Gestión de datos distribuidos

Definiciones y conceptos

Base de datos distribuida

> 💡 Una **base de datos** es una abstracción de los datos para su explotación y aprovechamiento.

Entonces a una base de datos distribuida la podemos ver como una base de datos que no se limita a existir en un solo sistema o red. Puede ser solo 1 servidor o varios. El hecho de que esté integrado en un sistema distribuido (o en un contexto de computación distribuida) ya lo clasifica como base de datos distribuida.

Gestores de bases de datos

> 💡 Un **gestor de base de datos** es un sistema que se encarga de organizar, gestionar y recuperar eficientemente la información de una base de datos.

En la definición más formal, los SGBD tienen:

- Lenguaje de modelado de información (DDL : *Data Definition Language*).
- Estructuras (índices, árboles B^+) de almacenamiento optimizadas para una gran volumetría de datos.
    - Atención: La gran volumetría de datos no es lo que se entiende ahora por Big Data.
- Lenguaje de recuperación/manipulación de información (DML: *Data Manipulation Language*).
- Mecanismos para integrarse en un sistema con acceso con control transaccional.

En los SGBDR (*Sistema Gestor de Base de Datos Relacional*) el DDL y el DML vienen dados por SQL (*Structured Query Language*). Este lenguaje especifica un estándar, pero luego cada SGBDR extiende de forma particular a SQL para implementar nuevas funcionalidades. Cabe destacar que los SGDBR son los sistemas más extendidos actualmente en las aplicaciones de producción (sean distribuidas o no).

▼ ¿Por qué necesitamos bases de datos y a sus gestores?

Originalmente se utilizaban los ficheros para el almacenado de los datos de las aplicaciones. Esta aproximación presentaba los siguientes problemas:

- **Volumetría muy limitada**: Los ficheros crecen descontroladamente a medida que aumentan los datos.
- **Redundancia** (datos repetidos en múltiples ficheros) e **Inconsistencia**.
- **Acceso ineficiente a los datos**: En parte por el punto anterior y le debemos añadir que cada fichero puede tener una representación (estructura) de los datos distinta.
- **Acoplamiento a la estructura de los datos**: Al no estar aislados los datos de las aplicaciones, si varía la estructura de los datos todas las aplicaciones se deben de reajustar.
- **Carencia de integridad**: Las restricciones a los datos las deben de hacer las aplicaciones y es muy complicado añadir nuevas restricciones a posteriori.
- **Inexistencia de la atomicidad**: Es difícil garantizar que los datos no se quedan en un estado inconsistente mientras se realizan una serie de acciones.
- **Condiciones de carrera**: Se puede permitir el acceso simultáneo, pero es más complicado garantizar que no se producen actualizaciones conflictivas.
- **Seguridad**: No es fácil restringir el acceso a los datos.

Para mitigar todos esos problemas surgen los conceptos de BD relacionales y de los SGBD. Estos últimos manifiestan las propiedades ACID (Atomicidad, Consistencia, Individualidad o Aislamiento y Durabilidad) que solventan todos los problemas anteriores.

Bases de datos relacionales en entornos distribuidos

Las bases de datos relacionales consideran a la **atributos** (columnas) como la unidad de información que se relacionan entre sí para describir a una entidad descrita por una **tupla** (registro). Luego, dichas entidades se agrupan en las **relaciones** (informalmente tablas) si comparten una relación lógica.

> 💡 Una **base de datos relacional** es un conjunto de **relaciones**.

Para operar sobre ellas ya introdujimos anteriormente a SQL, que se puede dividir en el DDL y el DML. Aquí unos ejemplos de que se encargan cada uno en este subtipo de bases de datos:

- DDL
  - Defincición, modificación y borrado de esquemas de relación y sus restricciones de integridad.
  - Creación de índices.
  - Definiciones de vistas y órdenes de acceso.
- DML : Se basa en el álgebra relacional y el cálculo de tuplas.
  - Operaciones CRUD (Create Recover Update Delete) sobre las tuplas.

