Grado en Ingeniería Informática

Aplicaciones de Bases de Datos

TEMA 1

Transacciones

Docentes:

Raúl Marticorena Sánchez
Jesús Maudes Raedo
Mario Martínez Abad

Índice de contenidos

I. Introducción

El concepto de ***transacción*** da título al tema y es la base fundamental para todo el curso. Las aplicaciones de bases de datos normalmente permiten actualizar y consultar la información en la base de datos. Estas operaciones, aunque hasta ahora han tenido lugar de una forma simple y transparente para nosotros utilizando DML de SQL, tienen por debajo una complejidad que tiene que ver con el mantenimiento de la **consistencia** de la información que al final registra la base de datos. También con cómo se gestiona el que se produzcan múltiples cambios, quizás incluso miles, de manera **concurrente.**

En este capítulo veremos que el concepto de transacción nos va a permitir definir y resolver estas complejidades. También veremos el alcance de hacer un buen o un mal uso de las transacciones, y explicaremos comportamientos aparentemente extraños de la base de datos, pero que son razonables cuando tenemos en cuenta que alguien puede estar cambiando la información a la vez que nosotros y/o cómo el SGBD está implementando por debajo la gestión de múltiples transacciones concurrentes.

II. Objetivos

OBJETIVO 1: Conocer el concepto de transacción, para qué sirve, qué propiedades ha de tener y cómo se gestiona desde una interfaz SQL.

OBJETIVO 2: Profundizar en la propiedad de aislamiento viendo detenidamente los dos niveles de aislamiento que se utilizan en la mayoría de SGBD. Incidir en el nivel de aislamiento serializable dándole un mínimo de soporte teórico.

OBJETIVO 3: Conocer los mecanismos de implementación de aislamiento para comprender mejor sus implicaciones en el rendimiento y en la respuesta del sistema (bloqueos y versiones).

OBJETIVO 4: Conocer cómo afectan los cambios concurrentes al mantenimiento de las restricciones en la definición de las tablas.

OBJETIVO 5: Intentar no centrar las explicaciones únicamente en un SGBD concreto. Aunque *Oracle* será el SGBD preferente en las explicaciones por ser el que se utiliza en el laboratorio, se intentará tanto omitir detalles innecesarios de implementación, como comparar ciertos comportamientos con otros dos SGBD de referencia (i.e., *PostgreSQL* y *SQL-Server*).

III. Contenidos específicos del tema

1. Transacciones: Autocommit, Commit y Rollback

Una transacción es un **conjunto de operaciones** sobre la base de datos que han de ejecutarse todas en conjunto. Es decir, **o se ejecutan todas o no se ejecutan ninguna**. Este principio de **todo o nada** evita estados inconsistentes de la base de datos.

Autocommit es una propiedad de una sesión[1] de la base de datos, que puede valer verdadero o falso. Si el *autocommit* está activado (verdadero) la base de datos ignora las transacciones que podamos definir, de manera que considera que cada operación SQL (cada, SELECT, cada INSERT, etc.) es en sí una transacción por separado.

Por tanto, para poder agrupar varias operaciones en una única transacción, lo primero que deberemos de hacer es establecer que la sesión tenga el *autocommit* a falso. Por cuestiones de eficiencia, normalmente, por defecto, las bases de datos tienen el *autocommit* a verdadero, por lo que deberás ser tú el que lo cambies (ver Tabla 1).

SGBD	Interfaz SQL	Tipo de Interfaz	Autocommit por defecto	Cambio comportamiento por defecto
Oracle	*SQL*Plus*	Línea de comandos	Desactivado	Se activa con el comando `set autocommit on`
	Sql Developer	Interfaz gráfica		
PostgreSQL	*pgAdmin*	Interfaz gráfica	Activado	Desactivar desde el menú de la aplicación. También es posible utilizar el comando *begin transaction*, que se comenta en la sección 1.1.
SQL-Server	*SQL Server Management Studio*			
	SQLCMD	Línea de Comandos	Activado	Se desactiva con el comando `SET IMPLICIT_TRANSACTIONS ON` También es posible utilizar el comando *begin transaction*, que se comenta en la sección 1.1.
PostgreSQL	*psql*		Activado	Se desactiva con el comando `\set AUTOCOMMIT off` También es posible utilizar el comando *begin transaction*, que se comenta en la sección 1.1.

Tabla 1: Autocommit en disitintos productos de bases de datos.

En la Tabla 1 podemos ver que las interfaces que se utilizan en el laboratorio (i.e.; *SQL*Plus y sql Developer* de *Oracle)* ya tienen desactivado por defecto el *autocommit*. Puedes comprobarlo tecleando `show autocommit`. Pero en el resto de productos de la tabla está activado por defecto.

Existen dos comandos SQL para la gestión de las transacciones: ***commit*** y ***rollback***. Cuando estamos en una

1 En muchos casos en bases de datos utilizamos el término *sesión* o *conexión* de manera indistinta. Hemos tratado de utilizar el término más apropiado, que es *sesión.* Por ejemplo, para *Oracle* la conexión es la parte física del sistema de comunicación con la base de datos, normalmente basada en un protocolo de red, entre un proceso en el cliente y el servidor de base de datos. La sesión es la estructura de datos lógica que mantiene la secuencia de operaciones SQL y la información relativa correspondiente. La misma conexión *Oracle* por tanto, puede contener varias sesiones lógicas *Oracle* ([Kyte 2010] capítulo 5); y normalmente cuando hablamos de *conexión* realmente lo que queremos decir es *sesión.*

transacción, desde la sesión actual -lógicamente- se pueden ver todas las modificaciones que estemos haciendo a la base de datos. Por ejemplo, con una SELECT podremos ver el resultado de todas las altas, bajas y modificaciones que hayamos hecho durante lo que va de la transacción actual. Sin embargo, "en principio" el resto de usuarios **no deberían poder ver** los cambios que estemos haciendo sobre la transacción hasta que finalice haciendo ***commit***. Si por el contrario, hago ***rollback*** la transacción también finaliza, pero se *"desharán"* o anularán los cambios correspondientes a la transacción.

En castellano, llamaremos ***cometer*** la transacción a hacer ***commit***, y ***retroceder*** la transacción a hacer ***rollback***.

Veamos un **ejemplo** con dos sesiones *SQL*Plus*, y comprobemos que los cambios de una sesión no siempre se ven desde la otra. Suponemos que está previamente creada la tabla:

```
CREATE TABLE mi_tabla (
      id        integer primary key,
      conn      char(6) )
```

la cual no tiene aún filas. La idea es que el campo `conn` (i.e.; número de conexión) registre un identificador de la sesión desde la cual se ha insertado cada fila. En la primera sesión cambiaremos el ***autocommit,*** para poder definir transacciones, activándolo. En la segunda sesión dejaremos el ***autocommit*** desactivado, que es su valor por defecto.

SESION 1 (autocommit = ON)

```
SQL> set autocommit on
SQL>
```

SESION 2 (autocommit = OFF)

```
SQL > show autocommit
autocommit OFF
SQL>
```

Ahora insertamos una fila desde cada una de las dos sesiones:

SESION 1 (autocommit = ON)

```
SQL> insert into mi_tabla values (1,'conn1');
1 row created.
Commit complete.
SQL>
```

SESION 2 (autocommit = OFF)

```
SQL> insert into mi_tabla values (2,'conn2');
1 row created
SQL>
```

Como la sesión 2 no tiene *autocommit,* la fila 2 podemos entender como que *"no se ha grabado"* aún en la base de datos, y por eso solo es visible desde la sesión 2. Así:

SESION 1 (autocommit = ON)

```
SQL> select * from mi_tabla;
        ID CONN
---------- ------
         1 conn1
SQL>
```

SESION 2 (autocommit = OFF)

```
SQL> select * from mi_tabla;
        ID CONN
---------- ------
         2 conn2
         1 conn1
SQL>
```

Ahora ambas sesiones hacen *rollback:*

SESION 1 (autocommit = ON)

```
SQL> rollback;
Rollback complete.
SQL>
```

SESION 2 (autocommit = OFF)

```
SQL> rollback;
Rollback complete.
SQL>
```

En el caso de la sesión 2, se perderán los cambios por tener desactivado el *autocommit*. En el caso de la sesión 1, el *rollback* no tiene efecto, es ignorado, por tener activado el *autocommit*, por lo que ambas sesiones solo muestran la fila 1.

SESION 1 (autocommit = ON)

```
SQL> select * from mi_tabla;
        ID CONN
---------- ------
         1 conn1
SQL>
```

SESION 2 (autocommit = OFF)

```
SQL> select * from mi_tabla;
        ID CONN
---------- ------
         1 conn1
SQL>
```

Ahora la sesión 2 inserta la fila 3 y hace *commit*, por lo que ambas sesiones percibirán el cambio:

SESION 2 (autocommit = OFF)

```
SQL> insert into mi_tabla values (3,'conn2');
1 row created.

SQL> commit;
Commit complete.

SQL> select * from mi_tabla;

        ID CONN
---------- ------
         3 conn2
         1 conn1
SQL>
```

SESION 1 (autocommit = ON)

```
SQL> select * from mi_tabla;

        ID CONN
---------- ------
         3 conn2
         1 conn1
SQL>
```

1.1. Comienzo y fin de una transacción al detalle

Para definir una transacción tenemos que saber cuándo empieza y cuándo acaba. En ese sentido:

1. Según el estándar SQL ([Melton y Simon 2002] sección 15.2) una transacción comienza implícitamente con la primera operación SQL que se ejecute en la misma. Típicamente comenzará con la primera sentencia DML de la misma (INSERT-DELETE-UPDATE-SELECT), pero cualquier otro comando SQL daría lugar al comienzo de la transacción (por ejemplo un comando DDL como CREATE TABLE, aunque los comandos DDL no están sujetos ni a *commit* ni a *rollback*). También, según el estándar, provoca el comienzo de la transacción, el comando estándar START TRANSACTION, que no está presente en Oracle (si bien PostgreSQL y SQL-Server tienen un comando similar a éste llamado BEGIN TRANSACTION). Dado que Oracle es el SGBD que utilizaremos en el laboratorio analizaremos sus peculiaridades a este respecto. **Otras cosas que son diferentes en Oracle** que en el estándar SQL son:
   1. El comando *SET TRANSACTION* (que utilizaremos en la sección 3) que sirve para configurar la transacción, provoca el comienzo de la misma ([Oracle 2015] capítulo 10).
   2. Los comandos DDL (e.g., ALTER/CREATE/DROP TABLE/VIEW etc ...) no dan lugar a comienzo de transacción, sino al final de la transacción en curso. Es más, Oracle hace un ***commit*** implícito antes de procesar un comando DDL (para más información ver la *Nota 1* del anexo 4.1).

   Lo que **no provoca** el comienzo de la transacción es:
   1. *Logearse*/conectarse, ni tampoco
   2. Que la transacción anterior de la misma sesión haya terminado con ***commit*** o ***rollback***.

   Esto significa que, en cualquier SGBD, a lo largo de una sesión SQL **existen tiempos muertos** que no pertenecen a ninguna transacción, y que, en el caso de Oracle, están comprendidos entre un *commit/rollback/log-in* y el primer comando SQL-DML/*SET TRANSACTION* que se ejecute detrás en esa misma sesión.
2. Según el estándar SQL ([Melton y Simon 2002] sección 16.2) al cerrar una sesión sin que haya finalizado la transacción en curso, el SGBD debería de devolver un error, pero esto no es así en la mayoría de los SGBD. Lo cierto es que distintos entornos SQL pueden comportarse de forma diferente
   - Por un lado en entornos gráficos es común que el sistema pregunte al usuario si hacer *commit* o *rollback* de una transacción en curso (e.g.; en *SQL-Server Management Studio* al cerrar una ventana de una transacción activa, también *SQL-Developer* cuando cerramos la aplicación).
   - En el caso de Oracle, en el cierre de la sesión provoca que el SGBD haga un un ***commit*** implícito de la transacción, por lo que se guardarían los cambios. Pero el propio *Oracle SQL*Plus* admite el comando `set exitcommit off` que permite alterar este comportamiento, haciendo que un cierre de la sesión provoque un *rollback* implícito.
   - *PostgreSQL* (tanto *pgAdmin* como *psql)* y *SQL-Server* hacen *rollback* implícito (este último cuando salimos con *\q* de SQLCMD).

   Por esta razón no dejes que el SGBD decida por ti para evitar sorpresas; NO abandones nunca la sesión sin antes haber cerrado la transacción previamente con *commit* o *rollback*.
3. Al producirse algún tipo de fallo del sistema que rompa la sesión (e.g. una caída de la red o de la base de datos) se hace un ***rollback* implícito**, por lo que también finaliza la transacción, pero descartándose los cambios.

La Tabla 2 resumen todos estos comportamientos para el caso de Oracle (para experimentar por ti mismo el contenido de la Tabla 2 ver la *Nota 2* del anexo 4.1).

Causa	**Efecto en Oracle**
`SET TRANSACTION ...` como siguiente orden a un `COMMIT` o `ROLLBACK`	Comienza la transacción con la configuración estipulada en el `SET TRANSACTION ...`
Comando DML `INSERT – DELETE -UPDATE – SELECT` como siguiente orden a un `COMMIT` o `ROLLBACK`	1. Comienza la transacción 2. Ejecuta el propio comando DML
Comando DDL `ALTER – CREATE - DROP`	1. Hace *commit* 2. Ejecuta el propio comando DDL (que no se puede retroceder) Nota que **no** comienza una nueva transacción tras la sentencia DDL
Salir de la sesión voluntariamente o `Exit`	1. Por defecto hace *commit* 2. Sale de la sesión
Logearse o iniciar la sesión	No comienza aún una transacción.
La sesión ha terminado involuntariamente	Es como si hiciera *rollback* justo antes de que termine la sesión.
`Commit/rollback`	Finaliza la transacción con `commit/rollback`. No comienza aún la siguiente transacción.

Tabla 2: Eventos y su consecuencia en el comienzo o fin de la transacción en Oracle.

Siguiendo con el ejemplo anterior. Insertamos una cuarta fila en la sesión 2 (que no tiene *autocommit*) y la cerramos sin hacer ni *commit* ni *rollback*.

SESION 2 (autocommit = OFF)

```
SQL> insert into mi_tabla values (4,'conn2');
1 row created.

SQL> exit
Disconnected from Oracle Database 11g Express
Edition Release 11.2.0.2.0 - Production

C:\Users\alumno>
```

Y comprobamos desde la sesión 1 que el cambio se ha grabado (*commit* implícito por salir **voluntariamente** de la sesión):

SESION 1 (autocommit = ON)

```
SQL> select * from mi_tabla;

        ID CONN
---------- ------
         3 conn2
         4 conn2
         1 conn1
SQL>
```

Ahora abrimos **otra sesión 3**, no cambiamos su *autocommit*, luego está desactivado, e insertamos una nueva fila:

SESION 3 (autocommit = OFF)

```
SQL> insert into mi_tabla values (5,'conn3');
1 row created.

SQL>
```

Para simular un cierre de la sesión por algún problema (por ejemplo por un fallo en la red, en el servidor, etc.) cerraremos la ventana de comandos de la sesión 3, por ejemplo clickando en el aspa de la esquina superior derecha de la ventana, y comprobaremos que en la sesión 1 no se ve la fila 5, porque la sesión 3 ha hecho *rollback* implícito por

abandono **involuntario** de la sesión:

SESION 1 (autocommit = ON)

```
SQL> select * from mi_tabla;

        ID CONN
---------- ------
         3 conn2
         4 conn2
         1 conn1
SQL>
```

2. ACID

Las bases de datos idealmente deben de soportar las características llamadas **ACID**, donde **A** viene de *Atomicity*/Atomicidad, **C** de *Consistency/Consistencia,* **I** de *Isolation*/Aislamiento y **D** de *Durability*/Durabilidad.

1. La **Atomicidad** viene dada porque las operaciones que se agrupan en una misma transacción son un todo indivisible, o se ejecutan todas o no se ejecuta ninguna. Por eso solo hay dos posibilidades en cómo acaba una transacción (*commit* o *rollback)* y si se cierra la sesión se toma una de estas opciones. Si el cierre de la sesión es por un accidente, hemos visto que se hace un *rollback* implícito, esto tiene que ver con la siguiente característica: la Consistencia.
2. Como su nombre indica la **Consistencia** busca que la base de datos no contenga estados inconsistentes. Por ejemplo, si hacemos una venta quizás haya varias operaciones SQL involucradas: un INSERT en una tabla de *pedidos* y quizás un UPDATE en la tabla de *existencias* decrementándolas. Si agrupamos las dos operaciones en la misma transacción, garantizamos que no haya un estado de la base de datos en el que se registre la inserción pero no la actualización y viceversa; porque al agruparlos en una transacción garantiza que se guarden los cambios de las dos operaciones, o ningún cambio. Por eso, cuando hay un accidente a mitad de una transacción que hace abortar la sesión, se opta por no guardar ninguna de las operaciones de esa transacción hechas hasta el momento (*rollback* implícito).
3. El **Aislamiento** es la propiedad por la que al acceder a la base de datos, ésta nos proporciona la sensación virtual de que solamente estamos accediendo nosotros a pesar de que pueda haber múltiples sesiones concurrentes. Para ello, una cuestión importante es que los cambios que transcurren en una transacción no puedan ser vistos desde otra, evitando que puedan leerse cambios que luego puedan ser retrocedidos. El aislamiento que proporciona la base de datos se puede configurar, pudiendo ser más estricto o menos.
4. Finalmente la **Durabilidad** tiene que ver con que los cambios se guarden de forma persistente en la base de datos. Esta característica parece obvia, pero no lo es tanto si tenemos en cuenta que las bases de datos, para ir más deprisa y ahorrar accesos a disco, guardan en memoria los bloques de disco más usados recientemente, de manera que los cambios que hacen las transacciones en principio solo se reflejan en la copia en memoria de cada bloque de disco, pero en el disco normalmente no se reflejan hasta más adelante. Por tanto, si se cae la base de datos, en principio los cambios en memoria podrían no estar aún persistidos en disco, quebrando esta propiedad de la *durabilidad* de la que estamos hablando.

   Afortunadamente este problema se resuelve mediante ficheros de *log* (ver anexo 2 Estrategias de recuperación mediante logs).

3. Soporte del Aislamiento en SQL

El aislamiento es una de las características ACID descritas en la sección anterior. Existen tres síntomas (fenómenos) claros de ausencia de aislamiento total:

1. ***Lectura sucia:*** es el peor síntoma, se manifiesta cuando una transacción T1 puede leer cambios que ha hecho otra transacción T2 que está en curso (i.e.; T2 todavía no ha hecho *commit* de esos cambios).
2. ***Lectura no repetible*****:** se manifiesta cuando una transacción al ejecutar dos veces la misma consulta lee valores distintos porque ha leído los cambios (UPDATEs) o borrados (DELETEs) producidos por otra(s) transacción(es) que han sido cometidas (confirmadas).
3. ***Lectura fantasma:*** se manifiesta cuando una transacción al ejecutar dos veces una misma operación DML que "*afecta a varias filas*"[2], obtiene **nuevas filas**, porque la segunda vez se han incluido nuevas filas **producto de inserciones** producidas por otras transacciones cometidas (confirmadas).

SQL distingue cuatro niveles de aislamiento según eviten, o no, los tres problemas anteriores. A saber:

Nivel de aislamiento SQL	Lectura sucia	Lectura no repetible	Lectura fantasma
READ UNCOMMITTED	Sí	Sí	Sí
READ COMMITTED	No	Sí	Sí
REPETEABLE READ	No	No	Sí
SERIALIZABLE	No	No	No

*Tabla 3: "Sí" significa que el problema de la columna **sí** se da (no se evita) con el nivel de aislamiento consignado en esa fila. Por ejemplo, la lectura sucia **Sí** se da en READ UNCOMMITTED.*

Buena parte de los SGBD actuales admiten **dos niveles de aislamiento**: ***READ COMMITTED*** (que suele ser el nivel de aislamiento por defecto) y ***SERIALIZABLE***. *Oracle* admite únicamente estos dos niveles, *PostgreSQL* amplía además de estos dos admite REPEATABLE READ. Por el contrario, *SQL-Server* admite los cuatro niveles de aislamiento que propone el estándar, si bien no es muy normal hacer uso del REPEATABLE READ, y no tiene demasiado sentido usar READ UNCOMMITTED.

