipos de datos complejos.
- Idea: mezclar ambos modelos para obtener una BD que permita dominios (tipos de los atributos) no atómicos. Por ejemplo arreglos, sets, tuplas, etc.
- Retiene el fundamento matemático del modelo relacional.
- Viola la 1FN (todos los argumentos son atómicos).
- Permite atributos complejos: colecciones (arreglos, sets, etc.) y structs (tipos de datos definidos por el usuario).
- Permite features de POO, como herencia.
- Una relación que contiene tipos complejos se denomina anidada (nested).
- Unnesting. Transformación de una relación en una que está en 1FN (no tiene atributos de tipo complejo).
- Nesting. Proceso opuesto a unnesting. Es la creación de atributos de tipo complejo.
- También podemos usar atributos de tipo referencia. Estos atributos son una referencia verdadera a objetos en otras tablas. Esto permite evitar hacer joins. En lugar de eso, usamos **path expressions** (atrib_1 -> atrib_2 para viajar del atrib_1 de una tabla, al objeto apuntado atrib_2 de otra tabla).

XML

- Datos estructurados
  - Representados con un formato estricto.
  - Ejemplo: datos de una BD.
- Datos semiestructurados
  - Poseen cierta estructura, aunque no toda la información tiene necesariamente la misma estructura.
  - Información del esquema mezclada con los valores de los datos.
  - Self-describing data. Esquemas que por sí mismos describen la información que contienen. Por ejemplo JSON.
- Datos no estructurados
  - Limitada indicación sobre los datos que contiene.
- XML
  - Extensible Markup Language
  - Documentos dan la estructura de los datos a través de tags HTML.
  - <un_tag> … </un_tag>
  - Un documento XML se puede pensar como un árbol. Los tags son los nodos, y los hijos de un nodo son los tags anidados. La raíz es el primer tag.
  - Como es extensible (podemos definir nuestros propios tags) es utilizada para transmitir información (y no sólo almacenar documentos semi-estructurados) en diversas áreas.
  - Document Type Descriptors (DTD): forma de describir la estructura que debe tener un XML.
  - Document Object Model (DOM): mecanismo OO para representar y manipular documentos XML.
- Queries en XML
  - XPath
    - Consultas utilizando caminos en el árbol XML.
  - XQuery
    - for … let … where … order by … result …
    - for = SQL from
    - where = SQL where
    - order by = SQL order by
    - result = SQL select
    - let permite variables temporales; no tiene equivalente SQL

Data Warehousing

- Colección de datos orientada a un determinado ámbito, variable en el tiempo, no volátil e integrado, que ayuda a la toma de decisiones en el ámbito donde se utiliza (también llamado *business intelligence*).
    - **Orientada a temas** (subject oriented). Los datos del DW proveen información sobre temas en particular. Los datos relativos al mismo tema están agrupados.
    - **Variable en el tiempo.** Los cambios producidos a lo largo del tiempo quedan registrados en el DW. Esto permite conocer cómo eran los datos en un momento particular del tiempo (una snapshot de ese momento) y así saber cómo variaron los datos a lo largo del tiempo.
    - **No volátil**. La información no se modifica ni se elimina una vez almacenada en el DW. Se mantiene para futuras consultas.
    - **Integrado**. El DW contiene datos de todas las áreas de la organización. Dichos datos son integrados en forma consistente en el DW.
- Data Marts. Subconjunto del DW orientado a una finalidad específica del negocio (marketing, finanzas, producción, etc.)
- Explotación del DW. **Extract, Transform, Load (ETL)**
    - Se debe obtener los datos de las fuentes, limpiarlos, convertir su formato y cargarlos en el DW.
    - Etapas:
        - Migración. Tomar los datos de los sistemas operacionales y llevarlos a un área de preparación (staging area).
        - Limpieza.
            - Corregir, estandarizar y completar los datos.
            - Identificar datos redundantes.
            - Identificar outliers.
            - Consolidar tipos de datos.
        - Transformación.
            - Agregar metadata a los datos. El objetivo es facilitar el acceso e interpretación posterior del contenido del DW.
                - Fuente
                - Descripción de operaciones de transformación
            - Desnormalización de los datos.
            - Sumarizaciones. Muestran los datos de una manera más resumida. Permiten:
                - Facilitar las agregaciones.
                - Facilitar el análisis.