Información general de SQL

Orden de ejecución de consultas SQL

1. FROM
2. WHERE
3. GROUP BY
4. HAVING
5. SELECT

Algebra de conjuntos

- UNION [ALL]

Tipos de JOIN

- (INNER) JOIN: selecciona las tuplas que cumplen la condición de unión.
- LEFT (OUTER) JOIN: igual que el JOIN pero añade a las tuplas de la tabla izquierda que no cumplieron la condición (en la parte derecha estará a NULL)
- RIGHT (OUTER) JOIN: igual que LEFT JOIN pero ahora se incluyen

- INTERSECT [ALL]
- EXCEPT [ALL]

Si se especifica ALL se incluyen los duplicados.

las tuplas de la derecha.

- NATURAL JOIN: similar al JOIN pero asume que la condición es de igualdad entre atributos comunes entre las dos tablas.
- FULL (OUTER) JOIN: LEFT + RIGHT.

Subconsultas

- s {<, >, =, …} ALL (SELECT … FROM …) , cierto si cumple la condición para todos los valores.
- s {<, >, =, …} ANY (SELECT … FROM …), cierto si cumple la condición en cualquiera de los valores.
- s IN (SELECT … FROM …), cierto si S está en la subconsulta
- EXISTS (SELECT … FROM …), cierto si la subconsulta no devuelve un conjunto vacío. Se suele utilizar en consultas correlacionadas.

Acceso a los datos desde la aplicación

SQL interactivo y no interactivo

💡 **El SQL interactivo** es aquel que se ejecuta desde el cliente del SGBD y permite probar consultas, realizar prototipos y definir la estructura de la base de datos.

💡 **El SQL no interactivo** consiste en el acceso a la base de datos a través un middleware conocido como *driver de base de datos*. Estos se suelen desarrollar de forma específica para el lenguaje de la aplicación y el SGDBR. Estos definen una API común (ODBC, JDBC), para poder facilitar la transición de un SGDBR a otro.

Mecanismos de acceso a los datos

En orden de evolución, existen:

- SQL embebido
    - Las sentencias SQL se encuentran incrustadas en el código y se componen en base a strings dinámicamente que luego se pasan al SGDB a través del driver.
- Sentencias preparadas
    - Ahora la sentencia se encuentra precompilada y acepta parámetros mediante binding.
    - Mejora tanto el tiempo de respuesta (menor envío de información) como la seguridad.
- DataSources lógicos
    - Patrón de diseño que abstrae el acceso a los datos.
    - Mejora tanto la reusabilidad, mantenibilidad y los tiempos de acceso.

Como se puede ver, el acceso a los datos ha ido evolucionando hacia la separación de responsabilidades y eso nos lleva al **ORM** (*Object Relational Mapping*). Este constituye la abstracción del acceso en sí a los datos hasta el punto que es completemente transparente al programador gracias a frameworks de persistencia (ahora el código SQL en la aplicación no existe).

Frameworks ORM conocidos son:

- MyBatis (Java)
- Hibernate ORM (Java)
- SQLAlchemy (Python)
- Django (Python)

Procedimientos almacenados y triggers

Mientras que es verdad que se pueden acceder los datos a nivel de aplicación, las BBDD SQL han evolucionado para poder implementar funcionalidad dentro de la propia base de datos a través de funciones. Distinguimos dos tipos:

- **Funciones y procedimientos almacenados**: se ejecutan a petición del usuario.
- **Triggers**: Se ejecutan automáticamente al ocurrir un evento en una tabla (operaciones CRUD).

Ambos son tratados como cualquier otro objeto de la BBDD y se gestionan con CREATE, ALTER y DROP. Además, pueden estar escritos en SQL puro o PL/SQL

(Procedural language for SQL) e incluso en lenguajes como C, Python,TCL … tal y como soporte el SGDB. Su sintaxis se valida en tiempo de ejecución, no durante la creación.

Una diferencia fundamental entre ambos tipos es que **los procedimientos se invocan** (CALL) y a las **funciones** se les incluye en una **sentencia SQL** (SELECT * FROM funcion(a)).

Ventajas de los procedimientos almacenados

- Mejoran el rendimiento frente a la ejecución de SQL desde la aplicación.
- Su acceso está controlado por los mecanismos de seguridad.
- Aceptan parámetros de entrada.
- Reducen el uso de red y la latencia vs lanzar consultas SQL equivalentes desde la aplicación.