La palabra *serializable* quiere decir que el sistema se comporta de forma equivalente a que todas las transacciones se ejecutasen ***en serie*** (i.e., ninguna transacción estuviese solapada en el tiempo con otra). Claramente una implementación *en serie* tal cual, por ejemplo mediante una cola de transacciones, es inadecuada para un sistema concurrente, pues provocaría bastantes esperas, y por ejemplo, una transacción sin cerrar bloquearía por completo el sistema. Aumentar el nivel de aislamiento no es gratuito, a mayor nivel de aislamiento también la base de datos consumirá mas recursos y podría ir más lenta, pues como veremos más adelante, el aislamiento se implementa a base de bloquear información, y/o a base de mantener una copia o versión de los datos antes de ser modificados.

Las transacciones desde SQL interactivo pueden configurarse con el comando SET TRANSACTION de manera que si lo usamos será el primer comando de la transacción actual. Por ejemplo podemos ejecutar los comandos:

- SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
- SET TRANSACTION ISOLATION LEVEL READ COMMITTED;

  Nota para el uso de Oracle en el laboratorio: *Oracle* solo permite ejecutar el comando SET TRANSACTION si es el primero de cada transacción, en caso de que no sea así devolverá "*ORA-01453:*

2 Por ejemplo una orden SELECT/DELETE/UPDATE con una condición tipo WHERE afecta a varias filas. Pero además, esta misma orden sin WHERE, también se pueden considerar que afectan a varias filas, pues en este último caso todas las filas de la tabla se ven afectadas si se trata de un UPDATE/DELETE, o todas las filas que se obtienen de un FROM se ven afectadas si se trata de una SELECT.

SET TRANSACTION must be the first statement of transaction". Por tanto, no se pueden poner tampoco 2 comandos SET TRANSACTION en la misma transacción, porque el segundo ya no sería el primer comando de la transacción. Sin embargo, en *Oracle* es posible hacer que toda la sesión tenga un determinado nivel de aislamiento con

```
ALTER SESSION SET ISOLATION_LEVEL=SERIALIZABLE
```

o con

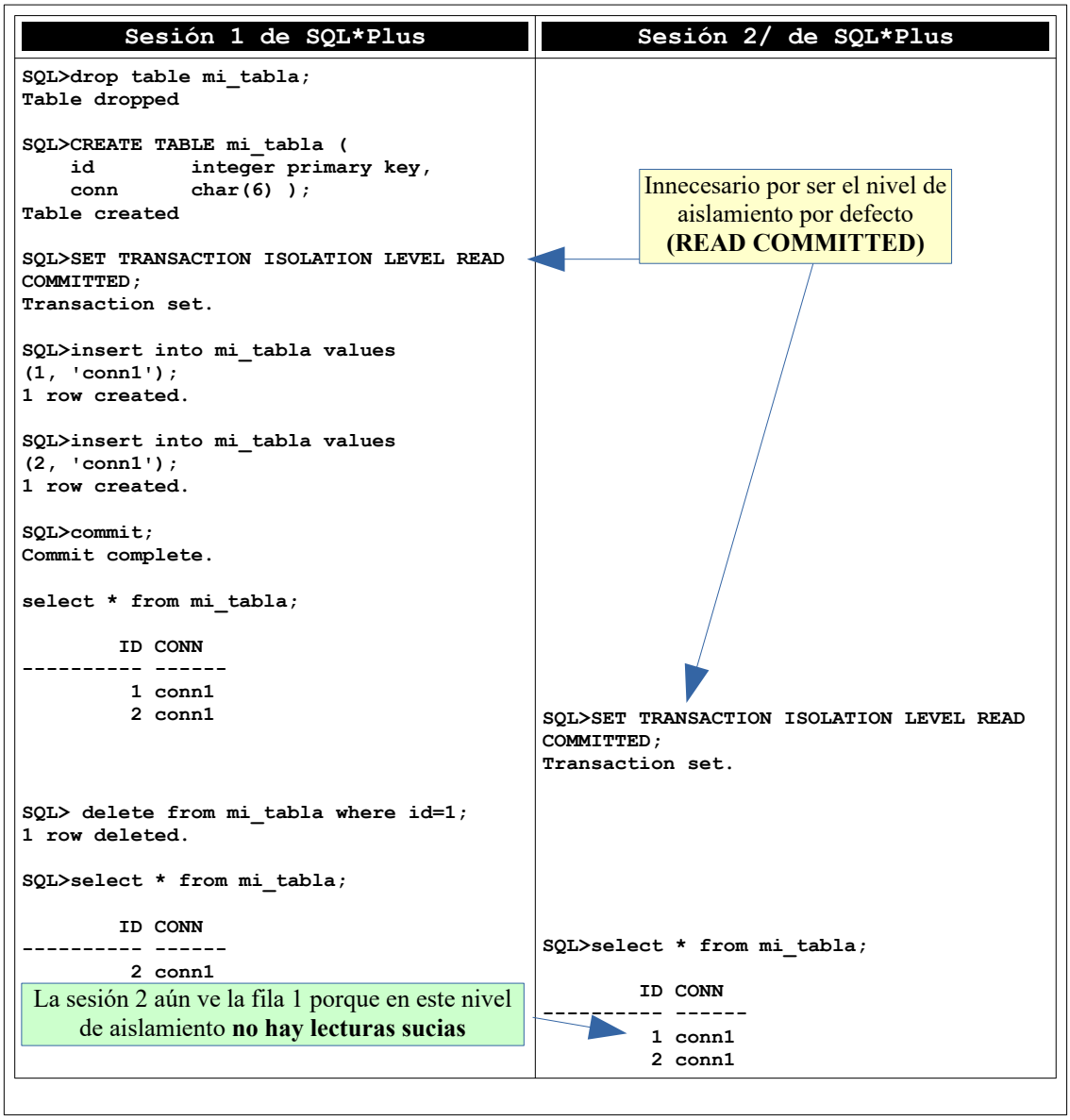
```
ALTER SESSION SET ISOLATION_LEVEL=READ COMMITTED
```

pero esta sintaxis no es estándar[3].

3.1. Ejemplo diferencia entre READ COMMITTED y SERIALIZABLE

La traza que se presenta muestra que la diferencia entre estos dos niveles de aislamiento es que el READ_COMMITTED no evita ni la lectura no repetible, ni el fenómeno fantasma.

3 En PostgreSQL existe un comando equivalente: *SET SESSION CHARACTERISTICS AS TRANSACTION.* En SQL-Server no hace falta un comando similar, pues SET TRANSACTION determina el nivel de aislamiento de las transacciones subsiguientes hasta que se vuelva a cambiar con otro SET TRANSACTION.


Sesión 1 de SQL*Plus
Sesión 2/ de SQL*Plus
SQL>drop table mi_tabla;
Table dropped
SQL>CREATE TABLE mi_tabla (
id integer primary key,
conn char(6));
Table created
SQL>SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
Transaction set.
SQL>insert into mi_tabla values (1, 'conn1');
1 row created.
SQL>insert into mi_tabla values (2, 'conn1');
1 row created.
SQL>commit;
Commit complete.
select * from mi_tabla;
ID CONN
1 conn1
2 conn1
SQL> delete from mi_tabla where id=1;
1 row deleted.
SQL>select * from mi_tabla;
ID CONN
2 conn1
La sesión 2 aún ve la fila 1 porque en este nivel de aislamiento no hay lecturas sucias
Innecesario por ser el nivel de aislamiento por defecto (READ COMMITTED)
SQL>SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
Transaction set.
SQL>select * from mi_tabla;
ID CONN
1 conn1
2 conn1

```
SQL>commit;
Commit complete.
```

La sesión 2 **repite la misma consulta** en la misma transacción y obtiene **resultados diferentes** a causa de **cambios o borrados** en las transacciones cometidas
=> ***Lectura no repetible***

```
SQL>SET TRANSACTION ISOLATION LEVEL READ
COMMITTED;
Transaction set.

SQL>insert into mi_tabla values
(3, 'conn1');
1 row created.

SQL>select * from mi_tabla;

        ID CONN
---------- ------
         2 conn1
         3 conn1
```

```
SQL>commit;
Commit complete.
```

La sesión 2 **repite la misma consulta** en la misma transacción y obtiene **resultados diferentes** a causa de **inserciones** en las transacciones cometidas
=> ***Fenómeno Fantasma***

Cambio al nivel de aislamiento **máximo (SERIALIZABLE)**. Durante una transacción la BD debe **parecer congelada** en el tiempo respecto de los cambios en otras sesiones

```
SQL> delete from mi_tabla where id=2;
1 row deleted.

SQL> select * from mi_tabla;

        ID CONN
---------- ------
         3 conn1
```

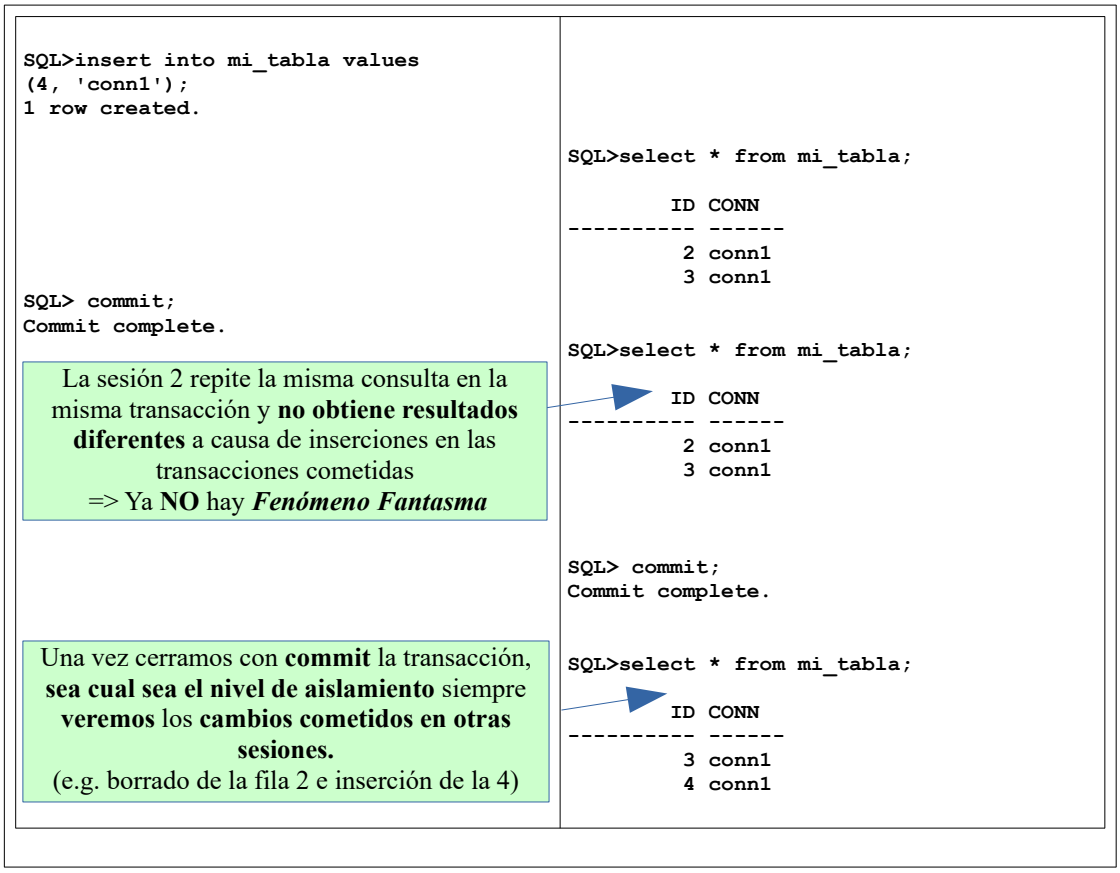
```
SQL> commit;
Commit complete.
```

La sesión 2 repite la misma consulta en la misma transacción y obtiene **resultados idénticos** aunque hay cambios o borrados en las transacciones cometidas
=> Ya **NO** hay ***Lectura no repetible***

```
SQL>select * from mi_tabla;

        ID CONN
---------- ------
         2 conn1
```

Innecesario por ser el nivel de aislamiento por defecto

```
SQL>select * from mi_tabla;

        ID CONN
---------- ------
         2 conn1

SQL>select * from mi_tabla;

        ID CONN
---------- ------
         2 conn1
         3 conn1

SQL>commit;
Commit complete.

SQL> set transaction isolation level
serializable;
Transaction set.

SQL>select * from mi_tabla;

        ID CONN
---------- ------
         2 conn1
         3 conn1

SQL>select * from mi_tabla;

        ID CONN
---------- ------
         2 conn1
         3 conn1
```



4. El concepto de serializabilidad

4.1. Operaciones primitivas

Para poder definir y discutir el concepto de serializabilidad necesitamos previamente identificar qué operaciones se realizan en una transacción por un SGBD, sea del tipo que sea (en nuestro caso un SGBD relacional). En una transacción las operaciones primitivas podemos asumir que son ([García-Molina et al. 2009] sección 17.1.4):
Sea un dato X, entenderemos cada dato X como la unidad mínima de información que es capaz de procesar el SGBD (e.g.; una pagina de disco en SGBDs muy primitivos, o una fila de una tabla en SGBDs actuales).

- READ (X, t): El dato X es leído en una variable local de la transacción que llamamos t. En el caso de SQL representarían las SELECTs
- WRITE (X, t): El contenido de la variable t es escrito en el dato X. En el caso de SQL representarían UPDATEs, INSERTs y DELETEs. Podemos asumir que DELETE consiste en escribir una marca especial en la fila que significa que la fila ha sido borrada. INSERT consiste en reservar un espacio del bloque o página[4] del SGBD para la nueva fila y escribir sus valores iniciales. Luego WRITE es una primitiva que nos sirve no solo para representar los UPDATEs de SQL, sino también DELETEs e INSERTs.

4 De ahora en adelante utilizaremos los términos *bloque* y *página* de manera indistinta para designar igualmente a la unidad mínima de datos, usualmente ubicados de forma contigua en disco, sobre los que el SGBD es capaz de hacer una operación de E/S (i.e.; cuando el SGBD hace una lectura o una escritura, lee o escribe una página entera).

- A mayores, en una transacción pueden ocurrir operaciones de cómputo sobre esa variable t; como por ejemplo operaciones aritméticas, bifurcaciones, bucles etc...

El dato X en un SGBD moderno representa normalmente una fila.

> **Nota 1**: Efectivamente, en un SGBD moderno X representa normalmente una fila, pero podría ser un bloque de disco etc ... depende de la granularidad con la que trabaje la base de datos. En bases de datos antiguas se trabajaba con granularidad de bloque o página de disco.
>
> **Nota 2**: Los bloques de la base de datos pueden corresponderse, o no, con los de disco del sistema operativo. De hecho, es normal que un bloque o página de un SGBD contenga varias páginas o bloques del sistema operativo. Por ejemplo, en *Oracle* actualmente las páginas del SGBD pueden configurarse diversos tamaños: 2Kb, 4Kb, 8Kb y 16Kb, el tamaño por defecto de Oracle son 8Kb.

Las operaciones READ y WRITE no se hacen directamente sobre disco, sino sobre un área de memoria que se suele llamar *buffer* del SGBD. Ese área está organizado en bloques o páginas del SGBD, de manera que un bloque del SGBD en el *buffer* se corresponde siempre con un bloque del SGBD en el disco. Lo contrario, en general, no es cierto; pues normalmente el tamaño del *buffer* es inferior al del disco y por lo tanto no todas las páginas del disco caben en el *buffer*[5].

El SGBD tiene un módulo llamado gestor del *buffer* que se encarga de llevar a memoria las páginas en las que está ese dato X que se necesita leer o escribir en cada momento:

- Si esa página con el dato X ya estaba en memoria (denotémosla por P1), el gestor del *buffer* no tiene que hacer nada, pero …
- Pero si P1 aún no está en memoria, el gestor del *buffer* ha de subirla de disco a memoria. En este último caso podría darse la circunstancia de que el *buffer* estuviera lleno, por lo que tendrá previamente que desalojar alguna página P2 del *buffer* (quizás la que lleve más tiempo sin usarse) para así poder traerse P1. Además, si en P2 se hubieran hecho operaciones WRITE, su contenido podría ser distinto que en el disco (i.e., en ese caso sería un *bloque sucio*), por lo que adicionalmente se debería de escribir en algún momento la versión de P2 en memoria al disco.

(Para más información sobre la operativa del gestor del *buffer* ver el anexo 2).

Además del gestor del *buffer*, el SGBD tiene otro módulo llamado *scheduler* o **planificador de transacciones**, que es el encargado de que las transacciones mantengan un determinado nivel de aislamiento. Esta tarea se conoce como **control de concurrencia**. Por tanto, el planificador de transacciones decide el orden de las operaciones WRITE/READ de las transacciones concurrentes que haya en cada momento, estableciendo esperas en la ejecución de esas operaciones si fuese necesario.

4.2. Planificación en serie

Una **planificación** de un conjunto de transacciones {T1 ... TK} es una secuencia ordenada del conjunto global de todas las operaciones que realizan esas transacciones. Las planificaciones han de respetar el orden interno de las operaciones de cada transacción Ti, pero las operaciones de una transacción Ti pueden intercalarse con las de cualquier otra transacción de esa planificación. El planificador de transacciones visto en el punto anterior, sería, por tanto, el encargado de definir las planificaciones pertinentes en cada momento.

Veamos un ejemplo: Tenemos un cliente de un banco con 2 cuentas en el mismo. La cuenta CC, representa su cuenta corriente, mientras la cuenta CP, representa un préstamo que tiene que amortizar en 10 pagos. Por simplificar, no hay intereses, y en cada pago se abona la décima parte de la deuda. Se definen las transacciones T1 y T2:

5 En el anexo 2.5 se introducen los SGBD *in-memory* en los que, por el contrario, sí que ocurre que toda la información persistente, también se encuentra en memoria.

- La transacción T1, representa una amortización anticipada de parte de la deuda. El cliente transfiere de manera voluntaria/eventual una cantidad fija desde su cuenta corriente a la cuenta del préstamo para que en el siguiente plazo se reduzca la cuota mensual. En el ejemplo, supondremos que el cliente hace en T1 una amortización de 100 euros.
- La transacción T2, representa el pago mensual del préstamo. Se retirará un 10% de lo que queda por pagar.

Supongamos que el cliente tiene una cuenta corriente CC con 1000 euros (CC inicialmente vale 1000), y que tiene un préstamo en CP de 500 euros (CP inicialmente vale -500, nota el signo negativo). Podríamos sintetizar el mantenimiento de la consistencia ACID en este ejemplo, con el mantenimiento de la siguiente restricción: *la suma de los saldos de CC y CP ha de ser la misma, antes y después de las transacciones T1 y T2.* En este caso partimos de un estado en el que la suma de los dos saldos es 1000 – 500 = 500.

> **Nota:** La consistencia tiene que ver con mantener de forma coherente varias informaciones relacionadas, y podríamos haber encontrados ejemplos más realistas, con su correspondiente restricción que exprese en qué consiste la consistencia en cada caso, pero serían ejemplos más complejos para las discusiones que vendrán a partir de ahora y por eso nos hemos quedado con un ejemplo menos realista pero más simple. Otros ejemplos de restricciones que representen un estado consistente a mantener serían: "el total de la factura ha de ser igual a la suma de los subtotales de las lineas de factura"; o "el *stock* del almacén sumado a la cantidad vendida es igual a la cantidad comprada" etc …

Las dos columnas de la izquierda de la Ilustración 1 mostrarían en qué consiste la transacción T1 utilizando las primitivas antes definidas, y las 2 columnas de la derecha mostrarían en qué consiste la transacción T2.

Operación	T1	Operación	T2
T1.1	READ(CP, cp)	T2.1	READ(CP, cp)
T1.2	cp := cp+100 /*-500+100*/	T2.2	cp_new := cp+abs(cp)*10% /*-500+500*0.1*/
T1.3	WRITE(CP, cp) //400	T2.3	WRITE(CP, cp_new) //450
T1.4	READ(CC, cc)	T2.4	READ(CC, cc)
T1.5	cc := cc-100 /*1000-100*/	T2.5	cc := cc-abs(cp)*10% /*1000-500*0.1*/
T1.6	WRITE(CC, cc) //900	T2.6	WRITE(CC, cc) //950

Ilustración 1: Ejemplo de dos transacciones T1 y T2 que podrían ejecutarse concurrentemente.