            - Proveer varias vistas del mismo conjunto de datos detallados (dimensiones en el “modelo dimensional” que se describe más luego).
        - Carga.
            - Carga de los datos en un soporte físico (por ejemplo una BD).
            - Métodos:
                - Full Refresh. Cargamos todos los datos de vuelta.
                - Incremental. Sólo cargamos los últimos datos.
        - Conciliación o validación.
            - Verificar integridad de los datos. Deben ser correctos y estar completos.
            - Métodos:
                - Completa, al final del proceso.
                - Por etapas, a medida que se cargan los datos.
    - Herramientas para la explotación
        - OLAP (Online Analytical Processing)
            - Herramienta para realizar consultas sobre grandes volúmenes de datos. Utiliza estructuras multidimensionales (cubos OLAP) que contienen sumarizaciones de grandes bases de datos.
        - Procesos manuales. Por ejemplo consultas SQL o programas que consulten la BD.
- **Modelado de datos**
    - Es importante porque determina la forma en que voy a poder consultar y visualizar los datos.
    - Técnicas:
        - Modelo ER: entidades, relaciones, atributos.
        - Modelo Dimensional: hechos, dimensiones, medidas.
    - Modelo Dimensional
        - Hechos. Colección de ítems de datos. Cada hecho representa una transacción o un evento del negocio.
        - Dimensión. Colección de miembros del mismo tipo (cosas relacionadas).
            - Cada hecho está conectado a **una** dimensión.
            - Determinan el contexto de los hechos.
            - Por ejemplo: Tiempo es una dimensión de miembros (Meses, Trimestres, Años).

            - Una dimensión puede estar formada por una jerarquía de miembros (o sea, no necesariamente es una lista de miembros).
        - Medida. Es un valor numérico de un hecho que representa el desempeño de un hecho relativo a una dimensión.
        - Las dimensiones son una forma de clasificar los datos. Los datos son las medidas, y los definimos a partir de los hechos recabados.
        - Cubo OLAP: cada dimensión del cubo es una dimensión del modelo. Según entiendo, no necesariamente está fija en 3 la cantidad de dimensiones.
        - Operaciones (sólo se entienden bien con ejemplos):
            - Slice - Dice: navegación por las dimensiones.
            - Drill down: enfocarse en algo de una dimensión.
            - Roll up: alejarse dentro de una dimensión.
        - Modelos básicos dimensionales
            - Star: una tabla de hechos (tabla fact) que contiene los datos para el análisis, rodeada de las tablas dimensionales.
            - Snowflake: algunas de las dimensiones se implementan con más de una tabla de datos, con la finalidad de normalizar las tablas.

Data Mining

- Es un área de la computación que estudia procesos para descubrir patrones en grandes volúmenes de datos. Utiliza métodos de IA, machine learning, estadística y BD.
- DM sólo se limita a encontrar patrones en un set de datos. No interviene en la recolección ni preparación de los datos, ni en la interpretación de los resultados.
- Aplicaciones
    - Asociación
        - Correlación y causalidad de hechos.
        - Por ejemplo, estudiar el nivel de estudios alcanzado de una persona según la categoría social de su familia.
    - Clasificación y predicción
        - Generar modelos o funciones que sean capaces de clasificar elementos en clases.
        - Árboles de decisión, redes neuronales.
        - Por ejemplo, predecir si una persona estará interesada en un crédito, según su edad y estatus financiero.