Ventajas de los triggers

- Es fácil implementar auditoría de los datos de forma automática e independiente de la aplicación.
- Sirven como mecanismo alternativo para validar la integridad de los datos.
- Pueden ser utilizados como planificadores de tareas dentro de la propia BBDD.

```
-- Este es el callback que ejecutá el trigger
CREATE OR REPLACE FUNCTION tr_function() RETURNS TRIGGER AS $$
BEGIN
  -- En [INSERT] OLD = NULL, EN [DELETE] NEW = NULL
  NEW.c3 = NEW.c1 + NEW.c3          -- NEW es la nueva tupla / OLD la antigua
  RETURN NEW;                       -- Tupla final insertada en la tabla
END;
$$ LANGUAGE 'plpgsql';

-- Esto es la creación del trigger
CREATE OR REPLACE TRIGGER tr_tabla  -- Ahora se podrá ver en el catálogo como tr_tabla
BEFORE INSERT ON tabla              -- BEFORE / AFTER [ACCIÓN] tabla
FOR EACH ROW EXECUTE
PROCEDURE tr_function();
```

Volumetría en el acceso de datos

Actualmente las aplicaciones del mundo real manejan grandes volúmenes de información e incluso pueden llegar de distintas fuentes y con formato heterogéneo

(es decir, formato distinto). Para solventar la heterogeneidad se suelen realizar ETL (*Extract Transform Load*) de las distintas fuentes. Pero un problema claro es el del rendimiento (en la mayoría de sistemas de los que se saca la información queda estipulado en el SLA un tiempo de respuesta de 3-5 segundos).

¿Cómo optimizamos las consultas?

Cada vez que se ejecuta una consulta, el SGDB crea el *explain plan* de la sentencia. Este plan indica la forma en la que el SGDB busca y/o inserta los datos (formas de cruzar las tablas, uso de índices, orden de ejecución de subconsultas, etc). Analizando este *explain plan* podemos identificar cuellos de botella y identificar alternativas más eficientes.

Para que dicho plan sea realmente efectivo se deben de mantener las estadísticas de las tablas actualizadas, especialmente tras grandes actualizaciones. En PostgreSQL se utiliza ANALYZE [tabla].

Tipos comunes de estrategias de acceso

- CONST: Directo o constante
  - Generalmente con tablas de una sola tupla o con un índice.
- EQ_REF: Cruce por clave única
- REF: Clave no única
- INDEX_MERGE
- UNIQUE_SUBQUERY
- INDEX_SUBQUERY
- RANGE: rango en índice
- FULL INDEX SCAN
- FULL TABLE SCAN
  - El acceso más ineficiente de todos.

Para realmente apreciar el uso del planificador/optimizador la BBDD debe contener un gran volumen de datos. Esto se debe a que para determinar que camino es el óptimo se utiliza un sumatorio de costes.

Factores generales que afectan al rendimiento

- Una única sentencia frente a varias (ejemplo de INSERT).
- La existencia de claves primarias.
- La existencia de índices.
  - En cierto modo la pkey es un índice implícito.

    - Podemos crear índices explícitos sobre atributos no primos (numéricos y texto) para acelerar las consultas.

Entornos de computación distribuida para grandes volúmenes de datos

Actualmente el volumen de datos disponibles era imposible de aprovechar en su totalidad con las técnicas anteriores. De ahí surge el concepto del “Big Data”, que es similar al “Small Data” (los volúmenes que se manejaban anteriormente) pero en mayor magnitud. Esta diferencia de tamaño nos lleva al desarrollo de nuevas técnicas, herramientas y arquitecturas para manejar el Big Data.

En parte el Big Data es posible gracias a la aparición de tecnologías de bajo coste que permite el almacenado y procesamiento de datos.

Originalmente el Big Data se definía por las tres “V”, pero actualmente se define por 5 “V”:

1. **Velocity** : Real time, Near Real Time, Periodic,…
2. **Volume** : PB, TB, GB, …
3. **Variety** : Heterogeneidad de los datos.
4. **Veracity** : Certeza de la veracidad de los datos.
5. **Value** : Valor real de los datos obtenidos.