1. La secuencia de operaciones {**T1.3**,T1.4, T2.1, T2.2, T1.1, **T1.2**, T2.3, T2.4, T2.5, T1.5, T1.6, T2.6} no es una planificación de {T1, T2} porque no se preserva el orden de las operaciones dentro de alguna de las transacciones T1 y T2. Por ejemplo T1.3 transcurre antes que T1.2.
2. La secuencia de operaciones {**T1.1**,**T1.2**, T2.1, T2.2, **T1.3, T1.4**, T2.3, T2.4, T2.5, **T1.5, T1.6**, T2.6} (se ha remarcado en negrita las operaciones de T1 para que resalten sobre las de T2) sí es una planificación de {T1, T2} porque sí se preserva el orden de las operaciones dentro de ambas transacciones T1, T2.

Una **planificación** de las transacciones {T1 ... TK} se dice que es ***en serie***, si cada transacción Ti se ejecuta por completo sin permitir que otra transacción Tj haga ninguna operación de forma concurrente mientras se están ejecutando operaciones de Ti. Es decir, una planificación en serie consiste simplemente en encolar las transacciones una detrás de otra e ir ejecutando las operaciones de cada transacción, de manera que el resto de transacciones permanecen en cola hasta que las llega su turno.

En el ejemplo anterior hay 2 posibles planificaciones en serie que salen de ejecutar primero T1 y luego T2, o de hacer esa ejecución en orden inverso:

1. {**T1.1,T1.2, T1.3, T1.4, T1.5, T1.6**, T2.1, T2.2 ,T2.3, T2.4, T2.5, T2.6} que representamos como (T1, T2)
2. {T2.1,T2.2, T2.3, T2.4, T2.5, T2.6, **T1.1, T1.2 ,T1.3, T1.4, T1.5, T1.6**} que representamos como (T2, T1)

La restricción que ha de mantenerse mediante la consistencia nos exige que la suma de los saldos de las dos cuentas siga siendo 500. T1 ejecutada en solitario mantiene esa propiedad, y T2 también. Por tanto, la planificación en serie garantiza que la consistencia no se rompa, porque cada transacción por separado se supone que está programada convenientemente para mantener la consistencia. Luego cada vez que se ejecuta en solitario una transacción, la base de datos pasa de un estado consistente a otro; y si se ejecutasen varias transacciones secuencialmente, la base de datos irá saltando de transacción en transacción de un estado consistente a otro, para acabar finalmente en un estado consistente también.

4.3. Planificaciones serializables

La planificación en serie provoca que cada transacción T acabe esperando en una cola a que las transacciones anteriores a ella acaben, para que así T pueda disponer en solitario de toda la base de datos; lo cual es inaceptable en un entorno concurrente, y además, el disponer de toda la base de datos, es innecesario; pues en general la transacción solo escribe y lee en una pequeña parte de la base de datos.

Decimos que una **planificación** P es **serializable** si existe una planificación en serie S tal que para cualquier estado inicial de la base de datos, si se ejecuta P sobre ese estado, se llega al mismo estado final de la base de datos que si se ejecutara S sobre ese mismo estado inicial.

Por lo tanto, se trata de una planificación que es equivalente a una de las posibles planificaciones en serie. Dado que cualquier planificación en serie mantiene la consistencia, una planificación equivalente también. Por ello, las planificaciones serializables tienen 2 ventajas:

1. Mantienen la consistencia
2. No exigen necesariamente que las transacciones se ejecuten una detrás de otra, pudiéndose intercalar operaciones de varias transacciones concurrentes, evitando esperas innecesarias.

Sobre el ejemplo anterior, recordemos que el dato CP vale -500, y CC vale 1000; y que la propiedad de consistencia en el ejemplo nos exige que la suma de ambos valiese 500.

Operación	T1	Operación	T2	CP	CC
				-500	1000
T1.1	READ(CP, cp)				
T1.2	cp := cp+100				
T1.3	WRITE(CP, cp)			-400	
		T2.1	READ(CP, cp)		
		T2.2	cp_new := cp+abs(cp)*10%		
		T2.3	WRITE(CP, cp_new)	-360	
T1.4	READ(CC, cc)				
T1.5	cc := cc-100				
T1.6	WRITE(CC, cc)				900
		T2.4	READ(CC, cc)		
		T2.5	cc := cc-abs(cp)*10%		
		T2.6	WRITE(CC, cc)		860

Ilustración 2: Efecto de una planificación serializable sobre el ejemplo de la Ilustración 1.

Se observa que la planificación de la ilustración:

P={ **T1.1, T1.2, T1.3**, T2.1, T2.2, T2.3, **T1.4, T1.5, T1.6**, T2.4, T2.5, T2.6}

es equivalente a la planificación en serie S12=(T1, T2), ya que ambas planificaciones terminan dejando a la base de datos en el mismo estado (en el ejemplo parten de un estado en que CC+CP=500 y acaban en un estado similar); si bien no es equivalente a la planificación en serie S21=(T2, T1), pues S21 primero aplica T2 amortizando el 10% de 500 (50 euros), y luego transferiría los 100 euros de T1; de donde CP acabaría con -350 y CC con 850.
Al ser P equivalente a una de las posibles planificaciones en serie, podemos decir que es una **planificación serializable**.

> **Importante:** Nota como desde este punto de vista teórico de la serializabilidad, el estado final de la base de datos no es relevante con tal de que se mantenga la consistencia. Es decir, no importa el orden de las transacciones en la planificación en serie a la que P es equivalente (en el ejemplo, la planificación P equivalente a (T1,T2) no deja la base de datos en el mismo estado que otra planificación P' equivalente a (T2,T1), y sin embargo tanto P como P' serían serializables).

Sin embargo en la planificación

Q={ **T1.1, T1.2, T1.3**, T2.1, T2.2, T2.3, **T1.4, T1.5,** T2.4, T2.5, T2.6, **T1.6**}

1. Los 3 primeros pasos de T1 resultarían en CP = -500 + 100 = -400
2. Los 3 primeros pasos de T2 resultarían en CP = -400 + 40 = -360
   (hasta ahora todo igual que la Ilustración 2)
3. Al ejecutarse las 2 siguientes operaciones de T1 y las 3 siguientes operaciones de T2 CC = 1000 – 40 = 960
4. Al ejecutarse la última operació de T1 CC = 1000 – 100 = 900

Claramente -360 + 900 = 540 ≠ 500, no manteniéndose la consistencia, por lo que Q no es equivalente a ninguna de las 2 planificaciones en serie (T1, T2) ni (T2, T1).

4.3.1 Solo importa READ y WRITE

Sin embargo, con la misma planificación Q que resulta no ser serializable en el ejemplo anterior, si que resultaría serlo si ese año el interés baja al 0% y sustituimos las operaciones “cp*10%” por “cp*0%” en T2 (i.e., el banco mensualmente carga al mes cero euros, debas lo que debas):

1. Los 3 primeros pasos de T1 resultarían en CP = -500 + 100 = -400
2. Los 3 primeros paso de T2 resultarían en CP = -400 + 0 = -400
3. Al ejecutarse las 2 siguiente operaciones de T1 y las 3 siguientes operaciones de T2
   CC = 1000 – 0 = 1000
4. Al ejecutarse la última operación de T1 CC = 1000 – 100 = 900

donde -400 + 900 = 500. Como T2 no tiene efecto tanto la planificación en serie (T1,T2), como la planificación en serie (T2,T1), como la planificación Q son equivalentes (Q es equivalente a las 2 posibles planificaciones en serie en este caso).
Sin embargo, el *scheduler o planificador de transacciones* del SGBD no puede entrar a analizar cuáles son las operaciones concretas que realiza cada transacción ([García-Molina et al. 2009] sección 18.1.14); el caso del ejemplo es sencillo, pero la casuística que nos podemos encontrar si consideramos todo tipo de operaciones algebraicas, de manejo de cadenas, bifurcaciones, bucles, llamadas a otros subprogramas etc ... es inmensamente compleja; por lo que claramente, lo único que el *scheduler* puede tener en cuenta para intentar trabajar de una manera razonablemente abordable son las operaciones READ y WRITE.
A este fin, el *scheduler* actúa asumiendo el peor escenario posible, en una actitud conservadora/pesimista:

1. Que todo dato X que escribe una transacción T que ha hecho alguna lectura sobre el estado inicial de la base de datos, normalmente habrá sido cambiado por T en función de ese estado inicial que ha leído, y
2. Que esos cambios si eventualmente suceden concurrentemente a otros cambios que pueda hacer otra

transacción T' sobre ese mismo dato X, no tienen por qué llegar a un estado consistente de la base de datos.

4.3.2 Notación para transacciones y planificaciones

Como el *scheduler* no puede analizar lo que se hace con las variables locales "t" las eliminaremos de la notación, y así en la transacción adoptaremos las siguientes abreviaturas tomadas de [García-Molina et al. 2009], sección 18.1.5:

- WRITE(X, t) en la transacción T lo representaremos como $w_T(X)$
- READ(X, t) en la transacción T lo representaremos como $r_T(X)$

Por lo tanto, las transacciones T1 y T2 de los ejemplos se pueden representar como

- T1 : $r_{T1}(CP)$; $w_{T1}(CP)$; $r_{T1}(CC)$; $w_{T1}(CC)$;
- T2 : $r_{T2}(CP)$; $w_{T2}(CP)$; $r_{T2}(CC)$; $w_{T2}(CC)$;

4.3.3 Serializabilidad en cuanto a conflictos

La serializabilidad en cuanto a conflictos es un tipo de serializabilidad más exigente que la serializabilidad definida hasta ahora en la sección 4.3. O dicho de otra manera, el conjunto de las planificaciones *serializables en cuanto a conflictos* de un conjunto de transacciones concurrentes T1 … TK es un subconjunto del conjunto de todas las posibles planificaciones *serializables* de esas transacciones concurrentes. La *serializabilidad en cuanto a conflictos* es la que usualmente se limitan a implementar los sistemas comerciales. Existen otras formas de serializabilidad menos exigentes, pero más complejas de examinar de manera eficiente, como por ejemplo la serializabilidad *en cuanto a vistas* ([Silberschatz et al. 2015] sección 15.4.3, [Connolly & Begg. 2005] sección 20.2.2).

Se dice que existe un ***conflicto*** entre dos transacciones Ti y Tj ($i \neq j$) si Ti contiene al menos una operación de lectura o escritura que si se realiza antes que otra operación de Tj el resultado final de al menos una de las dos transacciones puede ser distinto ([García-Molina et al. 2009], sección 18.2). Para que exista un conflicto han de darse una de estas dos situaciones:

1. Las dos intentan escribir sobre el mismo dato, o bien
2. Una hace una lectura de un dato que ha sido escrito por la otra.

O dicho en una sola frase: ***hay conflicto si existiendo un dato común sobre el que leen o escriben al menos 2 transacciones concurrentes, y al menos una de ellas escribe ese dato*** ([García-Molina et al. 2009], sección 18.2.1).

Para demostrar que una planificación P es serializable en cuanto a conflictos tendremos que convertirla en una planificación en serie haciendo intercambios entre operaciones consecutivas de la planificación, pero con las siguientes restricciones:

1. Naturalmente, no se puede intercambiar el orden de dos operaciones de la misma transacción. El orden interno de las operaciones dentro de una transacción no se puede alterar.
2. No podemos intercambiar $w_{T1}(X)$ con $w_{T2}(X)$; porque están en conflicto.
3. No podemos intercambiar $r_{T1}(X)$ con $w_{T2}(X)$ (o viceversa: $w_{T1}(X)$ con $r_{T2}(X)$); porque están en conflicto.

Si somos capaces de reordenar las operaciones concurrentes de un conjunto de transacciones haciendo intercambios sujetos a esas tres restricciones y obtenemos así una planificación en serie, entonces la planificación P, no es solo serializable, sino que además es serializable en cuanto a conflictos.

Por ejemplo, la planificación:

$$P = r_{T1}(CP); w_{T1}(CP); r_{T2}(CP); w_{T2}(CP); r_{T1}(CC); w_{T1}(CC); r_{T2}(CC); w_{T2}(CC);$$

Puede someterse a los siguientes intercambios entre operaciones consecutivas de P que no están en conflicto (en negrita las operaciones que se van a intercambiar, en subrayadas las que se han intercambiado):

$$P = r_{T1}(CP); w_{T1}(CP); r_{T2}(CP); \mathbf{w_{T2}(CP); r_{T1}(CC);} w_{T1}(CC); r_{T2}(CC); w_{T2}(CC);$$

$$r_{T1}(CP); w_{T1}(CP); \mathbf{r_{T2}(CP); \underline{r_{T1}(CC);}} \underline{w_{T2}(CP);} w_{T1}(CC); r_{T2}(CC); w_{T2}(CC);$$

$$r_{T1}(CP); w_{T1}(CP); \underline{r_{T1}(CC); r_{T2}(CP);} \mathbf{w_{T2}(CP); w_{T1}(CC);} r_{T2}(CC); w_{T2}(CC);$$

$r_{T1}(CP)$; $w_{T1}(CP)$; $r_{T1}(CC)$; **$r_{T2}(CP)$; $w_{T1}(CC)$;** $w_{T2}(CP)$; $r_{T2}(CC)$; $w_{T2}(CC)$;

$r_{T1}(CP)$; $w_{T1}(CP)$; $r_{T1}(CC)$; $w_{T1}(CC)$; $r_{T2}(CP)$; $w_{T2}(CP)$; $r_{T2}(CC)$; $w_{T2}(CC)$;

Al llegar finalmente a una planificación en serie (T1, T2), podemos decir que P es serializable en cuanto a conflictos porque no hemos hecho ninguna reordenación que cambie el orden de dos operaciones en conflicto.

4.3.3.a Detección de la *serializabilidad en cuanto a conflictos* mediante grafos

Para el caso de que haya más de dos transacciones concurrentes en la misma planificación, se utilizan técnicas basadas en grafos para analizar si las planificación es serializable en cuanto a conflictos. Se crea un **grafo de precedencia** que expresa el orden entre las transacciones en una supuesta planificación en serie equivalente. El grafo tiene por nodos las transacciones. **Se dibujan arcos que van de una transacción Ti a una Tj si la Ti se debe de empezar a ejecutar antes que Tj a causa de un conflicto entre ambas** ([Connolly & Begg. 2005] sección 20.2.2, [García-Molina et al. 2009] sección 18.2.2, [Silberschatz et al. 2015] sección 14.6). **Es decir, el arco marca cuál de las dos transacciones se ejecutaría primero** en una planificación en serie equivalente. En el ejemplo de la sección anterior a la transacción T2 le llegaría un arco de T1 indicando que T1 ha de comenzar primero (i.e.; T1 → T2), en este caso porque ha de leer CP antes de que T2 lo cambie (uno de los posible conflictos que se da en el ejemplo, que tiene aún más conflictos).

Si el grafo resultante no presenta ciclos, es que se pueden reordenar las operaciones de manera que todos los conflictos conducen a que solo hay un orden posible de las transacciones en la planificación en serie equivalente. En el ejemplo, solo es posible ejecutar T1 antes que T2 en la planificación en serie equivalente, y de hecho tampoco hay ciclos porque hay solo una flecha. Por el contrario, un ciclo representaría que existen conflictos que exigen que una transacción Ti debiera empezar antes que otra Tj, pero además hay otros conflictos que exigen lo contrario (que Tj vaya antes que Ti), lo cual es imposible.

5. Implementación del aislamiento mediante bloqueos

La propiedad de aislamiento requiere que el resto de sesiones no vean los cambios no cometidos en nuestra sesión, y viceversa. Una forma de implementar el aislamiento es el uso de bloqueos. Actualmente, no es la tendencia en cuanto a implementar el aislamiento. Por ejemplo, ***Oracle* y *PostgreSQL* no lo utilizan, pero *SQL-Server* sí que mantiene esta técnica "por defecto"**[6], porque es la que garantiza la serializabilidad en todos los casos, como se discutirá en la sección 6.3. Distinguimos dos tipos de bloqueos:

- En un **bloqueo de lectura (o compartidos)** el ítem bloqueado por una sesión no puede ser modificado por otras sesiones, pero sí que puede ser leído.
- En un **bloqueo de escritura (o exclusivos)** el ítem bloqueado por una sesión no puede ser ni modificado ni leído por otras sesiones.

5.1. Aislamiento serializable mediante bloqueos: Protocolo 2PL

En un SGBD los bloqueos se gestionan de manera automática a través de un módulo llamado *Gestor de Concurrencia*. Esta gestión automática para el caso serializable funciona como muestra la Ilustración 3 ([Connolly & Begg. 2005] sección 20.2.3, [Silberschatz et al. 2015] secciones 14.9.1, 15.1.2) y [García-Molina et al. 2009]

6 Es posible hacer que *SQL-Server* se comporte de forma similar a *Oracle* o *PostgreSQL* para el caso READ COMMITTED, ver anexo 3.3.

secciones 18.3 y 18.4).

Esta forma de actuar por parte del gestor de concurrencia se conoce como *Protocolo de bloqueo de dos fases (ó 2PL = 2 Phase Locking)*. En 2PL hay una primera fase en la que solo se van adquiriendo bloqueos, o bien, convirtiendo bloqueos de lectura en bloqueos más estrictos (de escritura), y hay una segunda fase en la que solo se liberan bloqueos.

> "*Se dice que una transacción cumple con el protocolo de bloqueo de dos fases si todas las operaciones de bloqueo preceden a la primera operación de desbloqueo de la transacción*" ([Connolly & Begg. 2005] sección 20.2.3.).

Se puede demostrar matemáticamente, que el protocolo de dos fases garantiza la *serializabilidad* ***en cuanto a conflictos*** de las transacciones (ver sección 18.3.4 en [García-Molina et al. 2009]).

ADQUISICIÓN DE BLOQUEOS:

Si un dato D va a **leerse** por la transacción T:

1. Si T ya tiene un bloqueo sobre D (bien de lectura o bien de escritura) => se lee D.
2. Si no:
   1. Si es necesario esperar hasta que D **NO esté bloqueado para escritura** por ninguna otra transacción T'.
   2. Hacer un bloqueo de lectura sobre D.

      Nota: Si el dato ya estuviera bloqueado para lectura, habría al menos una segunda transacción manteniendo ese bloqueo. Es decir, el mismo dato podría estar bloqueado para lectura por distintas transacciones, y no está desbloqueado para lectura "del todo" hasta que todas las transacciones liberen dicho bloqueo.
   3. Leer D.

Si un dato D va a **escribirse** por la transacción T:

1. Si T ya tiene un bloqueo de escritura sobre D => se escribe en D
2. Si no:
   1. Si es necesario esperar hasta que D **NO esté bloqueado por ninguna otra transacción, ni para lectura, ni para escritura.**

      Nota: Si está bloqueado para lectura, se espera a que todas las transacciones que mantienen el bloqueo de lectura lo liberen.
   2. Si T ya tenía un bloqueo de lectura sobre D, lo **convierte** en bloqueo de escritura.

      Si no: hace un bloqueo de escritura sobre D.
   3. Escribir en D.

LIBERACIÓN DE BLOQUEOS: Al terminar la transacción se liberan todos los bloqueos.

Ilustración 3: Pseudocódigo 2PL.

5.1.1 Granularidad de los bloqueos

La *granularidad* de un bloqueo es la cantidad de información que se bloquea. En principio nos interesa que la granularidad sea lo más pequeña posible para favorecer que varias transacciones concurrentes no compitan por el mismo ítem bloqueado, lo cual da lugar a más y mayores esperas; y también es una fuente potencial de *deadlocks* o interbloqueos. Sin embargo, cuanto más fina es la granularidad exige más complejidad y recursos para mantenerla adecuadamente.

Hay bloqueos de **granularidad de fila**, de **granularidad de bloque o página** y de **granularidad de tabla**

típicamente. En el primer caso se bloquea la fila entera, en el segundo se bloquea por entero una página de la base de datos. En las implementaciones comerciales actuales la granularidad más fina de bloqueo es la fila (i.e., **no hay bloqueos a nivel de campo**. Esto es porque el gestor de concurrencia tendría que guardar mucha información para gestionar qué fila de qué campo es bloqueada, por qué transacción, y se asume que normalmente una granularidad de fila es más que suficiente).

Se dice que un bloqueo sufre una **escalada o escala,** cuando en una operación se tienen demasiados bloqueos de baja granularidad sobre la misma tabla (típicamente múltiples bloqueos de fila) y el SGBD decide que es más eficiente *"escalar"* el bloqueo a una granularidad superior, por ejemplo de bloque o de tabla. Así se ahorra:

1. Mantener la información de qué filas están bloqueadas, y
2. Cada vez que otra transacción concurrente T intente hacer una actualización, ya no tendría que comprobar una por una todas las filas bloqueadas para ver si alguna coincide con la fila que T pretende actualizar. Por ejemplo, si un UPDATE afecta a todas las filas de empleados masculinos, podría ser menos costoso bloquear la tabla entera que llevar la cuenta de qué filas (aproximadamente la mitad) corresponden a hombres, y bloquearlas. Además seguramente haya empleados masculinos en casi todos las páginas de la base de datos, por lo que escalar el bloqueo a granularidad de páginas tampoco es la mejor opción.