    - Cluster analysis
        - Agrupar elementos para formar clases (clusters).
        - Busca maximizar la similitud entre elementos de una misma clase, y minimizar la misma entre elementos de distintas clases.
        - Por ejemplo, agrupar ads del mismo rubro.
    - Análisis de outliers
        - Por ejemplo, para detectar fraudes o eventos raros.
    - Análisis de tendencias y evolución
- Técnicas
    - Se clasifican en supervisadas y no supervisadas.
        - Supervisadas: aquellas que requieren cierto tipo de entrenamiento o intervención humana, al principio.
        - No supervisadas: aquellas que no lo requieren.
    - Supervisadas
        - Redes neuronales
            - Sistemas capaces de aprender, adaptarse a condiciones variantes y hacer predicciones.
            - No son algorítmicas. No siguen una secuencia predefinida de instrucciones.
            - Aprenden por ejemplos, y de sus propios errores.
            - Red neuronal artificial (RNA): redes neuronales basadas en modelos simplificados de las neuronas reales.
            - Entrenamiento:
                - Pasarle todos los valores de entrenamiento. Para cada uno, hacemos lo siguiente:
                    - Calcular la diferencia de la salida con la esperada.
                    - Corregir la salida de modo tal que la diferencia se achique.
                    - No se busca que la diferencia tienda a cero rápidamente, sino que se achique de a poco.
                    - Si la diferencia se achica muy de a golpe, se corre el riesgo de que cada vez que se aprende algo nuevo, se modifique demasiado lo que se aprendió antes.
            - Tipos:
                - Perceptrón multicapa. Red formada por varias capas.
                - Red de Hopfield.
                - Red de Kohonen.

        - Las redes neuronales son buenas para:
            - Problemas que son muy difíciles de computar.
            - No requieren respuestas exactas, sino sólo rápidas y buenas.
        - Son malas para:
            - Cálculos exactos.
            - Procesamiento en serie.
            - Procesar datos en los que no haya ningún tipo de patrón.
    - Árboles de decisión
        - Idea: clasificar objetos, en base a una serie de preguntas sobre sus atributos.
        - Los nodos internos son preguntas sobre atributos.
        - Las hojas representan las clases resultantes.
        - Construcción:
            - Tenemos un set de objetos, ya clasificados. Estos son los datos de entrenamiento.
            - Comenzamos con todos los objetos en la raíz del árbol.
            - Se dividen recursivamente, en base a preguntas sobre ciertos atributos.
            - Eventualmente paramos la división y creamos las hojas.
        - Prunning:
            - Identificar y remover ramas que representan outliers o ruido.
        - De los árboles de decisión podemos extraer reglas de clasificación.
        - **Overfitting**
            - Sobreentrenar un algoritmo de aprendizaje, con ciertos datos para los que se conoce el resultado. Específicamente, cuando se lo entrena demasiado, o cuando se entrena con datos extraños, el algoritmo puede quedar ajustado a características muy específicas de esos datos, y no tener verdadera relación con la función objetivo.
            - En el caso de árboles de decisión, evitamos overfitting haciendo prunning.

            - Preprunning. Interrumpir la construcción del árbol en forma anticipada.
            - Postprunning. Quitar ramas de un árbol ya construído.
      - Orígenes de outliers
         - Variabilidad de la fuente. Aquellos que realmente ocurren en la población que muestreamos.
         - Errores del medio. Cuando no se dispone de una técnica que pueda medir adecuadamente el medio.
         - Errores del experimentador.
            - Error de planificación. La población sobre la que se toma la muestra no es elegida correctamente.
            - Error de realización. Medición mal hecha u otros errores humanos.
   - Regresión lineal
      - Técnica estadística para modelar e investigar la relación entre dos o más variables.
      - Variables independientes: aquellas que se toman para clasificar los casos. Por ejemplo, edad de la persona.
      - Variable dependiente: variables que podrían estar influídas por las variables independientes. Por ejemplo, cantidad de operaciones.
      - Tipos:
         - Simple: sólo se maneja una variable independiente.
         - Múltiple: maneja varias variables independientes.
- No supervisadas
   - Clustering
      - Objetivo: agrupar datos en clusters.
      - Hay que definir la noción de similitud entre elementos. Se hace a través de una función distancia.
      - A veces se asigna “peso” a las variables, según su importancia.
      - Formas de obtener un cluster:
         - Jerárquicas
            - Usa las distancias como criterio.
            - La idea es ir aglomerando los clusters, que inicialmente son los elementos aislados.
            - Métodos de enlace.