Problemas del Big Data

La evolución de la volumetría de datos sigue aumentando (2011 → 1 ZB, actualmente → 3-4 ZB, 2040 → 40ZB) y nos estamos dando cuenta que las técnicas vigentes (desde ~1970) ya no son suficientes para algunas casuísticas. De ahí surgen dos necesidades relacionadas estrechamente:

- Procesamiento distribuido
- Almacenamiento distribuido

Ambas implican un crecimiento y escalado del sistema.

Tipos de escalabilidad y su utilidad frente al Big Data

- **Escalabilidad vertical** : Implica aumentar la potencia del nodo donde se ejecuta el software mediante mejoras del hardware.

    - Es simple de hacer si el software está preparado para manejar los nuevos recursos.
    - Es muy caro y más pronto que tarde nos encontraremos con limitaciones, ya sean por costes o incluso por el SO.
- **Escalabilidad horizontal** : Implica aumentar la cantidad de nodos donde se ejecuta el software para repartir la carga computacional (a través de un balanceador de carga).
    - El límite es mucho más alto
    - Cuanto más capacidad, más barata es frente a la escalabilidad vertical.
    - Normalmente al doblar el número de nodos doblamos la capacidad/rendimiento (predecible).
    - Sin embargo, requiere de software diseñado e implementado específicamente para ser ejecutado de forma distribuida.

Teorema del CAP

La escalabilidad horizontal implica la posibilidad de que al menos algún nodo falle eventualmente o la comunicación con él. En estas condiciones existen tres propiedades deseables:

- Consistencia: La información distribuida en los nodos es consistente
- Disponibilidad: Garantía de recibir una respuesta tras una petición a un nodo.
- Tolerancia al particionado: Capacidad de mantener el funcionamiento a pesar de que un nodo o conexión se ha caído.

El teorema del CAP establece que es imposible cubrir las tres propiedades simultáneamente, solo pudiendo llegar a dos como máximo.

- Sistemas CA: SGDBR
- Sistemas CP: BBDD NoSQL (MongoDB)
- Sistemas AP: Apache Cassandra

Tecnologías del Big Data: Apache Hadoop y Apache Spark

Es cierto que las BBDD se pueden distribuir, pero a coste de sus propiedades ACID, por lo que tenemos un dilema de qué hacer al caerse un servidor. Para solventarlo, surgieron nuevas técnicas y tecnologías para solventar este problema. Este es el ejemplo de Apache Hadoop.

Apache Hadoop

- Se basa en el modelo de programación MapReduce con una implementación de código abierto.
- Su implementación se basa en el HDFS de Google.
- Se encuentra incluido por defecto en Java.
- Actualmente fue superado por otras tecnologías.

HDFS - Hadoop Distributed File System

- Permite aprovechar y trabajar con la totalidad de la información, como si fuese un un solo nodo.
- Es resistente a fallos mediante replicación.

MapReduce

- El problema objetivo debe ser divisible en subtareas que puedan ser ejecutadas por separado.
    - **Fase Map** : el problema se divide en subproblemas con los Mappers (workers).
    - **Fase ejecución**: los subproblemas se resuelven de forma paralela.
    - **Fase Reduce**: los resultados se sintetizan de forma que dan solución al problema original con los Reducer (otros workers).

Luego cabe añadir que realmente Hadoop se basa en un gran ecosistema que le impulsó (Sqoop, Hive, Hue, …). Sin embargo, su problema principal es su mantenimiento, complejidad, rijidez y el hecho de que no todos los problemas son paralelizables.

De ahí surge su sucesor **Apache Spark**, que es entre 10 y 100 veces mejor e incluso tiene un ecosistema mejor integrado entre sí.

Bases de datos NoSQL

Son bases de datos sin esquemas, con interfaces distintas al SQL, flexibles, distribuidas y proporcionan soporte a gran volumetría de datos a través de la escalabilidad horizontal. Distinguimos varios tipos:

- Basadas en pares clave-valor: *Redis*
    - Es el más flexible e incluso se utiliza en el cliente.

- Basadas en grafos: *Neo4j*
    - Utilizan un grafo para establecer las conexiones entre datos y realizar consultas más eficientes.
- Basadas en columnas: *Cassandra*
    - Se estructuran en filas y columnas, pero las filas tienen nombres y formato variables.
- Basadas en documentos: *MongoDB*
    - Su unidad básica es el documento. Tienen una gran flexibilidad y capacidad de almacenamiento.