   Cuando un SGBD tiene la opción de escalar los bloqueos, la granularidad que decide tomar es, por tanto, una cuestión que forma parte del plan de la consulta.

Sin embargo, bloquear la tabla entera supone que cualquier otra transacción que tenga que actualizar esas tabla esté esperando el desbloqueo, y además favorece la aparición de *deadlocks*.

Oracle no escala nunca los bloqueos ([Oracle 2015] Capítulo 9, *Data Concurrency and Consistency*). Sin embargo *SQL-Server* si que lo hace ([SQL-Server 2016] ver sección *Dynamic Row-Level Locking,* pg 212 o sección *Lock Escalation* pg 1636), pudiendo escalar los bloqueos de fila a bloqueos de página (páginas de 8Kb), y de ahí a bloqueos de tabla. Según [Kriegel 2011] (pg 263), *PostgreSQL* tampoco escala[7].

Asumiendo que la granularidad que utiliza el SGBD es la más fina posible. Esto es, granularidad de fila:

1. El UPDATE provocaría un bloqueo de escritura sobre el dato modificado, bloqueando para escritura cada fila donde haya modificado un dato.
2. EL DELETE provocaría un bloqueo de escritura sobre cada fila borrada.
3. El INSERT en 2PL provoca el bloqueo de escritura sobre cada fila recién insertada. (ver sección 18.6.3 en [García-Molina et al. 2009]).
4. La SELECT en 2PL provoca un bloqueo de lectura sobre cada fila que contuviera un dato leído para que ninguna otra transacción lo cambie.

5.1.2 Ejemplos 2PL sobre SQL-Server

Ejemplo 1

Se crea la tabla actores según el siguiente *script*:

7 Según [Kriegel 2011] (pg 263), tampoco escala *MySQL* con *InnoDB*, pero *IBM DB2* 9.7 sí que lo hace.

```
drop table dbo.actores;

create table dbo.actores(
  id char(2) primary key,
  nombre varchar(15)
);

insert into actores values ('A1','Pepe'),('A2','Ana'),('A3','Juan'),('A4','Maria');
```

Ilustración 4: Datos de ejemplo utilizados en esta sección.

Sean las transacciones concurrentes T1, T2 y T3 cuyo comienzo ha sido marcado como comentarios en negrita en la Ilustración 5.

Explicación de la traza de ejecución paso a paso: En el paso 0 configuramos ambas como serializables (en *SQL-Server* hay que hacerlo antes de comenzar la transacción). En *SQL-Server* las transacciones comienzan con "`begin transaction`", que es una sintaxis recogida por estándar (ver sección 1.1).

En el paso 1 T1 hace una SELECT bloqueando para lectura todas las filas; por lo que el UPDATE en T2 se queda esperando a que acabe T1. Una vez que T1 termina tras el paso 3, T2 consigue el bloqueo de escritura y continua.

En el paso 4 T3 necesita bloquear par lectura todas las filas de la tabla, pero T2 mantiene el bloqueo de escritura sobre el actor A1, por lo que se queda en espera. Cuando T2 hace *commit*, el bloqueo de escritura se libera y T3 adquiere el bloqueo de lectura de A1 pudiendo ejecutar la SELECT.

Paso	Sesión 1	Sesión 2
0	`set transaction isolation level serializable;`	`set transaction isolation level serializable;`

1	`begin transaction;  --T1`
	
	`select * from actores;`	

2		`begin transaction;  --T2`
		
		`update actores`
		`set nombre='Jose'`
		`where id='A1';`

3	`commit;`	

4	`begin transaction;  --T3`
	
	`select * from actores;`	

5		`Commit;`
6	`Commit;`	

Ilustración 5: Esquema del Ejemplo 1.

Verificación de que la planificación obtenida por 2PL es serializable en cuanto a conflictos: La fila en conflicto es la del actor A1. El grafo de precedencia tiene los siguientes arcos:

- T1 → T2 (T1 ha de ejecutarse antes que T2 porque la SELECT de T1 es anterior al UPDATE y ambas están en conflicto con el actor A1).
- T2 → T3 (T2 ha de ejecutarse antes que T3 porque la SELECT de T3 es posterior al UPDATE y ambas transacciones tienen el mismo conflicto que antes).

Como cabía esperar de 2PL, el grafo no presenta ciclos, y la planificación que ha generado 2PL es serializable en cuanto a conflictos. Si observamos la explicación de cómo se han ido realizando los bloqueos y las esperas, la planificación ejecutada es equivalente a la planificación en serie (T1,T2,T3), que es la planificación que también podemos deducir del grafo T1 → T2 → T3.

Ejemplo 2

El esquema con las transacciones se presenta en la Ilustración 6

Explicación de la traza de ejecución paso a paso:

- En el paso 1 T1 adquiere un bloqueo de lectura sobre el actor A1.
- En el paso 2 T2 adquiere también un bloqueo de lectura sobre el actor A1. Lo importante es notar que no espera a T1, ya que los bloqueos de lectura son compatibles entre ellos.
- En el paso 3 T3 obtiene otro bloqueo de lectura sin esperar.
- En el paso 4 la T1 pide un bloqueo de escritura del actor A1. En realidad lo que pide es una conversión del bloqueo de lectura que ya tenía, en uno de escritura. Pero le toca esperar, pues las transacciones T2 y T3 aún tienen ese dato bloqueado para lectura.
- En el paso 5 la T3 finaliza su transacción liberando el bloqueo de lectura que ejercía. Pero ahora el actor A1 sigue bloqueado para lectura por T2, por lo que T1 sigue esperando.
- En el paso 6 T2 finaliza su transacción liberando el bloqueo de lectura que ejercía. Ya no quedan más bloqueos de lectura sobre el actor A1, salvo el de T1, por lo que T1 deja de esperar y consigue convertir su bloqueo de lectura en bloqueo de escritura, ejecutando acto seguido el UPDATE.
- En el paso 7, T1 ejecuta sin esperar la SELECT, pues tiene un bloqueo de escritura sobre el actor A1.
- Finalmente en el paso 8 se libera el bloqueo de escritura que tenía T1.

Paso	T1	T2	T3
0	`set transaction isolation level serializable;`	`set transaction isolation level serializable;`	`set transaction isolation level serializable;`

1	`begin transaction;`
	
	`select * from actores where id='A1';`		

2		`begin transaction;`
		
		`select * from actores where id='A1';`	

3			`begin transaction;`
			
			`select * from actores where id='A1';`

4	`update actores set nombre='Jose' where id='A1';`		
5			`commit;`
6		`commit;`	
7	`select * from actores where id='A1';`		
8	`commit;`		

Ilustración 6: Esquema del Ejemplo 2.

Verificación de que la planificación obtenida por 2PL es serializable: La fila en conflicto es la del actor A1. El grafo de precedencia sobre la planificación 2PL tiene los arcos:

- T2 → T1 :

  T2 ha de ejecutarse antes que T1 porque la SELECT de T2 lee datos anteriores al UPDATE de T1, y
- T3 → T1:

  T3 ha de ejecutarse antes que T1 porque la SELECT de T3 lee datos anteriores al UPDATE de T1

Como no hay ciclos, hay una planificación equivalente serializable en cuanto conflictos, que es bien (T2,T3,T1) – la

que se ha obtenido con 2PL – aunque también existiría otra posible: (T3,T2,T1).

Ejemplo 3

El esquema con las transacciones se presenta en la Ilustración 7. Detrás de cada "`begin transaction`" ha sido marcado como comentarios en negrita el nombre de cada transacción.

Paso	Sesión 1	Sesión 2	Sesión 3
0	`set transaction isolation level serializable;`	`set transaction isolation level serializable;`	`set transaction isolation level serializable;`

1	`begin transaction; --T1`
	
	`select * from actores where id='A1';`		

2		`begin transaction; --T2`
		
		`select * from actores where id='A1';`	

3			`begin transaction; --T3`
			
			`update actores set nombre='Jose' where id='A1';`

4	`commit;`		

5	`begin transaction; --T4`
	
	`select * from actores where id='A1';`		

6		`commit;`	
7	`commit;`		

8		`begin transaction; --T5`
		
		`update actores set nombre='Curro' where id='A1';`	

9			`Commit;`
10		`Commit;`	

Ilustración 7: Esquema del ejemplo 3.

Explicación de la traza de ejecución paso a paso siguiendo literalmente el procedimiento en la Ilustración 3:

- En los pasos 1 y 2 las transacciones T1 y T2 adquieren sendos bloqueos de lectura sobre el actor A1.
- En el paso 3 T3 se queda esperando a obtener un bloqueo de escritura.
- En el paso 4 T1 termina liberando su bloqueo de lectura, pero T3 sigue esperando porque T2 sigue ejerciendo en el dato un bloqueo de lectura.
- En el paso 5 T4 adquiere otro bloqueo de lectura sobre A1. Ahora mismo el actor A1 tiene 2 bloqueos de lectura, uno por parte de T2 y otro por parte de T4.
- En el paso 6 se libera el bloqueo por parte de T2, pero persiste el de T4, por lo que T3 sigue en espera.
- En el paso 7 se libera el bloqueo por parte de T4; T3 consigue por fin el bloqueo de escritura que venía esperando y se ejecuta su UPDATE.
- En el paso 8 T5 pide el bloqueo de escritura, pero no lo obtiene porque T3 ha bloqueado el dato para escritura.

- En el paso 9 T3 libera su boqueo de escritura y entra en acción el UPDATE de T5.

Verificación de que la planificación obtenida por 2PL es serializable: La fila en conflicto es la del actor A1. El grafo de precedencia sobre la planificación 2PL tiene los arcos:

- T1 → T3, T2 → T3 y T4 → T3 así como T1 → T5, T1 → T4 y T4 → T5
  Ya que T1, T2 y T4 han de ejecutarse antes que T3 y T5 porque sus SELECTs leen datos anteriores a los *commits* de los UPDATEs de T3 y al de T5.
- T3 → T5
  T3 ha de ejecutarse antes que T5 porque el UPDATE de T5 es el último

Como cabe esperar, no hay ciclos, y a través de la traza paso a paso podemos ver que la planificación serie equivalente en cuanto a conflictos que se ha obtenido con 2PL es (T1, T2, T4, T3, T5).

Ahora bien, como se indica al principio, esto sería la "*explicación de la traza de ejecución paso a paso* ***siguiendo literalmente el procedimiento en la Ilustración 3***". Si lo probásemos en SQL-Server, y casi con seguridad en cualquier otro SGBD que utilizase 2PL, la planificación que obtendríamos sería sin embargo, (T1, T2, **T3, T4**, T5). Esto se debe a una cuestión avanzada de implementación que se explica en el anexo "*VII.5 Consideraciones sobre la implementación de 2PL*".

Problema:

(Basado en consulta en tutorías del alumno Jorge Bernal curso 2018-19)
En un SGBD basado en bloqueos y con nivel de aislamiento SERIALIZABLE, hay dos transacciones concurrentes como las de la figura:

Paso	Transacción 1	Transacción 2
1	`BEGIN TRANSACTION`	

2		`BEGIN TRANSACTION`
		
		`UPDATE tabla_2`
		`SET campo_1='X'`
		`WHERE ID=5;`

3		`COMMIT;`
4	`SELECT * FROM tabla_2;`	

Ilustración 8: Traza de comandos correspondiente al problema.

Supongamos que existe la fila con ID=5, entonces en el paso 2, la transacción 2 en algún momento habrá ganado el bloqueo de escritura de dicha fila. En el paso 3, se liberaría dicho bloqueo.
Como la fila ha sido desbloqueda ¿veríamos en el paso 4 el valor nuevo?; y si es así ¿se rompe entonces la serializabilidad?
Solución:
Efectivamente en el paso 4 se ve el valor nuevo (se ha comprobado en *SQL-Server* que así es). Pero no se rompe la serializabilidad, ya que el resultado es el mismo que ejecutar en serie primero la transacción 2 y luego la transacción 1. Es decir, existe una planificación serializable (en cuanto a conflictos) equivalente que es (T2, T1).
Variante del problema: ¿Qué hubiera ocurrido si añadimos `SELECT * FROM tabla_2;` en el paso 1 de la transacción 1, tras el BEGIN TRANSACTION?, ¿las dos SELECTs darían distintos resultados violando la serializabilidad?.

Respuesta: La SELECT que hemos añadido ahora provoca un bloqueo de lectura, el UPDATE por tanto no ganará el

bloqueo de escritura hasta que la transacción 1 finalice, por lo que el UPDATE se queda en espera y ambas SELECTs darían el mismo resultado, tal y como exige la serializabilidad. Es decir, se ejecutan en serie pero ahora en orden inverso: (T1, T2).

Experimento 1

¿2PL bloquea las filas al insertarlas? El siguiente experimento sobre *SQL-Server* prueba que las filas recién insertadas son bloqueadas cuando se trabaja en nivel de aislamiento SERIALIZABLE.

Paso	T1	T2
0	`set transaction isolation level serializable;`	`set transaction isolation level serializable;`

1	`begin transaction;`
	
	`insert into actores values ('A5','XX');`	

2		`begin transaction;`
		
		`update actores`
		`set nombre='YYY'`
		`where ID='A5';`

3	`COMMIT;`	

Ilustración 9: Traza de comandos correspondiente al experimento.

Explicación de la traza de ejecución paso a paso:

- En el paso 1 la transacción T1 reserva un espacio para insertar la fila, crea la fila bloqueándola para escritura, y la rellena con los valores del actor A5. (ver sección 18.6.3 en [García-Molina et al. 2009], sección 15.18.2 en [Silberschatz et al. 2015]).
- En el paso 2 T2 pide un bloqueo para escritura de la fila del actor A5, como aún está bloqueada para escritura ha de esperar a que T1 termine y libere el bloqueo

Se puede comprobar que el grafo de precedencia solo tiene el arco T1 → T2, y que la planificación obtenida es equivalente a (T1, T2), (de hecho es (T1, T2)).

5.1.3 El fenómeno fantasma: extensión de 2PL con el protocolo de bloqueo de índices

¿Cómo se previene el fenómeno fantasma utilizando bloqueos? En la Ilustración 10, T2 presenta 2 consultas, antes y después de una inserción. Si la fila nueva se viese tras el *commit* de T1, se produciría un fenómeno fantasma.

Paso	T1	T2
0	`set transaction isolation level serializable;`	`set transaction isolation level serializable;`

1		`begin transaction;`
		
		`select * from actores`
		`where nombre > 'Juan';`

2	`begin transaction;`
	
	`insert into actores values ('A5','XX');`	

3		`select * from actores` `where nombre > 'Juan';`
4	`COMMIT;`	
5		`COMMIT;`

Ilustración 10: Fenómeno fantasma.

Explicación de la traza de ejecución paso a paso con 2PL puro:

- En el paso 1 la transacción T2 bloquea para lectura los actores de la tabla con nombres mayores que Pepe (María y Pepe según la Ilustración 4). Ejecuta entonces la SELECT y muestra las 2 filas correspondientes a estos dos actores.
- En el paso 2 la transacción T1 reserva un espacio para insertar la fila, crea la fila bloqueándola para escritura, y la rellena con los valores del actor A5.
- En el paso 3 T2 vuelve a ejecutar la consulta, pero necesita bloquear para lectura la nueva fila de A5, ya que el actor A5 se llama *XX* y cumple ser mayor que *Juan.* Como está bloqueada para escritura no lo consigue y tiene que esperar.
- En el paso 4 termina T1 liberando el bloqueo para escritura, entonces T2 adquiere el bloqueo de lectura sobre la fila recién insertada, vuelve a realizar la consulta que ahora vería la fila recién insertada junto las de María y Pepe, provocándose así un fenómeno fantasma.

5.1.3.a Protocolo de bloqueo de índices

Por tanto, 2PL por si mismo no es suficiente para garantizar que no ocurra el fenómeno fantasma, porque solo detecta conflictos entre datos reales; se dice que no contempla conflictos con datos *fantasma* (i.e., recién insertados en una transacción concurrente). La serializabilidad requiere que una consulta Q ejecutada durante una transacción T obtenga siempre el mismo conjunto de filas. Si esa consulta Q intenta acceder a una fila que aun no existe, la fila no debe de ser insertada por otras transacciones T' antes de que T termine. Si se permitiera a T' hacer la inserción de esa fila aparecería en las siguientes ejecuciones de Q, que es lo que se conoce como una fila *"fantasma".*

Para resolverlo se utiliza el **protocolo de bloqueo de índices**, la idea es, que asumiendo que las tablas tienen índices y se utilizan en las consultas, hay que bloquear las hojas del índice donde podrían ir a parar las filas que se insertasen y entrasen en conflicto con una determinada consulta.

El protocolo de bloqueo de índices sigue las siguientes **reglas** (tomado de la sección 15.8.2 en [Silberschatz et al. 2015], en la que se adapta el protocolo al caso en el que el índice se implemente con árbol B^+):

Regla 1. Toda tabla ha de tener al menos un índice (esto, en general, es cierto porque al menos toda tabla está indexada por su clave primaria, pero también por sus claves ajenas, UNIQUEs y otros índices que queramos añadir con CREATE INDEX).

Regla 2. Toda transacción Ti puede acceder a las filas de una tabla, tanto con el fin de leerlas como de escribirlas, solamente después de haberlas encontrado previamente utilizando uno o más índices. A efectos de este protocolo, el caso especial de que hubiera que acceder a todas las filas de la tabla o buscar por un campo no indexado lo trataremos como si fuese una exploración por "todas" las hojas de alguno de los índices.

Regla 3. Toda transacción Ti que realiza una búsqueda o consulta, debe de bloquear en modo compartido (i.e.; con bloqueos de lectura) todos los nodos hojas del índice al que acede.

Regla 4. Toda transacción Ti no puede insertar, actualizar o borrar una fila sin actualizar todos los índices de la tabla adecuadamente, por lo que debe de obtener bloqueos en modo exclusivo (i.e.; bloqueos de escritura) sobre todos los nodos hoja de cada uno de los índices afectados por la operación (que en el caso del INSERT y el DELETE, al afectar a todos los campos, son todos los índices definidos en la tabla: su clave primaria, claves ajenas, UNIQUEs e índices *ad hoc* creados con CREATE INDEX para esa tabla).

Para seguir la explicación del protocolo tomaremos el ejemplo en [SQL-Server 2016] , sección *Key-Range Locking* (pg 1633) que muestra la Ilustración 11. Tenemos una tabla de *empleados* con un índice para el campo *nombre.* Por simplicidad el árbol solo tiene dos niveles; la raíz y las hojas. Cada entrada en el índice, por ejemplo, (*Adam, Byron*) apunta en el siguiente nivel al nodo en el que la entrada de menor valor es *Adam* y la entrada de mayor valor es

Byron. Cuando se realiza una búsqueda, por ejemplo *Carl*, en el nodo raíz se busca la primera entrada en la que el valor de la primera columna es mayor o igual que el valor buscado, que en el ejemplo es, [*Carl-Duncan*], En las hojas, cada entrada del índice mantiene apuntadores a las distintas páginas de la tabla que contienen datos en el rango de esa entrada de índice. Por ejemplo, la entrada del índice (*Frazer, Gabriel*) mantiene un puntero a la página *n* por contener ésta al menos a *Freddie,* y otro a la *n+1* por contener al menos a *Freya.*

Operaciones de lectura

- Si Ti hace una **SELECT** … WHERE <*predicado*>, debe bloquear para lectura todos los nodos hoja del índice utilizado (nota que el bloqueo es de lectura, es decir, en modo compartido para que esos nodos del índice puedan leerse por otras consultas, pero no cambiarse). El predicado puede consistir en una búsqueda de una fila, de un rango de filas, o de toda la tabla en el caso de que no hubiese WHERE.
  Además realizará el bloque de lectura 2PL sobre las filas seleccionadas.
  En el ejemplo, supongamos que T1 hace la consulta de rango
  `SELECT nombre FROM empleados WHERE nombre BETWEEN 'A' AND 'D';`
  esto provocaría que se bloqueasen todas las entradas de índice que hay desde (*Adam*, *Adrian)* hasta *(Clive, Dale)* inclusive. Esta acción previene que se inserten filas que necesiten modificar el índice en estas entradas. Por ejemplo, no sería posible insertar *Abigail* porque habría que modificar la entrada *(Adam, Adrian),* ni tampoco se puede añadir nada después de *Clive,* como por ejemplo *Connie*, porque se requeriría modificar la entrada (*Clive*, *Dale*).
  Si la consulta no devolviese ninguna fila, por ejemplo, suponiendo no hay ningún empleado llamado *Bill:*
  `SELECT nombre FROM empleados WHERE nombre = 'Bill';`
  también habría bloqueo de índice en las entradas de índice afectadas, en este caso (*Ben, Bing*), evitando que se insertase *Bill* por otra transacción.