                ■ Método de Ward. Usa la suma de las distancias al cuadrado dentro de los clusters. Agrupar clusters con incremento mínimo en la suma total.
                ■ Método del centroide. La distancia entre dos clusters se define como la distancia entre los centroides.
            ○ No jerárquicas
                ■ Dado k, encontrar una partición de los elementos, en k clusters, que optimice cierto criterio de partición.
    ■ Reglas de asociación
        ● Idea: generar reglas del tipo IF condición THEN resultado, en forma automática.

No-SQL

- Ya se resumió este tema. Agregamos sólo una cosita.
- Una BD No-SQL distribuída está en las condiciones del teorema CAP. Dado que es imposible evitar errores de red, hay que estar preparados para esta situación.
- Ante una caída de la red, se debe seguir una estrategia, que consiste al menos de los siguientes 3 pasos:
    - Detectar la partición. En general, esto ocurre como consecuencia de cierto timeout.
        - Podemos continuar con la operación, perdiendo consistencia.
        - Podemos cancelar la operación, perdiendo disponibilidad.
    - Entrar en modo particionado. Determinar qué operaciones pueden realizarse y cuáles no.
    - Volver al modo normal. Cuando la operación se restaura, es necesario subsanar los problemas de consistencias. Se utilizan los logs de los nodos.

Administrador de datos

- Diferencia entre DBA y administrador de datos.
    - Un DBA es un especialista en un motor de BD. Sabe manejar sus DDL, DML y DCL.
    - Un administrador de datos es un especialista en los datos de una organización.
- Tareas de un administrador de datos

    - Diseño lógico
        - Analizar requerimientos.
        - Realizar modelado a partir de los requerimientos.
        - Definir estándares (nombres de entidades, relaciones, etc.) y asegurar su cumplimiento.
    - Diseño físico
        - Asistir al DBA en la creación de modelos físicos a partir de los modelos lógicos.
        - Analizar el volúmen de datos y requerimientos de espacio.
        - Ejecutar backups y recoveries.
        - Verificar integridad de los datos en las BD.
- Datos en las organizaciones
    - Se encuentran en dos lugares:
        - En los modelos de datos. Están explícitos.
        - En la cabeza de las personas. Están implícitos. Por ejemplo, cierta información sobre el modelado de la BD sólo la tiene el administrador de datos en su cabeza.
        - Por esto también es que los administradores de datos son tan valiosos para las organizaciones.

Long-duration transactions

- Motivación: poder ejecutar transacciones muy largas, sin afectar a la performance de la BD.
- Una multilevel (o nested) transaction T es un conjunto T = {t1, …, tn} de subtransacciones, y un orden parcial P sobre T.
- Supongamos que el orden es t1 < … < tn. Entonces, T ejecuta primero t1, luego t2, y así sucesivamente. En general, T debe ejecutar las transacciones respetando P.
- Si una ti falla, no hace falta abortar todo T, sino sólo ti. Podemos reiniciar ti o bien decidir no ejecutarla hasta más luego.
- Si una ti commitea, no commitea a la BD sino a T. En caso de que T aborte, debemos poder hacer rollback de cada tj.
- Cada subtransacción ti puede ser una transacción típica, u otra multilevel transaction.
- En definitiva, la idea es descomponer una transacción muy larga, en otras transacciones más pequeñas, obteniéndose una multilevel transaction.
- Una saga es una multilevel transaction de larga duración.