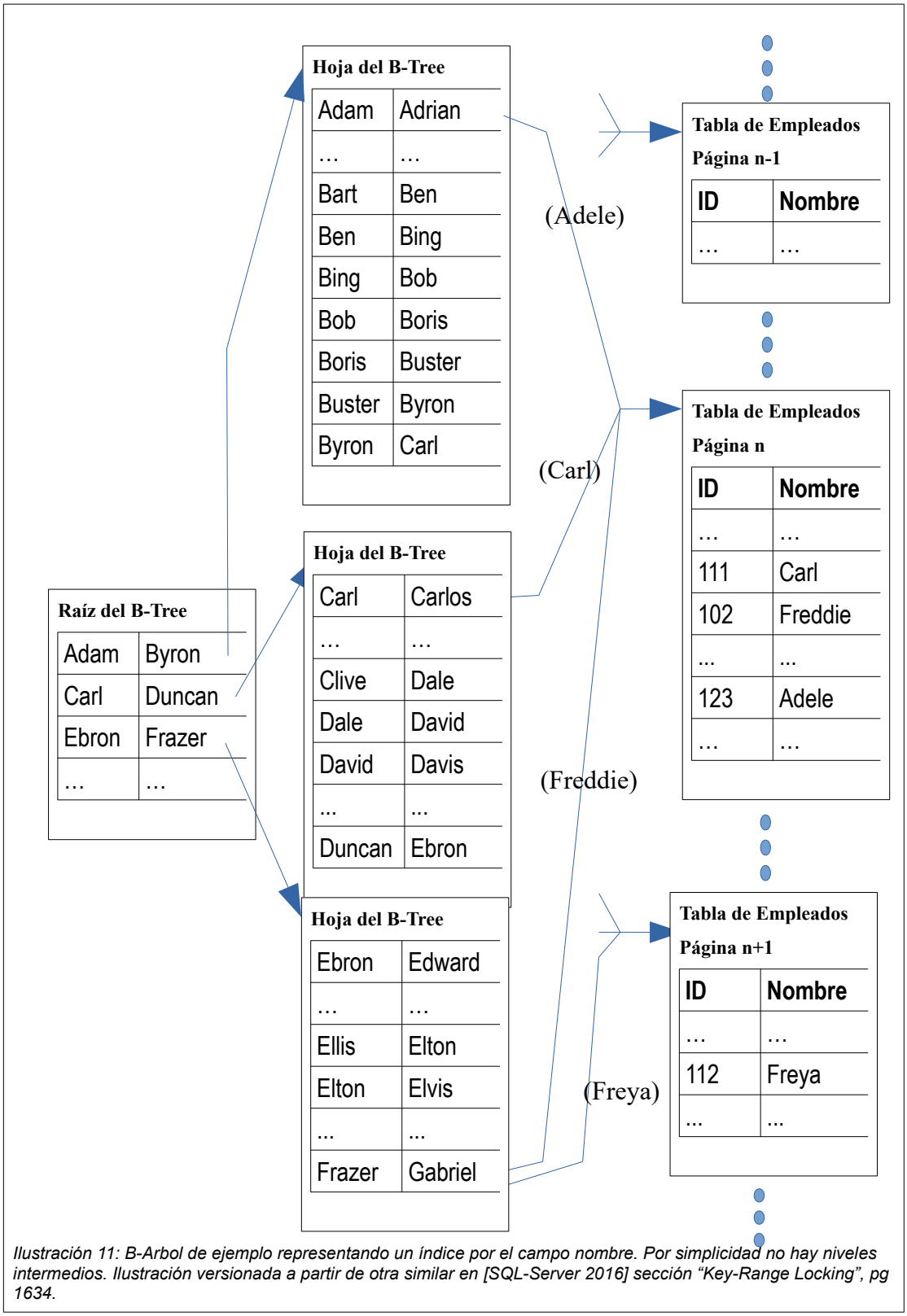


Ilustración 11: B-Arbol de ejemplo representando un índice por el campo nombre. Por simplicidad no hay niveles intermedios. Ilustración versionada a partir de otra similar en [SQL-Server 2016] sección "Key-Range Locking", pg 1634.

Por tanto en el ejemplo de la Ilustración 10, lo que realmente ocurre es:

Explicación de la traza de ejecución paso a paso con el protocolo de bloqueos de índice:

Supondremos que hemos indexado la tabla *actores* por el nombre del actor. Además también supondremos que no se escalan bloqueos a lo largo de la explicación.

- En el paso 1 la transacción T2
  - Bloquea alguna entrada del índice por *nombre* que se ha definido en modo compartido/lectura para que no se puedan insertar más actores que cumplan el WHERE. En este caso no es posible insertar actores con nombre mayor que Juan.

    En la *regla 4* hemos dicho que "Toda transacción Ti no puede insertar, actualizar o borrar una fila sin actualizar todos los índices de la tabla adecuadamente". Por tanto, bloqueando el índice en modo compartido, se bloquean para lectura las filas a las que apunta el índice, por lo que el bloqueo de lectura de las filas que lee la SELECT (i.e., María y Pepe) es innecesario por estar en el rango "`>Juan`" que se ha utilizado en el bloqueo de índice. Por tanto, si se aplica un protocolo de bloqueo de índice "puro" no hace falta bloquear las filas para lectura, basta bloquear el índice.
  - Se ejecuta entonces la SELECT y muestra las dos filas.
- En el paso 2 la transacción T1 solicita el bloqueo de todos los índices necesarios para insertar la fila (*A5,XX)* (i.e.; la clave primaria y el índice por el *nombre).* Como el índice de *nombre* está bloqueado impidiendo cambios en las filas con nombres mayores que Juan, la transacción T1 tiene que esperar.
- En el paso 3, T2 vuelve a ejecutar la consulta, pero la nueva fila de A5 no ha conseguido aún ser insertada. Por lo que la SELECT vuelve a devolver el mismo resultado que en la SELECT del paso 1
- En el paso 5, T2 termina liberando el bloqueo para lectura del índice, entonces T1 adquiere el bloqueo sobre el índice necesario para insertar el nombre XX, reserva un espacio para insertar la fila, crea la fila, la rellena con los valores del actor A5 y actualiza el índice convenientemente. Nota que no se bloquea la fila, solo el índice. Nuevamente, la *regla 4* hace también innecesario en el protocolo de bloqueo de índice bloquear para escritura las filas, basta bloquear el índice.
- En el paso 4, T2 libera el bloqueo de escritura sobre el índice.

El resultado final, por lo tanto, ha sido en este caso una planificación en serie (T2, T1).

Operaciones de escritura

Recordemos la *regla 4*: "Toda transacción Ti no puede insertar, actualizar o borrar una fila sin actualizar todos los índices de la tabla adecuadamente". Por ello, cuando se hace un **INSERT-DELETE-UPDATE** se pide un bloqueo exclusivo de las hojas de los índices correspondientes a las filas a insertar, borrar o actualizar.

- En el caso del **INSERT** se bloquearán las hojas correspondientes a los nuevos valores insertados (sección 15.8.2 en [Silberschatz et al. 2015]). En el ejemplo de los empleados, si hacemos:

  ```
  INSERT INTO empleados VALUES ( 888, 'Dan' );
  ```

  Por un lado se intenta un bloqueo de escritura sobre la entrada del índice (*Dale, David*). Por otro lado, asumiendo que el campo ID es la clave primaria (recuerda que toda clave primaria se implementa mediante un índice), también habría que intentar un bloqueo de escritura sobre la entrada correspondiente al 888 en el índice de la clave primaria.

  Si por ejemplo una SELECT anterior al INSERT en una transacción concurrente hubiera utilizado el índice de *nombre* para obtener, entre otros, empleados de la entrada (*Dale, David*):

  ```
  SELECT * FROM empleados WHERE nombre BETWEEN 'D' AND 'E';
  ```

  la inserción tendría que esperar a que la SELECT liberase el bloqueo de escritura de índice al final de su transacción, con lo que se garantiza que nunca va a leer la fila fantasma recién insertada.

Una vez obtenidos los bloqueos de escritura de índice se procede como en 2PL, reservando el espacio para la fila y rellenándola de valores.

Como ya sabemos del caso de lectura, no es necesario bloquear la fila insertada para escritura, si el SGBD mantiene el bloqueo de escritura en los índices hasta el final de la transacción, pues el propio bloqueo de índice impide que se modifique por otra transacción la fila recién insertada según la *regla 4*.

- En el caso del "**DELETE"** y del "**UPDATE cuando el campo actualizado no es el campo indexado utilizado en la búsqueda"**, se bloquearán las hojas correspondientes al valor de la clave de búsqueda (sección 15.8.2 en [Silberschatz et al. 2015]). Por ejemplo, tanto en:

  ```
  DELETE FROM empleados WHERE nombre = 'Bob';
  ```

  como en:

  ```
  UPDATE empleados SET telefono='123456789' WHERE nombre='Bob';
  ```

  supuesto existe ese campo teléfono y se ha buscado por el índice del *nombre*, se bloqueará para escritura la entrada para (*Bob, Boris*). Esto en el caso del DELETE impide que otra transacción modifique la fila borrada, y en el caso del UPDATE, impide que la borre o la modifique. En definitiva hace el mismo papel que el bloqueo de escritura que se haría sobre las filas con 2PL y por eso tampoco es necesario bloquear las filas, basta bloquear el índice.
- En el caso de que el "**UPDATE fuese sobre el campo de la clave de búsqueda**", por ejemplo:

  ```
  UPDATE empleados SET nombre='Bill' WHERE nombre='Bob';
  ```

  se bloquearían para escritura las entradas del índice tanto correspondientes al valor de búsqueda, es decir (*Bob, Boris*), como para el nuevo valor, en este caso (*Ben, Bing*) (sección 15.8.2 en [Silberschatz et al. 2015]). El primer bloqueo (*Bob, Boris*) se justifica de igual manera que el DELETE y el UPDATE *de un campo distinto del de búsqueda* (se evita que se modifique o borre esa fila). El segundo bloqueo (*Ben, Bing*) se justifica de igual forma que el que se hacía en el caso del INSERT, provocando que el UPDATE quede en espera si una SELECT de otra transacción concurrente hubiera leído el valor antiguo, evitando así una lectura no repetible o una lectura sucia según el caso. Por ejemplo, si otra transacción T2 ha hecho:

  ```
  SELECT * FROM empleados WHERE nombre = 'Bill';
  ```

  (suponemos no hay ningún *Bill* y devuelve cero filas) y, mientras está activa T2, nuestra transacción T1 intenta hacer el UPDATE y COMMIT

  ```
  UPDATE empleados SET nombre='Bill' WHERE nombre='Bob';
  COMMIT;
  ```

  tendrá que esperar a que acabe T1 (como ocurría con el INSERT), porque de lo contrario si T2 repitiese la SELECT ahora devolvería la fila modificada (lectura no repetible). Nota que no es necesario bloquear las filas porque en el fondo estos dos bloqueos de índice tienen el mismo papel que el bloqueo de la fila para escritura con 2PL puro.

5.1.3.b Combinación de 2PL y protocolo de bloqueo de índices

En la sección anterior nunca se bloqueaban las filas porque siempre existía un bloqueo de índice en cada operación, que además de garantizar que no aparecieran filas fantasmas, garantizaba que no aparecieran ni lecturas sucias ni lecturas no repetibles.

Sin embargo, el protocolo de índices visto tiene una suposición no demasiado realista en la *regla 2* "Toda transacción Ti puede acceder a las filas de una tabla, tanto con el fin de leerlas como de escribirlas, solamente después de haberlas encontrado previamente utilizando uno o más índices". Parece evidente que en el mundo real en no pocas ocasiones no va a existir un índice que se utilice en la consulta, porque por ejemplo no van a estar todos los campos

indexados normalmente, o porque la utilización del índice sea un ingrediente mas de una parte de la consulta, etc …
Es el planificador de consultas quien decide si utilizar un índice y cuál en cada caso.
Si obviamos esa *regla 2* podemos pensar en un protocolo híbrido que siga en parte 2PL y el protocolo de bloqueo de índices, como por ejemplo ocurre en *SQL-Server* que combina bloqueos de índice con bloqueos de fila ([SQL-Server 2016] sección "Key-Range Locking", pg 1633.). Para ello, al descartar la *regla 2,* la *regla 4* se ve afectada en que no todas las escrituras van a requerir actualizar el índice, y por tanto no van a bloquearlo siempre hasta final de la transacción. El protocolo resultante en ese caso es:

1. Acompañar las SELECTs con un bloqueo de lectura de las filas afectadas tras el bloqueo de lectura del índice, como se hacía en 2PL, por si por ejemplo se hace un UPDATE que no utilice el índice bloqueado.
2. Acompañar INSERT-DELETE-UPDATE del bloqueo de escritura de la fila, como se hacía en 2PL, tras el bloqueo de escritura de índice, por idénticas razones. Además:
    1. El bloqueo de índice que hace el INSERT se puede liberar una vez se obtiene ese bloqueo de fila, sin esperar la final de la transacción, pues solo se utiliza para comprobar que no entra en conflicto con una SELECT anterior al INSERT de una transacción activa.
    2. En el caso del DELETE y el UPDATE el planificador de consultas, al darnos el algoritmo óptimo para acceder a las filas afectadas por el WHERE del DELETE/UPDATE nos va a decir si se va a utilizar un índice o no, y con ello si se va a optar por bloquear los índices para escritura junto con las filas, o solamente las filas una a una. Por ejemplo:
        - Si las filas a las que se accede en el WHERE del DELETE/UPDATE están localizadas en bloques de disco muy concretos (por ejemplo, `WHERE provincia='Ceuta'` en una tabla de empresas es una condición que cumplirán muy pocas filas y lo normal es que haya filas con esa provincia en escasas páginas de la tabla), la velocidad de acceso a esas filas se beneficia del índice porque no hay que rastrear todas las filas de la tabla, y además el bloqueo de índice es oportuno porque habrá bastante coincidencia entre las páginas bloqueadas y las filas a borrar/actualizar.
        - Por el contrario habrá casos en los que haya una gran dispersión de las filas a borrar/actualizar a lo largo de todas las páginas (por ejemplo, `WHERE provincia='Madrid'` en una tabla de empresas es una condición que cumplirán muchas filas y lo normal es que haya filas con esa provincia en muchas páginas de la base de datos). En este caso el optimizador de consultas preferirá rastrear todas las filas de la tabla antes que utilizar el índice, y bloquear, por tanto, solo las filas de Madrid individualmente, y no las entradas del índice (si bien ese bloqueo de filas podría no llegar a producirse debido a una escalada del bloqueo de fila a bloqueo de tabla porque quizás la mayoría de las empresas de la tabla fueran de Madrid).

5.1.4 Aislamiento READ COMMITTED mediante bloqueos

El caso READ COMMITTED es menos exigente, pues permite la lectura de datos cometidos por transacciones concurrentes (i.e.; lectura no repetible), como es el caso del ejemplo de la Ilustración 12 basado en los datos del Ejemplo 1 (página 23).
Si intentamos probar en *SQL-Server* ese ejemplo (*SQL-Server* por defecto implementa la concurrencia mediante bloqueos), veremos lo siguiente:

1. En el paso 1 se ejecuta la SELECT. Podríamos pensar en coherencia con 2PL, que se ha generado un bloqueo de lectura de todos los actores.
2. En el paso 2, sin embargo, se produce el UPDATE sin realizar ninguna espera, es decir, si hubiera un

bloqueo de lectura en el paso anterior, no tendría ningún efecto.

3. La consulta del paso 3 se queda en espera, por lo que intuimos que la transacción T2 sí que ha hecho un bloqueo. Si en el paso 3, en lugar de una SELECT, hiciéramos un UPDATE, también habría una espera, por lo que todo indica que el bloqueo que se ha producido en el paso 2, es un bloqueo de escritura.
4. En el paso 4 T2 libera su bloqueo de escritura, entonces la SELECT del paso 3 deja de esperar y se ejecuta viéndose los cambios producidos por el UPDATE (i.e.; la lectura no repetible).

Paso	Transacción T1	Transacción T2
0	`set transaction isolation level read committed;`	`set transaction isolation level read committed;`

1	`begin transaction;`
	
	`select * from actores;`	

2		`begin transaction;`
		
		`update actores`
		`set nombre='Jose'`
		`where id='A1';`

3	`select * from actores;`	
4		`Commit;`
5	`Commit;`	

Ilustración 12. Esquema para una lectura no repetible con READ-COMMITTED.

Es decir, suponiendo la políticas de adquisición y liberación de bloqueos vistas para el caso SERIALIZABLE, **en el caso READ COMMITTED no son necesarios los bloqueos de lectura**, por lo que el protocolo se simplifica notablemente respecto a 2PL, como muestra la Ilustración 13.

ADQUISICIÓN DE BLOQUEOS:

Si un dato D va a **leerse** por la transacción T:

1. Si es necesario esperar hasta que D NO esté bloqueado para escritura por ninguna otra transacción T'.
2. Realizar la lectura (no se hace bloqueo de lectura)

Si un dato D va a escribirse por la transacción T:

1. Si T ya tiene un bloqueo de escritura sobre D => se escribe en D
2. Si no:
    1. Si es necesario esperar hasta que D NO esté bloqueado tanto para escritura, como para lectura por ninguna otra transacción T'.

       Nota: Aunque en READ COMITTED no hay bloqueos de lectura, D podría estar bloqueado para lectura eventualmente por otra/otras transacciones concurrentes con nivel de aislamiento serializable.
    2. T hace un bloqueo de escritura sobre D.
    3. Escribir en D.

LIBERACIÓN DE BLOQUEOS: Al terminar la transacción se liberan todos los bloqueos.

Ilustración 13: READ COMMITTED mediante bloqueos.

En el caso de las inserciones opera como si fuesen escrituras, por tanto la transacción bloquea para escritura las filas insertadas como ocurría en el caso serializable (ver Experimento 1)

Muy Importante; esto hace que READ COMMITTED sea más **ágil** que SERIALIZABLE, ya que:

1. En el caso de escritura de un dato D, no necesita esperar que se acaben todas las transacciones que

estuvieran bloqueando a D para lectura.

2. Además como en READ COMMITTED está permitido el fenómeno fantasma, tampoco es necesario hacer bloqueos de índice.

Por otro lado READ COMMITTED es suficiente en la mayoría de las ocasiones, por lo que debemos evitar exigir un aislamiento SERIALIZABLE si no es del todo necesario.

6. Aislamiento de Instantánea y Concurrencia Multiversión

El aislamiento, además de mediante bloqueos, se suele implementar en los SGBD comerciales utilizando un esquema de versionado de los datos. *Oracle*, *PostgreSQL* lo implementan en lugar de un esquema puramente basado en bloqueos, mientras que *SQLServer,* aunque por defecto usa bloqueos, permite configurar el SGBD para que implemente el aislamiento mediante versiones; y en general parece que ésta es la tendencia de los últimos años.

6.1. Protocolos Multiversión

Se conoce por MVCC[8] el control de concurrencia multiversión o a partir de múltiples versiones de los datos. En este tipo de implementación de la serializabilidad, cada operación de escritura sobre un dato D genera una nueva versión del dato D' y a la vez guarda la versión original D sobre que la que se ha aplicado la escritura. Cuando una transacción concurrente tiene que leer ese dato **nunca espera**, sino que lee la versión original D del dato para mantener la serializabilidad y así no cometer, bien una lectura sucia, bien una lectura no repetible, dependiendo respectivamente de si la transacción que cambió el dato sigue activa o cometió el cambio.
Los protocolos multiversión se basan en asignar una ***marca de tiempo*** a cada transacción. Esta marca de tiempo sirve para saber qué transacción ha comenzado antes, cara a mantener la serializabilidad. Las marcas de tiempo se pueden implementar mediante un *timestamp* que registre el comienzo de cada transacción, pero también podría ser un número secuencial que se va incrementando cada vez que comienza una nueva transacción, lo cual es lo más habitual en la práctica. Existen principalmente dos variantes de MVCC:

1. El protocolo de ordenación por marcas temporales multiversión (sección 20.2.6 en [Connolly & Begg. 2005], sección 18.8.5 en [García-Molina et al. 2009], sección 15.6.1 en [Silberschatz et al. 2015]).
2. El protocolo de aislamiento de instantánea (sección 15.7.1 en [Silberschatz et al. 2015]).

En ambas variantes de protocolo multiversión, cada versión de un dato estará etiquetada con la marca de tiempo correspondiente a la transacción que la generó, pero en el protocolo de ordenación por marcas temporales se requiere además guardar otra marca de tiempo de la última transacción que "leyó" cada versión Di de cada dato D para detectar posibles conflictos, lo que causa que guardar tantas marcas de tiempo para cada versión del dato acabe añadiendo complejidad y restando eficiencia. La variante del protocolo multiversión que no tiene este problema (i.e.; solo se guarda la marca de tiempo de la transacción que genera la versión), y que es la que implementan los sistemas comerciales, es el llamado ***aislamiento de instantánea***.

6.2. Aislamiento de Instantáneas

En esta implementación del aislamiento el SGBD es capaz de utilizar las versiones de los datos para reconstruir una instantánea[9] del contenido de la base de datos en un determinado instante para una determinada sesión. En el caso

8 *Multi-Version Concurrency Control.*
9 El lector debe tomar la palabra instantánea (*snapshot* en inglés) como un símil de fotografía que captura el estado de la base de datos en un instante dado.

serializable esto permite reconstruir el estado de la base de datos en el instante del comienzo de cada transacción T, de manera que si T hace dos veces la misma SELECT devolverá el mismo resultado incluso a pesar de los cambios cometidos en otras transacciones. Como cualquier tecnología de versiones, elimina por completo los bloqueos de lectura permitiendo que las lecturas nunca esperen a las escrituras, pero también consume recursos en gestionar y almacenar las versiones, y además no resuelve en todos los casos las planificaciones serializables de forma correcta, como veremos más adelante (ver sección 6.3).

Se dice que una transacción **T' es una transacción concurrente con T** si en el momento de inicio de T, T' aún no ha terminado (Sección 15.7 en [Silberschatz et al. 2015]). Para saber si dos transacciones son concurrentes se necesitan registrar dos marcas de tiempo por cada transacción: el momento de inicio de T que denotaremos **ts_inicio(T)**, y el momento final de T, que denotaremos como **ts_fin(T)**. En los SGBD comerciales normalmente la marca de tiempo no se implementa mediante un *timestamp*, sino mediante una secuencia (i.e.; un generador de números enteros consecutivos). En *Oracle* la marca temporal que se implementa vía una secuencia se conoce como *SCN* o *System Change Number,* en *PosgreSQL* *txid* o *transaction ids*, en SQL-Server *XSN* o *Transaction Sequence Number*, etc...

Supongamos que la transacción T en curso, en un instante t de la misma, quiere saber si T' es o ha sido en algún momento, concurrente con ella. Hay que analizar varios casos de concurrencia (ver Ilustración 14).

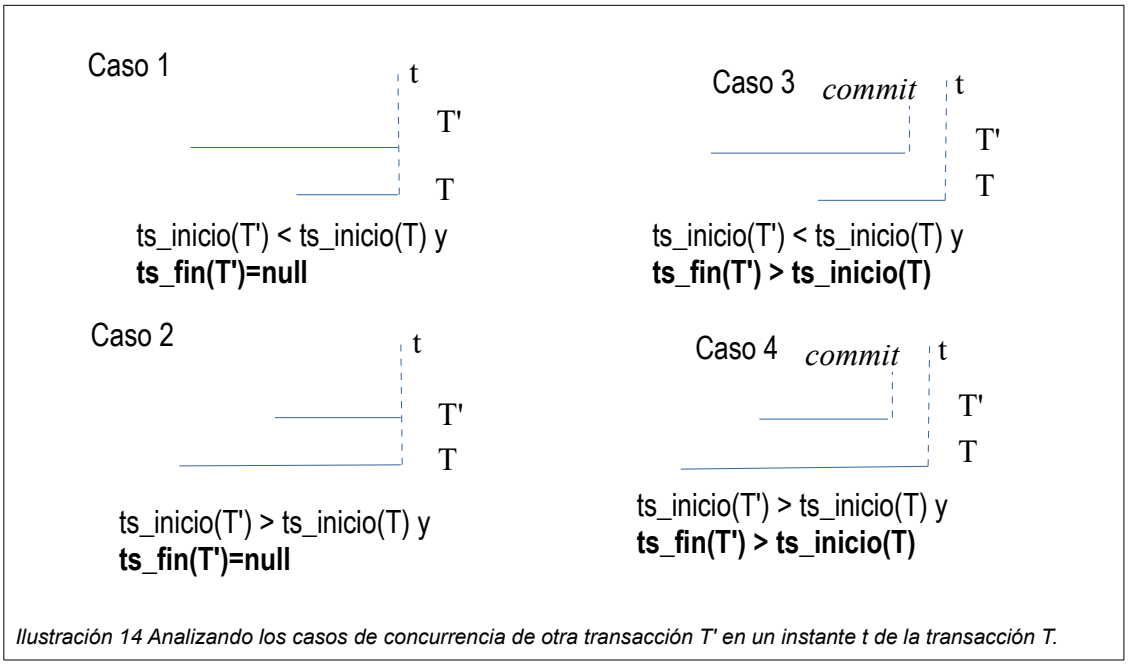


Ilustración 14 Analizando los casos de concurrencia de otra transacción T' en un instante t de la transacción T.

- Casos 1 y 2, independientemente de cuando comience T', en el instante t T' no ha finalizado. Cuando una transacción aún no ha terminado ts_fin(T) tomará un valor especial (por ejemplo, *null*; en el caso de *Oracle* toma SCN=0).
- Casos 3 y 4, T' había terminado ya con *commit*, pero durante un tiempo si que fue concurrente con T. En ese caso ts_fin(T') > ts_inicio(T).

6.2.1 Generación de versiones

Asumiremos que cada versión de fila tiene internamente al menos estas informaciones:

1. Algún tipo de identificador que permita saber a qué fila se corresponde esa versión.
2. Algún marcador *booleano*, llamémosle *borrado,* que permita saber si la versión ha sufrido un DELETE.
3. El ts_inicio de la transacción T que creó esa versión. Concretamente:

1. Cada fila **insertada** por T genera una nueva versión que es etiquetada con el ts_inicio(T) de la transacción que la insertó. Si la fila insertada es posteriormente modificada/borrada en la misma transacción T, no se genera una nueva versión, sino que se modifica la versión existente, por lo que conserva el ts_inicio(T) del momento en que fue insertada.
2. Cada fila insertada por otra transacción T' y **modificada** por la transacción T, la primera vez que se modifica por T genera una nueva versión que es etiquetada con el ts_inicio(T). Si la fila insertada es modificada/borrada en la misma transacción T no se genera una nueva versión, sino que se modifica la versión existente, por lo que conserva el ts_inicio(T) del momento en que fue insertada.
3. Cada fila **borrada** en T, en la que el borrado es la única operación que se hace en T, también genera una nueva versión etiquetada con ts_inicio(T), en la que es irrelevante el contenido de la fila. El marcador *borrado* ha de ponerse a TRUE.
   Si la fila borrada en T sufrió anteriormente otras operaciones en dicha transacción (e.g.; inserción y/o modificaciones), no se genera una nueva versión, sino que se modifica la versión existente producto de esas operaciones, por lo que conserva el ts_inicio(T) del momento en que fue insertada, pero el marcador *borrado* ha de ponerse a TRUE.

6.2.2 Creación de la instantánea

De cara a la creación de la instantánea para una transacción T:

1. Se necesita una *tabla de transacciones* que registre para cada una de las transacciones recientes:
   1. Su **estado:** Activa, Cometida, Retrocedida.
   2. Su **ts_inicio** y su **ts_fin**. En el caso de una transacción inacabada (i.e.; activa), ts_fin tomará un valor especial. Por ejemplo, *null*.
   3. El ***nivel de aislamiento***
2. Se necesita un protocolo de creación de instantánea para cada vez que una transacción necesite acceder a la versión adecuada de cada fila. Este protocolo obedece las siguientes reglas para cada fila que vaya a leer T
   1. Si existe una versión generada por T (i.e.; ts_inicio de esa versión es el mismo que ts_inicio registrado para T en la tabla de transacciones)
      - Se devuelve esa versión.
   2. En caso contrario
      1. Se crea una colección de versiones candidatas a ser seleccionadas, las cuales han de cumplir:
         1. Que hayan sido cerradas con *commit*. Para ello, tomamos el ts_inicio de la versión, lo buscamos en la tabla de transacciones, y examinamos el estado de dicha transacción comprobando que es *cometido*.
            Esta condición es suficiente en el caso READ COMMITTED. Si el aislamiento fuera serializable, además se exigiría que
         2. Que la versión la haya generado una transacción que haya sido cometida antes de que empezara T. Para ello, tomamos la entrada de la tabla de transacciones examinada en el paso anterior y leemos el valor de ts_fin. Comprobamos que ese ts_fin sea menor (i.e.; anterior) al ts_inicio(T).
      2. Una vez generada la lista de versiones candidatas, se toma aquella versión generada por la transacción T' con ts_fin(T') máximo (i.e., la que ha acabado más recientemente).

Según el SGBD que estemos utilizando existen distintas implementaciones más eficientes de esta mismo procedimiento que pueden guardan las mismas informaciones de manera distinta a como muestra el presente apartado

(ver anexo 3).

6.2.2.a Ejemplo de construcción de instantánea:

En la Ilustración 15 se muestra un ejemplo con mas de dos versiones. Las transacciones han sido numeradas y marcadas como comentarios en negrita.

Paso	Sesión 1	Sesión 2 (serializable)	Sesión 3 (serializable)
1	**--Trans T1** INSERT INTO alumnos VALUES ('Pepe', 3);		
2	COMMIT;		
3		**--Trans T2** SELECT * FROM alumnos WHERE nombre='Pepe';	
4	**--Trans T3** UPDATE alumnos SET nota=5 WHERE nombre='Pepe';		
5	COMMIT;		
6		SELECT * FROM alumnos WHERE nombre='Pepe';	
7			**--Trans T4** SELECT * FROM alumnos WHERE nombre='Pepe';
8	**--Trans T5** UPDATE alumnos SET nota=9 WHERE nombre='Pepe';		
9			SELECT * FROM alumnos WHERE nombre='Pepe';
10	SELECT * FROM alumnos WHERE nombre='Pepe';		
11	UPDATE alumnos SET nota=7 WHERE nombre='Pepe';		
12	SELECT * FROM alumnos WHERE nombre='Pepe';		
13	COMMIT;		
14		COMMIT;	
15		--T6 UPDATE alumnos SET nota= nota + 1 WHERE nombre='Pepe';	
16		COMMIT;	
17	**--Trans T7** DELETE FROM alumnos WHERE nombre='Pepe';		
18			SELECT * FROM alumnos WHERE nombre='Pepe';
19	COMMIT;		
20		**--Trans T8** SELECT * FROM alumnos WHERE nombre='Pepe';	
21		COMMIT;	COMMIT;

Ilustración 15: Ejemplo concurrencia de instantánea con más de una versión del dato.

En el **paso 1** comienza T1, ts_inicio(T1) = 1. Se inserta la fila de Pepe. Supongamos una tabla *alumnos* con los campos *nombre* y *nota.* El nombre del alumno insertado es Pepe y su nota un 3.

Hay una única versión de esa fila, con ese ts_inicio=1:

nombre **nota** ts_inicio borrada

```
Pepe        3          1          falso
```

La tabla de transacciones queda así en este momento:

Transacción	Estado	Aislamiento	ts_inicio	ts_fin
T1	Activa	(no especificado en el ejemplo)	1	*Null*

ts_fin es nulo porque la transacción aún no ha acabado. La columna transacción se ha añadido para que sean más fáciles las explicaciones, pues en principio las transacciones no tienen nombre, y en la práctica se identifican por su ts_inicio.

En el **paso 2** la transacción T1 hace *commit*, y la tabla de transacciones se actualiza cambiando el estado de la transacción T1 y el ts_fin:

Transacción	Estado	Aislamiento	ts_inicio	ts_fin
T1	Cometida	(no especificado en el ejemplo)	1	2

En el **paso 3** comienza la transacción T2. La tabla de transacciones lo refleja así:

Transacción	Estado	Aislamiento	ts_inicio	ts_fin
T1	Cometida	(no especificado en el ejemplo)	1	2
T2	Activa	Serializable	3	*Null*

Para realizar la SELECT del paso 3 habrá que crear la instantánea. No hay versiones creadas por T2, ya que no hay versiones que tengan ts_inicio igual al ts_inicio(T2)=3. Así que hay que crear la lista de versiones candidatas creadas por otras transacciones. Solo hay una versión de fila que cumple el WHERE , la cual tiene ts_inicio=1.

1. Primero comprobamos que esa versión está cometida. Vamos a la tabla de transacciones con ese ts_inicio, vemos que se trata de la transacción T1 que tiene un estado cometido; luego en principio es una versión candidata a mostrarse en la SELECT.
2. Como el nivel de aislamiento de la transacción que hace la SELECT (i.e.; T2) es serializable, además hay que comprobar que T1 se cometiera antes de que empezara T2 (con *read committed* no habría que hacer esta comprobación). Efectivamente, es así porque ts_fin(T1)=2 < ts_inicio(T2). Luego nuevamente es una versión candidata a mostrarse en la SELECT.
3. Como no hay más versiones, el siguiente paso, que consiste en tomar la versión más reciente entre todas las versiones candidatas (i.e.: aquella que tenga el ts_fin máximo), resulta trivial y selecciona la única versión de la fila de Pepe que hay. El resultado de la SELECT es:

```
nombre      nota
Pepe        3
```

En el **paso 4** comienza la transacción T3:

Transacción	Estado	Aislamiento	ts_inicio	ts_fin
T1	Cometida	(no especificado en el ejemplo)	1	2
T2	Activa	Serializable	3	*Null*
T3	Activa	(no especificado en el ejemplo)	4	*Null*

Se realiza el UPDATE que cambia la nota de Pepe, con lo que ahora hay dos versiones de la fila:

```
nombre      nota       ts_inicio   borrada
Pepe        3          1           falso
Pepe        5          4           falso
```

En el **paso 5** se comete T3 lo que produce la actualización de la fila de T3 en la tabla de transacciones.

Transacción	Estado	Aislamiento	ts_inicio	ts_fin
T1	Cometida	(no especificado en el ejemplo)	1	2
T2	Activa	Serializable	3	*Null*
T3	Cometida	(no especificado en el ejemplo)	4	5

En el **paso 6**, T2 realiza otra vez la misma consulta. Para construir la instantánea seguimos los mismo pasos que antes. Nuevamente, no hay versiones creadas por T2, así que hay que generar la lista de versiones candidatas creadas por otras transacciones.

1. Para la versión de Pepe con nota 3 y ts_inicio=1, el ts_inicio nos indica que la transacción que creó la versión fue T1. En la tabla de transacciones vemos
   1. Que T1 tiene estado cometido
   2. Que ts_fin(T1)=2 < ts_inicio(T2)=3, es decir, fue cometida antes de que empezase T2.

   Por tanto es una versión candidata a mostrarse.
2. Para la versión de Pepe con nota 4 y ts_inicio=4, , el ts_inicio nos indica que la transacción que creó la versión fue T3. En la tabla de transacciones vemos
   1. Que T3 tiene estado cometido
   2. Que ts_fin(T3)=5 ⊀ ts_inicio(T2), es decir, fue cometida después de que empezase T2.

   Por tanto es una versión que en una transacción serializable **no** es candidata a mostrarse.
3. Como hay una única versión candidata, el ts_fin de la transacción que la creó es el máximo y es la única que se muestra. Por lo que el resultado de la SELECT vuelve a ser:

```
nombre      nota
Pepe        3
```

En el **paso 7** comienza T4:

Transacción	Estado	Aislamiento	ts_inicio	ts_fin
T1	Cometida	(no especificado en el ejemplo)	1	2
T2	Activa	Serializable	3	*Null*
T3	Cometida	(no especificado en el ejemplo)	4	5
T4	Activa	Serializable	7	*Null*

Se realiza otra vez la misma consulta. Para construir la instantánea seguimos los mismo pasos otra vez. No hay versiones creadas por T4, ya que no hay versiones que tengan ts_inicio igual al ts_inicio(T4)=7. Así que hay que crear la lista de versiones candidatas creadas por otras transacciones:

1. Para la versión de Pepe con nota 3 y ts_inicio=1, el ts_inicio nos indica que la transacción que creó la versión fue T1. En la tabla de transacciones vemos
   1. Que T1 tiene estado cometido
   2. Que ts_fin(T1)=2 < ts_inicio(T4)=7, es decir, fue cometida antes de que empezase T4.

   Por tanto, es una versión candidata a mostrarse.
2. Para la versión de Pepe con nota 4 y ts_inicio=4, el ts_inicio nos indica que la transacción que creó la versión fue T3. En la tabla de transacciones vemos

   Que T3 tiene estado cometido

   Que ts_fin(T3)=5 < ts_inicio(T4)=7, es decir, fue también cometida antes de que empezase T4.

   Por tanto, es otra versión candidata a mostrarse.

3. Como no hay más versiones, el siguiente paso, que consiste en tomar la versión más reciente entre todas las versiones candidatas (i.e.: aquella que tenga el ts_fin máximo). Una tiene ts_fin(T1)=2, y la otra ts_fin(T3)=5, luego tomaremos esta última. El resultado de la SELECT, por tanto, es:

nombre	nota
Pepe	5

En el **paso 8** comienza la transacción T5.

Transacción	Estado	Aislamiento	ts_inicio	ts_fin
T1	Cometida	(no especificado en el ejemplo)	1	2
T2	Activa	Serializable	3	*Null*
T3	Cometida	(no especificado en el ejemplo)	4	5
T4	Activa	Serializable	7	*Null*
T5	Activa	(no especificado en el ejemplo)	8	*Null*

Se realiza el UPDATE que cambia la nota de Pepe, con lo que ahora hay tres versiones de la fila:

nombre	nota	ts_inicio	borrada
Pepe	3	1	falso
Pepe	5	4	falso
Pepe	9	8	falso

En el **paso 9** T4 repite la consulta, (sigue sin haber versiones generadas por T4). La nueva versión con ts_inicio=8, nos indica que corresponde a la transacción 5 que no tiene estado cometido, por lo que no es una versión candidata a mostrarse por la SELECT. El resto del proceso de evaluación de versiones candidatas será idéntico al del paso 7, por lo que la versión seleccionada será la misma que en ese paso.

En el **paso 10** es T5 la que realiza la consulta. En este caso si que hay una versión generada por T5. ts_inicio(T5)=8, y la versión con ese ts_inicio es la que devuelve la consulta:

nombre	nota
Pepe	9

En el **paso 11** T5 vuelve a modificar la nota de Pepe, por lo que no se crea una nueva versión sino que se modifica la existente:

nombre	nota	ts_inicio	borrada
Pepe	3	1	falso
Pepe	5	4	falso
Pepe	7	8	falso

En el **paso 12** T5 realiza la consulta. Nuevamente hay una versión generada por T5, pero ahora esa versión tiene la nota cambiada a 7. La consulta devuelve:

nombre	nota
Pepe	7

En el **paso 13** T5 hace *commit,* en el **paso 14** también hace *commit* T2, y en el **paso 15** comienza T6:

Transacción	Estado	Aislamiento	ts_inicio	ts_fin
T1	Cometida	(no especificado en el ejemplo)	1	2
T2	Cometida	Serializable	3	14
T3	Cometida	(no especificado en el ejemplo)	4	5
T4	Activa	Serializable	7	*Null*
T5	Cometida	(no especificado en el ejemplo)	8	13
T6	Activa	Serializable	15	*Null*

En el **paso 15** T6 hace UPDATE. Este UPDATE es especial, porque encierra una lectura de la nota anterior de Pepe (i.e.; "`nota= nota + 1`"). Esa lectura requiere una construcción de instantánea para saber qué versión de la nota

leer. Al construir la instantánea vemos que no hay versiones generadas por T6.

- Pepe con nota 3 => ts_inicio = 1. En la tabla de transacciones vemos que la entrada con ts_inicio=1 está cometida y tiene ts_fin = 2, que es menor que el ts_inicio de T6, que es 15. Por lo que la versión de Pepe con nota 3, es tomada como candidata.
- Pepe con nota 5 => ts_inicio = 4. En la tabla de transacciones vemos que la entrada con ts_inicio=4 está cometida y tiene ts_fin = 5, que es menor que el ts_inicio de T6, que es 15. Por lo que la versión de Pepe con nota 5, también es tomada como candidata.
- Pepe con nota 7 => ts_inicio = 8. En la tabla de transacciones vemos que la entrada con ts_inicio=8 está cometida y tiene ts_fin = 13, que es menor que el ts_inicio de T6, que es 15. Por lo que la versión de Pepe con nota 7, también es tomada como candidata.

De las 3 versiones, la que tiene el ts_fin máximo es la de nota 7, (ts_fin=13). Por lo que, el UPDATE genera una nueva versión con nota 8.

nombre	nota	ts_inicio	borrada
Pepe	3	1	falso
Pepe	5	4	falso
Pepe	7	8	falso
Pepe	8	15	falso

En el **paso 16** se comete la transacción T6 y se actualiza la tabla de transacciones convenientemente (ver más adelante en el paso 17).

En el **paso 17** comienza la transacción T7.

Transacción	Estado	Aislamiento	ts_inicio	ts_fin
T1	Cometida	(no especificado en el ejemplo)	1	2
T2	Cometida	Serializable	3	14
T3	Cometida	(no especificado en el ejemplo)	4	5
T4	Activa	Serializable	7	*Null*
T5	Cometida	(no especificado en el ejemplo)	8	13
T6	Cometida	Serializable	15	16
T7	Activa	(no especificado en el ejemplo)	17	*Null*

Se produce el borrado de la fila de Pepe. Conceptualmente lo podemos tratar como si apareciera una nueva versión en el que un campo *booleano* que nos dijera si la versión está borrada o no:

nombre	nota	ts_inicio	borrada
Pepe	3	1	falso
Pepe	5	4	falso
Pepe	7	8	falso
Pepe	8	15	falso
Pepe		17	true

En 6.2.1 se dijo que las versiones mantendrían algún tipo de identificador del sistema para poder reconocer que son versiones de la misma fila. Por simplicidad hemos supuesto que siendo el nombre clave primaria podríamos usarlo en el ejemplo a este fin. Es por eso que en la versión borrada se ha mantenido el nombre, para así poder significar que la fila borrada es la de Pepe.

En el **paso 18** T4 repite la consulta, (sigue sin haber versiones generadas por T4). Las versiones con ts_inicio 1 y 4 son analizadas de igual manera que en el paso 7, de manera que ambas versiones se consideran candidatas a ser mostradas en la instantánea.

- Para la versión de Pepe con nota 7 y ts_inicio=8, el ts_inicio nos indica que la transacción que creó la

versión fue T5. En la tabla de transacciones vemos

1. Que T5 tiene estado cometido
2. Que ts_fin(T5)=13 ⊀ ts_inicio(T4)=7, es decir, fue cometida después de que empezase T4.

Por tanto, es una versión que en una transacción serializable **no** es candidata a mostrarse.

- Para la versión de Pepe con nota 8, pasa algo similar. Su ts_inicio es 15. El ts_inicio nos indica que la transacción que creó la versión fue T6. En la tabla de transacciones vemos
  1. Que T6 tiene estado cometido
  2. Que ts_fin(T6)=16 ⊀ ts_inicio(T4)=7, es decir, fue cometida después de que empezase T4.

  Así que también es una versión que en una transacción serializable **no** es candidata a mostrarse.
- Para la versión borrada de Pepe, ts_inicio=17, el ts_inicio nos indica que la transacción que creó la versión fue T7. Como no está cometida, tampoco es una versión candidata a mostrarse en la instantánea.
- Como las dos únicas versiones candidatas son las mismas que las del paso 7, se muestra el mismo resultado que en aquel paso.

El **paso 19** comete la transacción T7 (ver tabla de transacciones en el siguiente paso).

En el **paso 20** comienza T8:

Transacción	Estado	Aislamiento	ts_inicio	ts_fin
T1	Cometida	(no especificado en el ejemplo)	1	2
T2	Cometida	Serializable	3	14
T3	Cometida	(no especificado en el ejemplo)	4	5
T4	Activa	Serializable	7	*Null*
T5	Cometida	(no especificado en el ejemplo)	8	13
T6	Cometida	Serializable	15	16
T7	Cometida	(no especificado en el ejemplo)	17	19
T8	Activa	Serializable	20	*Null*

Se repite por última vez la consulta, (no hay versiones generadas por T8):

1. Para la versión de Pepe con nota 3 y ts_inicio=1, el ts_inicio nos indica que la transacción que creó la versión fue T1. En la tabla de transacciones vemos
   1. Que T1 tiene estado cometido
   2. Que ts_fin(T1)=2 < ts_inicio(T8)=20, es decir, fue cometida antes de que empezase T8.

   Por tanto es una versión candidata a mostrarse.
2. Para la versión de Pepe con nota 5 y ts_inicio=4, el ts_inicio nos indica que la transacción que creó la versión fue T3. En la tabla de transacciones vemos
   1. Que T3 tiene estado cometido
   2. Que ts_fin(T3)=5 < ts_inicio(T8)=20, es decir, fue también cometida antes de que empezase T8.

   Por tanto es una versión candidata a mostrarse.
3. Para la versión de Pepe con nota 7 y ts_inicio=8, el ts_inicio nos indica que la transacción que creó la versión fue T5. En la tabla de transacciones vemos
   1. Que T5 tiene estado cometido
   2. Que ts_fin(T5)=13 < ts_inicio(T8)=20, es decir, fue también cometida antes de que empezase T8.

   Por tanto es una versión candidata a mostrarse.
4. Para la versión de Pepe con nota 8 y ts_inicio=15, el ts_inicio nos indica que la transacción que creó la versión fue T6. En la tabla de transacciones vemos
   1. Que T6 tiene estado cometido

   2. Que ts_fin(T6)=16 < ts_inicio(T8)=20, es decir, fue también cometida antes de que empezase T8.

   Por tanto es una versión candidata a mostrarse.

5. Para la versión de Pepe borrada y ts_inicio=17, el ts_inicio nos indica que la transacción que creó la versión fue T7. En la tabla de transacciones vemos
   1. Que T7 tiene estado cometido
   2. Que ts_fin(T7)=19 < ts_inicio(T8)=20, es decir, fue también cometida antes de que empezase T8.

   Por tanto es una versión candidata a mostrarse.
6. Como no hay más versiones, el siguiente paso, que consiste en tomar la versión más reciente entre todas las versiones candidatas (i.e.: aquella que tenga el ts_fin máximo). Esta es la versión borrada, pues la transacción que la creó, es T7 y ts_fin(T7)=19 que es el máximo. Por tanto al crear la instantánea nos quedamos con la versión borrada, y la SELECT acaba por devolver cero filas.

Muy Importante: En este ejemplo, por ganar simplicidad, las operaciones SQL afectaban solo a una fila, y por ello la instantánea a construir siempre era de una sola fila. En un caso real, las operaciones afectarán a múltiples filas, y por ejemplo, reconstruir la instantánea que devuelve una SELECT supone mezclar versiones con distinto ts_inicio, pues en general cada fila de esa instantánea corresponderá con una versión que podría haber sido creada en una transacción diferente.

6.2.3 Mantenimiento de la serializabilidad

De cara al mantenimiento de la *serializabilidad en cuanto a conflictos*, el aislamiento de instantánea:

1. Normalmente[10] no genera conflictos porque T lea un dato modificado por una transacción concurrente T', ni porque T modifique un dato leído por una transacción concurrente T'. Esto es así porque, como hemos dicho, el gestor de concurrencia se encarga de construir la instantánea que permite que la lectura se produzca sobre la versión del dato que se correspondería con un comportamiento serializable.
2. Sin embargo, no es capaz de resolver el conflicto debido a que T modifique el mismo dato que una transacción concurrente T'.

Existen dos variantes del aislamiento de instantánea para resolver esta última cuestión (Sección 15.7 en [Silberschatz et al. 2015]):

1. ***Primer commit gana***: En caso de conflicto de escritura, se resuelve en favor de la primera transacción que haga *commit*. En el momento de hacer *commit* cada transacción T comprueba si ha existido otra transacción concurrente T' que haya hecho *commit* y haya escrito algún dato D en el que T también haya escrito. Si existiese dicho dato, T abortaría (hace *rollback* implícito) y lanza un error.
2. ***Primera actualización gana***: Es la que utilizan los típicamente los SGBD comerciales. En esta estrategia **se han de mantener los bloqueos de escritura** sobre los datos modificados hasta el fin de la transacción. En caso de este conflicto de escritura se resuelve en favor de la primera transacción que bloqueó el dato para escritura.

   Cuando una transacción T va a modificar un dato pide el bloqueo de escritura **de la versión más reciente del mismo** si aún no lo tiene.

   Nota: Los SGBD implementan alguna forma de acceder rápidamente a la versión más reciente:

   1. Por ejemplo *Oracle* y *SQL-Server* guardan la versión más reciente en un espacio distinto que el

10 Hay casos raros en los que sí que hay conflictos lectura-escritura. Estos casos se reportan en la sección 6.3, pero son inusuales.

resto de versiones (i.e.; *Oracle* guarda las versiones que no son la actual en el *tablespace UNDO* y *SQL-Server* en una base de datos temporal llamada *tempdb*).

2. Por ejemplo *PostgreSQL* mantiene una lista enlazada de las versiones por orden cronológico. Al final de la lista está la versión más reciente.

Puede haber 2 casos:

Caso 1: Si el dato no está bloqueado, T consigue el bloqueo, se comprueba si el dato ya ha sido modificado por una transacción concurrente T' que ya ha hecho *commit*.

En términos de implementación esto supone en un primer paso examinar la marca de tiempo de cada versión de la fila, que contendrán cada una su propio valor ts_inicio(T'). Con esto se consigue saber quién es la transacción T' que hizo la modificación que generó esa última versión. Una vez identificada T' se comprueba en la tabla de transacciones si T' hizo *commit* durante el transcurso de T, lo cual ocurriría si se verifica que ts_fin(T') > ts_inicio(T)

- En ese caso T abortaría (hace *rollback* implícito)[11] y lanza un error.
- En caso contrario T escribe el dato y continua.

Caso 2: Si el dato está bloqueado por la transacción concurrente T', T ha de esperar a que T' termine.

Nota: Una forma de optimizar el acceso a la información de qué transacción bloquea qué fila, es que la propia versión más reciente de la fila contenga algún campo interno con el ts_inicio de la transacción que la bloquea. En caso de que ese campo interno valga *null*, es que no está bloqueada. *Oracle* hace algo parecido.

Para ello, nuevamente se examina en la tabla de transacciones si el estado de T' es *Activa.* En ese caso T' no habría acabado. Una vez acabe:

- Si T' termina con *rollback* liberará el bloqueo, T adquiere el bloqueo ejecutando a partir de ese momento todos los pasos del *Caso 1*.
- Si T' termina con *commit*, también se libera el bloqueo, pero en esta ocasión T aborta[11] (hace *rollback* implícito) y lanza un error.

Para entender en **un caso práctico** por qué es bueno que dé este error, tomemos el siguiente **ejemplo**: Supongamos una transacción mediante la que se reserva un asiento en un avión. En principio es una transacción que sigue los pasos de la Ilustración 16.

Nota: El ejemplo se toma alguna licencia. En próximos capítulos veremos que no es conveniente alargar las transacciones incluyendo los tiempos de interacción con los usuarios, como ocurre en el paso 2 de T1 y el paso 3 de T2.

Asumimos un nivel de aislamiento serializable. En esta ocasión estamos en el "caso 2": el UPDATE del paso 4 habría de esperar a que se liberase el bloqueo sobre la fila de *aviones* en el paso 5 y acto seguido la transacción T2 retrocedería y daría un mensaje de error.

Nota: El error que devuelve *Oracle* es "***ORA-08177: can't serialize access for this transaction."*** En el caso de *PostgreSQL* devuelve "***ERROR: no se pudo serializar el acceso debido a un update concurrente. Estado SQL:40001***." El aislamiento de instantáneas de *SQL-Server* solo implementa el nivel *read-committed*, por lo que no puede darse el caso en el que devolviese este tipo de error.

11 *Oracle* y *PostgreSQL* realmente no abortan la transacción, sólo abortan el comando. En el caso de Oracle con un segundo intento de *commit* podríamos incluso forzar el acometimiento de la transacción. PostgreSQL por el contrario no deja ejecutar ningún comando DML hasta que no se ejecute *commit* o *rollback;* pero aunque ejecutes *commit* lo ignora ejecutando un *rollback* implícito.

Si no lo hiciera así:

1. Pepe se quedaría sin asiento a pesar de estar en la creencia de que si que lo tiene, lo cual es un comportamiento erróneo que aparecerá más veces en este capítulo: ***El problema de la actualización perdida***. Se conoce con el nombre de problema de la actualización perdida ([Connolly & Begg. 2005] sección 20.2.1, [Silberschatz et al. 2015] sección 15.9.3, [Kyte 2010], capítulo 6) a la ocurrencia de una operación de actualización que en apariencia se ha completado con éxito por un usuario, pero sin embargo ha sido sobrescrita por otro usuario. La actualización perdida se hubiera dado con *read-committed*, pues no nos hubiera avisado con ningún mensaje de error.
2. Podemos hacer el grafo de precedencia y ver que hay un ciclo entre ambas transacciones: T1 → T2 (T1 ha de leer el estado del asiento antes de que T2 realice el UPDATE del mismo), y T2 → T1 (T2 ha de leer el estado del asiento antes de que T1 realice el UPDATE del mismo), por lo que no es una planificación serializable en cuanto a conflictos.

Observa que si por ejemplo el paso 2 de la transacción T1 se acortara en el tiempo, el *commit* de T1 podría haber ocurrido antes que el UPDATE de T2. Si fuese así, nota que al hacer el *commit* de T2 estaríamos en el "caso 1". T2 adquiere el bloqueo de escritura sin esperas y en el momento del *commit* descubre que el dato ya ha sido modificado por T1, con lo que también aborta y lanza un error, evitando otra actualización perdida.

Paso	**Transacción T1**	**Transacción T2**

1	--El operador obtiene la lista de asientos vacantes
	`SELECT nroAsiento`
	`FROM aviones`
	`WHERE avion='x'`
	`AND ocupadoPor IS NULL;`	

2	--El operador habla con el cliente dándole a elegir	--El operador obtiene la lista de asientos vacantes
		`SELECT nroAsiento`
		`FROM aviones`
		`WHERE avion='x'`
		`AND ocupadoPor IS NULL;`

3	--El cliente (Pepe) hace la reserva del asiento 10.
	`UPDATE aviones`
	`SET ocupadoPor = 'Pepe'`
	`WHERE avion='x'`
	`AND nroAsiento=10;`
		--El operador habla con el cliente dándole a elegir

4		--El cliente (Juan) hace la reserva del asiento 10.
		`UPDATE aviones`
		`SET ocupadoPor = 'Juan'`
		`WHERE avion='x'`
		`AND nroAsiento=10;`

5	`Commit;`	
6		`Commit;`

Ilustración 16. Ejemplo de conflicto de escritura en concurrencia de instantánea "Primera actualización gana".

6.2.3.a Conflictos de Escritura y otros niveles de aislamiento

En el caso 2 de *Primera actualización gana* si las dos transacciones serializables concurrentes intentan cambiar la misma fila (i.e.; si hay un conflicto de escritura), se ha visto que hasta que no acabe la primera transacción no podrá

continuar la segunda. Existe un bloqueo de escritura, el cual también aparece en el aislamiento *read-committed*.
En la Ilustración 17 vemos un esquema de experimento para comprobarlo, en el que las dos transacciones intentan hacer un UPDATE sobre la fila con *id=30*. En el experimento iremos cambiando el nivel de aislamiento de la transacción de la sesión 2 (i.e.; la transacción que se cierra la última). Veremos que el nivel de aislamiento en la sesión 1 (i.e.; la transacción que se cierra primero) no influirá en los resultados de los experimentos.
Los experimentos se han hecho con dos SGBD que soportan aislamiento serializable mediante instantánea, como son *Oracle* y *PostgreSQL* (*SQL-Server* solo soporta *read-committed* en su implementación del aislamiento de instantánea).

Paso	**Sesión 1 de SQL*Plus**	**Sesión 2 de SQL*Plus**
1		SQL>insert into mi_tabla values (30, 'x'); 1 row created. Commit; *Commit complete.*
2	SQL>SET TRANSACTION ISOLATION LEVEL <el que sea, depende del experimento>; Transaction set.	SQL>SQL>SET TRANSACTION ISOLATION LEVEL <el que sea, depende del experimento>; Transaction set.
3	SQL> update mi_tabla set conn='c1' where id=30; 1 row updated.	
4		SQL> update mi_tabla set conn='c2' where id=30; ***Se queda esperando a que se cierre la transacción de la sesión 1***
5	SQL> commit *o rollback depende del experimento*	
6		*Hay casos en los que el update conn='c2' tiene lugar, hay otros casos en los que da error.*

*Ilustración 17: Esquema de experimento de dos transacciones que intentan cambiar la misma fila. La ilustración intenta emular una interfaz Oracle SQL*Plus, pero el experimento es extrapolable a cualquier SGBD que soporte aislamiento de instantánea serializable.*

Las transacción de la sesión 2 se ejecuta con READ COMMITTED

Supongamos que la segunda transacción se intentan ejecutar en *read-committed*. En la transacción de la primera sesión, probaremos los niveles de aislamiento serializable y *read-committed,* viendo que no influirá en el resultado del experimento. En la ilustración se muestra que asumimos que el UPDATE de la sesión 1 se realiza **un poco antes** que el de la sesión 2.

Entonces, cualquiera que sea el aislamiento de la sesión 1, observamos que la sesión 2 se quedará **esperando** hasta que la sesión 1 hace acaba su transacción. Es decir, la sesión 1 hace un bloqueo de escritura cualquiera que ea su nivel de aislamiento:

- **Si la sesión 1 acaba la transacción con *commit*.** La fila 30 guardará 'c1' en el campo *conn,* pero al

liberarse el bloqueo de escritura sobre la fila 30, la sesión 2 dejará de estar en espera, con lo que finalmente la sesión 2 cambiará su valor a 'c2' (i.e.; otro caso de *actualización perdida*).

- **Si la sesión 1 acaba la transacción con *rollback***. El resultado final es el mismo. En este caso la fila 30 en la sesión 1 conservará el valor 'x' en el campo *conn,* pero al liberarse el bloqueo de escritura sobre la fila 30, acto seguido también la sesión 2 dejará de estar en espera, con lo que finalmente la sesión 2 cambiará nuevamente su valor a 'c2'.

El valor final del campo *conn* no dependerá de si la sesión 1 acaba con ***commit*** o ***rollback***.

La transacción de la sesión 2 se ejecuta con SERIALIZABLE

Si repetimos el experimento anterior con la sesión 2 en modo SERIALIZABLE **NO** pasa lo mismo. Independientemente del nivel de aislamiento en la sesión 1, nuevamente se produce el bloqueo de la fila por parte de la sesión 1. Además:

- **Si la primera transacción acaba con *commit***, la segunda transacción retrocede y devuelve un error cuando es liberado el bloqueo de la fila (i.e.;en el paso 5 finalizado con *commit*), tal y como habíamos descrito en el caso 2 de *Primera Actualización Gana* en la sección 6.2.

  Nota importante: Aunque *PostgreSQL* sí que retrocede la transacción de la sesión 2 (i.e.; *rollback* implícito) tal y como habíamos dicho, ***Oracle* solo aborta la operación UPDATE y no hace *rollback* implícito** de la transacción de la sesión 2 (paso 4). Esto significa que en *Oracle* la transacción de la sesión 2 sigue viva a la espera de que en dicha transacción se hagan más operaciones SQL y/o se cierre con *commit/rollback*; si bien lo más práctico/habitual es hacer *rollback* de la transacción que ha generado el error, y volver a ejecutarla desde el principio.
- **Por el contrario, si la primera transacción hubiera acabado en *rollback***, el bloqueo de la fila 30 se libera sin cambios, y la transacción 2 escribe 'c2'.

La Tabla 4 resume los posibles resultados del experimento.

Caso	Nivel de aislamiento de la sesión 2	La sesión 1 hace commit o rollback	¿Qué ocurre?
A	*READ COMMITTED*	*Commit o Rollback* indistintamente	La transacción 2 espera a que acabe la 1, y se graban los cambios de la transacción 2.
B	*SERIALIZABLE*	*Commit*	La sesión 2 se queda esperando hasta que la sesión 1 hace *commit*. La sesión 2 devuelve error: En Oracle **ORA-08177: can't serialize access for this transaction.** La sesión 2 solo aborta la operación de la transacción que estaba en espera, pero las operaciones anteriores a ésta siguen adelante. Se recomienda hacer *rollback* de la transacción de la sesión 2 y volver a ejecutarla desde el principio. En PostgreSQL ***ERROR: no se pudo serializar el acceso debido a un update concurrente. Estado SQL:40001***. Además la sesión 2 hace rollback implícito.
C	*SERIALIZABLE*	*Rollback*	La transacción 2 espera a que acabe la 1 con *rollback*, y se graban los cambios de la transacción 2.

Tabla 4: Resumen del experimento en el que dos transacciones intentan modificar la misma fila.

IMPORTANTE: Estás experimentando con *Oracle* y te aparece el error *ORA-08177* cuando realmente el conflicto de escritura no existe, ya que cada transacción está escribiendo en una fila distinta de la misma

tabla. Esto es un problema de cómo implementa *Oracle* el aislamiento de instantánea, y tiene una solución sencilla. En ese caso se recomienda la lectura del anexo 4.5. donde se explica cómo solucionarlo. *PostgreSQL* no da nunca este problema.

¿Por qué para provocar el error solo, es necesario que la transacción serializable sea la que se queda esperando?

1. Lógicamente el SGBD nunca hace la comprobación de si hay un conflicto de escritura en las transacciones *read committed.* Esto obliga a que al menos la transacción de la sesión 2 tenga que ser serializable.
2. Cuando una transacción es serializable, como en el caso de la transacción de la sesión 2, se aplica el protocolo de *primera actualización gana* considerando todos los posibles conflictos de escritura con las transacciones concurrentes con independencia de su nivel de aislamiento. Es decir, la sesión 2, al ser serializable, se comporta como si el resto de sesiones concurrentes también lo fueran (i.e.; comprueba que la planificación de esa transacción de la sesión 2, junto con el resto de transacciones que son concurrentes a ella, tengan una planificación equivalente en cuanto a conflictos).
3. La transacción de la sesión 1 del experimento, aunque fuese serializable, en el momento de su *commit* no devuelve ningún error según lo visto en el protocolo de *primera actualización gana*, porque es la primera en hacer *commit,* por lo que no influye su nivel de aislamiento.

El experimento anterior trata de dos UPDATE sobre la misma fila, pero del mismo podría extrapolarse a cuando **una de las transacciones borra esa fila mientras la otra la modifica**. Se deja para el alumno la deducción y experimentación de estos casos.

6.2.3.b Conflicto de escritura en dos filas que borran la misma fila

Otra caso interesante con operaciones DELETE ocurre cuando las dos transacciones intentan borrar la misma fila, tanto *Oracle* como *PostgreSQL* devuelven los mismos errores *ORA-08177* y *SQL:40001* respectivamente, a pesar de que nuestra intuición nos juegue una mala pasada haciéndonos pensar que quizás intentar borrar una fila ya borrada no debería crear ningún conflicto. Sin embargo, sí que puede que haya un ciclo T1 → T2 → T1, depende del algoritmo que esté implementando esa transacción. Quizás la transacción solo se hace el borrado en el caso de que algunos campos de esa fila tomen unos determinados valores, para lo cual la fila debe consultarse primero, y por tanto debe de existir.

Paso	Transacción T1	Transacción T2 (serializable)

1	`select campo1`
	`from mi_tabla where id=1;`
	
	`/*guardo en una variable X el`
	`valor devuelto por la select*/`
	
	`if x is null then`
	`  delete from mi_tabla where id=1;`
	`end if;`	

2		`select campo1`
		`from mi_tabla where id=1;`
		
		`/*guardo en una variable X el valor`
		`devuelto por la select*/`
		
		`if x is null then`

		`delete from mi_tabla where id=1;` `end if;`
3	`Commit;`	
4		`Commit;`

Ilustración 18. Conflicto de escritura en dos transacciones que borran la misma fila.

Si otra transacción concurrente borra la fila, ya no existiría y ya no se podrían comprobar sus valores para saber si tenía o no que borrarla. El ejemplo de la Ilustración 18 muestra este caso. El grafo de concurrencia, nos dice que hay un arco T1 → T2, pues T1 ha de leer el dato antes de que lo borre/escriba T2, y viceversa; por lo que no admite una planificación serializable.

6.2.3.c Conflicto de escritura entre campos distintos de la misma fila

Vamos a analizar otro caso parecido que quizás tiene una explicación más difícil. Supongamos que la tabla *mi_tabla* ahora es de la siguiente forma:

```
create table mi_tabla (
      id          integer primary key,
      otroCampo1  varchar(30),
      otroCampo2  varchar(30)
);

insert into mi_tabla values ( 1, 'a', 'b');
insert into mi_tabla values ( 2, 'c', 'd');
```

ahora, tenemos 3 campos. En la transacción de la sesión 1 hacemos:

```
SQL> update mi_tabla set otrocampo1='x' where id=1;
```

En la transacción de la sesión 2, que es una sesión SERIALIZABLE hacemos:

```
SQL> update mi_tabla set otrocampo2='y' where id=1;
```

Nota que el campo que estamos intentando cambiar en la segunda transacción no es el mismo que el que estamos intentando cambiar en la primera. Pues bien, si haces el experimento, verás que la transacción de la sesión 2 se queda esperando a que la transacción de la sesión 1 finalice, tal y como ocurriría si el campo que cambiáramos fuese el mismo. Pero ahí no queda la cosa, una vez la transacción 1 hace ***commit***, la transacción 2 nos devuelve el mismo error que en el caso de que el campo actualizado por ambas transacciones fuera el mismo (e.g..; *ORA-08177* en *Oracle,* o *SQL:40001* en *PostgreSQL).*

Esto se puede explica desde dos puntos de vista:

1. Por un lado porque las bases de datos actuales hacen los bloqueos usando **granularidad de fila**. Entonces es normal que al bloquear para escritura la fila, aunque el campo que vamos a cambiar sea otro, nos toque esperar. Es decir, la base de datos no se molesta en mirar que ambos cambios podrían considerarse compatibles a cualquier orden de procesamiento en serie, sino que actúa de manera ciega considerando el granulo de fila el gránulo más pequeño posible sobre el que considerar un conflicto de escritura. Considerar como gránulo mínimo cada campo y gestionar los conflictos y los bloqueos a nivel de campo sería

demasiado costoso, y en la práctica poco demandado por las aplicaciones reales, por lo que ningún SGBD lo hace.

2. Por otro lado, puede que este comportamiento sea del todo conveniente en un tipo de *actualización perdida* bastante frecuente. Supongamos el siguiente ejemplo basado en [Kyte 2010], capítulo 6: Una aplicación de mantenimiento de clientes que maneja el departamento comercial de una empresa, funciona según el siguiente caso de uso, que por otro lado es muy habitual.
   1. En una pantalla inicial el usuario busca por algún criterio el cliente que quiere actualizar. Por ejemplo, a través de palabras claves relacionadas con el nombre del cliente.
   2. La aplicación realiza una SELECT sobre la tabla de *clientes* y devuelve una lista con los posibles clientes que concuerdan con el criterio introducido.
   3. El usuario selecciona un cliente de esa lista y la aplicación le devuelve un formulario con los datos del mismo.
   4. El usuario actualiza el campo o campos oportunos del formulario.
   5. Pulsa el botón de guardar los cambios y se produce un UPDATE de esa fila de la tabla de *clientes*, y el correspondiente COMMIT.

   Supongamos que estamos trabajando en READ_COMMITTED, que es el nivel de aislamiento por defecto de la mayoría de las bases de datos, y más ligero que SERIALIZABLE. Si dos usuarios concurrentes acceden al mismo cliente, el usuario *uno* cambia por ejemplo el teléfono, el usuario *dos* cambia el e-mail; luego ambos usuarios hacen *commit,* pero el usuario *uno* lo hace antes que el *dos.* Tendríamos probablemente una *actualización perdida* del teléfono debido a que la aplicación en el momento de mostrar el formulario guarda en una colección de variables locales los valores actuales de los campos del cliente. Cuando el usuario dos cambia el e-mail, el teléfono queda en su formulario tal y como estaba; y al hacer *commit* se hace un UPDATE de la tabla de *clientes* volcando el contenido de todas las variables del formulario, incluyendo el valor antiguo del teléfono.
   Pero con aislamiento SERIALIZABLE, este ejemplo levantaría *ORA-08177* en el caso de *Oracle,* o *SQL:40001* con *PostgreSQL*, precisamente porque es un caso de actualización de distintos campos de la misma fila.

6.2.3.d Tratamiento de las actualizaciones perdidas con *Read-Committed*

Volviendo al ejemplo de la sección anterior: ¿entonces, es obligatorio utilizar aislamiento serializable en casos como el anterior? El aislamiento serializable consume más recursos que *read-committed*, incluso a veces no está disponible (e.g.; *SQL-Server* trabajando con concurrencia de instantánea). ¿Qué hacer en esos casos? Existen dos aproximaciones para evitar este problema ([Kyte 2010], capítulo 6):

1. Hacer un "***Bloqueo Pesimista***"
2. Hacer un *"**Bloqueo Optimista**"*

Bloqueo Pesimista

En el bloqueo pesimista la transacción bloquea "manualmente" la fila a modificar **antes de** que el usuario la vaya a modificar en la aplicación. En nuestro ejemplo, se trataría de hacer un bloqueo de la fila del cliente que se va a cambiar en el instante previo a editarlo. Por ejemplo, podríamos añadir un botón "*Editar"* al formulario. Tras pulsar ese botón la fila se bloquearía. En *Oracle* y en *PostgreSQL* podríamos hacerlo mediante una SELECT con la clausula ***FOR UPDATE***:

```
SELECT * FROM clientes
WHERE cli_id = variable_cli_di
FOR UPDATE;
```

Esa clausula FOR UPDATE realiza un bloqueo de escritura en la fila seleccionada. Nota que no se bloquea para lectura al resto de usuarios debido al aislamiento de instantánea. El SELECT FOR UPDATE no hace aún ninguna escritura. Para hacer la correspondiente escritura hay que recurrir a una sentencia UPDATE especial, como veremos en próximos temas.

En el ejemplo, se supone que *variable_cli_id* es una variable que contiene el número del cliente que se ha seleccionado de la lista tras aplicar el criterio de búsqueda.

Una vez se ha hecho el bloqueo, este persistirá durante todo el tiempo que el usuario estuviese actualizando los datos, y hasta el momento de pulsar el botón de guardar los cambios (COMMIT) u otro botón de cancelar los cambios (que podría hacer un COMMIT también, ya que si la aplicación aún no ha ejecutado ese UPDATE especial, realmente no se ha cambiado aun nada; y por ello no es necesario retroceder cambios, pero si que es necesario terminar la transacción para que libere el bloqueo).

Volviendo al ejemplo, cuando el usuario *dos* pulse el botón *editar* quedará esperando a que el usuario *uno* termine de editar el formulario (nota que en el fondo estamos forzando un comportamiento 2PL de forma manual). De esta forma, el usuario *dos* verá al cliente con el teléfono nuevo, y la actualización del teléfono ya no se perderá. Sin embargo, plantea la desventaja de tener bloqueada la fila durante un espacio de tiempo indeterminado (imaginemos que el usuario se va a tomar un café mientras está editando la fila).

Nota avanzada: Una posible mejora es utilizar el modificador NOWAIT de la clausula FOR UPDATE:

```
SELECT * FROM clientes
WHERE cli_id = variable_cli_di
FOR UPDATE NOWAIT;
```

que evita que el resto de usuarios que quieran bloquear esa fila esperen a que termine el usuario *uno*. En lugar de esperar recibirían el error ***ORA-00054 resource busy error*** en *Oracle* o ***ERROR: no se pudo bloquear un candado en la fila de la relación «clientes» Estado SQL:55P03*** en *PostgreSQL*. El programa de aplicación trataría esta excepción lanzando un mensaje de error menos agresivo y pidiendo al usuario *dos* que lo re-intente más tarde. Con esta mejora realmente no ganamos nada mas que la aplicación no se queda "colgada" mientras el usuario *uno* vuelve del café; pero seguimos teniendo el problema de que hasta que no vuelva de ese café no podemos actualizar el e-mail de ese cliente.

El bloqueo **pesimista** tiene ese nombre porque hemos programado la transacción **pensando en el peor escenario**, que varios usuarios concurrentes actualicen a la vez la misma fila. Nota que requiere que la sesión, y por tanto la conexión, con la base de datos se mantenga durante toda la operativa, desde que se muestra el formulario hasta el momento de cometer los cambios, lo cual no es factible, por ejemplo, en una aplicación web. Por ello, el bloqueo pesimista está cada vez más en desuso, es típico de las aplicaciones cliente-servidor de los años 90, donde típicamente había como máximo unas decenas de usuarios concurrentes cada uno con una conexión estable a la base de datos durante toda la sesión de funcionamiento de la aplicación.

Bloqueo Optimista

La idea de esta técnica es diferir el bloqueo al momento justo en el que las modificaciones vayan a ser llevadas a cabo en la base de datos. En nuestro ejemplo, se trataría de hacer el bloqueo justo nada más pulsar el botón de guardar los cambios (en el momento del *commit*), por lo que no es necesario utilizar SELECT … FOR UPDATE, ya que el correspondiente UPDATE realizaría el bloqueo:

```
UPDATE clientes
SET nombre = variable_nombre, direccion =
variable_direccion,
tfno = variable_tfno,          mail = variable_mail
WHERE cli_id = variable_cli_di;
```

El tiempo que la fila está bloqueada, ahora es instantáneo; no depende de cuanto dure la interacción del usuario. Sin embargo, si el usuario *dos* cambia el e-mail del cliente, y guarda los cambios podría nuevamente haber una actualización perdida que anulara el cambio del teléfono hecho por el usuario *uno.* De ahí la denominación **optimista,** pues la transacción se programa pensando que es **improbable** que dos usuarios modifiquen a la vez la misma fila de la misma tabla.

Para evitar este problema, una vez leídos en variables los valores que se muestran en el formulario, podríamos guardarlos en un juego de variables "*old_"*, mientras que los valores que edita el usuario quedan en variables "*new_*", y de esta forma solo actualizar si ningún valor antiguo ha cambiado. Por ejemplo:

```
UPDATE clientes
SET nombre = new_nombre,  direccion = new_direccion,
tfno = new_tfno,          mail = new_mail
WHERE cli_id = variable_cli_di /*El cli_id no se puede cambiar*/
AND nombre = old_nombre OR (nombre IS NULL AND old_nombre IS NULL))
AND direccion = old_direccion
                  OR (direccion IS NULL AND old_direccion IS NULL))
AND (tfno = old_tfno OR (tfono IS NULL AND old_tfno IS NULL))
AND (email = old_email OR (email IS NULL AND old_email IS NULL));
```

Existen variantes de este mismo esquema añadiendo a la tabla un campo adicional ([Kyte 2010] capítulo 6, [Silberschatz et al. 2015] sección 15.9.3) que puede contener un *timestamp* de la última modificación o un código *hash* de la concatenación de los campos de la fila o un número de versión incremental. En estas aproximaciones ese campo nos da una huella de si otra transacción ha cometido un cambio sobre la fila, y evitaría comprobar en el WHERE del UPDATE, campo a campo, si alguno ha cambiado.

En este caso, cuando el usuario *dos* fuese a cambiar el e-mail, el cambio de teléfono se supone ya cometido por el usuario uno, ya que el cambio es instantáneo, y en todo caso el usuario *uno* esperaría unos imperceptibles milisegundos el bloqueo que causa el UPDATE del usuario *uno.*

Como el cambio del usuario *uno* está ya cometido, la condición *"AND* `tfno = old_tfno`" no se cumpliría en la sesión del usuario *dos* que, como veremos en próximos temas, podría preguntar fácilmente el número de filas actualizadas por ese UPDATE, y en caso de que fuese cero lanzar un mensaje de error pidiendo al usuario que repita todo el proceso de actualización (lo cual enojaría al usuario) o bien que la propia aplicación reintente un nuevo UPDATE combinando los valores actualizados por otros usuarios concurrentes mientras no entren en conflicto con el cambio del e-mail del usuario *dos* (lo que implica volver a consultar todos los valores del cliente a modificar).

Observa que el bloqueo optimista no requiere de estar permanentemente conectado a la base de datos. Basta con que el usuario se conecte fugazmente en el momento de adquirir los datos del formulario y otra vez en el momento de cometer los cambios (2 sesiones muy cortas en el tiempo que implementan 2 transacciones). Esto permite que el servidor aguante más usuarios concurrentes[12], y por otro lado es un esquema que se adapta mucho mejor a las aplicaciones web actuales. Esta estrategia de bloqueo optimista será en general la que seguiremos a lo largo del curso.

6.2.4 INSERTs en el aislamiento de instantánea

Al contrario de 2PL, en el aislamiento de instantánea no es necesario bloquear para escritura las filas mientras se insertan por una transacción. Esto es debido a que cuando insertamos una fila, solo es visible a la instantánea

12 En próximos temas veremos el concepto de *Pool de Conexiones* que es una especie de *array* de conexiones que comparten múltiples transacciones, y que sirven perfectamente a este escenario de bloqueo optimista.

correspondiente a esa transacción, por lo que el resto de transacciones no pueden verla, y por tanto no pueden obtenerla en una SELECT (sería una lectura sucia) hasta que no sea cometida. Incluso si el INSERT es cometido, el resto de transacciones concurrentes serializables, tampoco verán esa fila en su instantánea cuando hagan una SELECT (fenómeno fantasma). Al no verla tampoco hay posibilidad de modificarla/borrarla (no hay conflictos de escritura).

Supongamos una tabla de *actores*:

```
create table actores(
  id char(2) primary key,
  nombre varchar(15)
);
```

A continuación se lanzan 2 transacciones en paralelo (ver Ilustración 19), con aislamiento *read committed,* pero puedes comprobar experimentalmente por ti mismo que el nivel de aislamiento no influye en el resultado del experimento. En una sesión se inserta un actor A5 con nombre XX, y desde otra sesión concurrente otra transacción intenta cambiar los actores con nombre XX dándoles el nuevo nombre ZZ.

Naturalmente si la transacción de la sesión 2 logra modificar el nombre de A5 sería porque es capaz de ver que la fila recién insertada cumple la condición WHERE del UPDATE incurriendo en una lectura sucia. Luego en ningún caso debe de producirse el UPDATE mientras la transacción en la sesión 1 no sea cometida. Precisamente, en 2PL (ver sección 5.1.3), la fila insertada era bloqueada para escritura y la sesión 2 tenía que esperar al *commit* de la sesión 1. Sin embargo, en *Oracle* y *PostgreSQL* no se produce ninguna espera en la sesión 2, y el efecto cuando ambas transacciones hacen *commit*, es que el actor A5 permanece con el nombre XX y que el UPDATE ha actualizado cero filas. Esto es porque la fila recién insertada por la sesión 1 no es visible por la sesión 2, y por tanto la sesión 1 no necesita bloquearla.

Paso	Sesión 1	Sesión 2
1	`insert into actores values ('A5','XX');`	

2		`update actores`
		`set nombre='ZZ'`
		`where nombre = 'XX';`

3	`Commit;`	
4		`Commit;`

Ilustración 19: Ejemplo que muestra que en el aislamiento de instantánea no hay bloqueo de filas insertadas.

Desde el punto de vista de la *serializabilidad en cuanto a conflictos* el grafo de precedencia nos muestra un arco T1 → T2, porque hay un conflicto de escritura por el que la sesión 2 necesita que la sesión 1 inserte primero la fila. Por tanto, si el aislamiento exigido hubiera sido serializable es "argumentable" decir que se ha generado una planificación equivalente a la planificación serie (T2, T1), pero no se ha generado una planificación equivalente a la única planificación serie en cuanto conflictos posible, que es (T1, T2), que debiera ser lo esperable. En la sección 6.3 veremos más anomalías que ponen en entredicho la capacidad del aislamiento de instantánea para cubrir cualquier escenario serializable.

6.3. El aislamiento de instantánea por si solo no asegura la serializabilidad

Existen varios escenarios excepcionales que el aislamiento de instantánea no es capaz de emular un comportamiento serializable al cien por cien ([Silberschatz et al. 2015] sección 15.7.2):

Caso 1, "Atasco de Escritura" ([Silberschatz et al. 2015] sección 15.7.2): Dos transacciones concurrentes leen dos filas y cada una escribe en una de ellas. En el ejemplo de la Ilustración 20 vemos que cada transacción bloquea para escritura filas distintas, por lo que no hay ningún problema en que se ejecuten concurrentemente. Si obtenemos el

grafo para verificar la *serializabilidad en cuanto a conflictos*, vemos que:

- Hay un arco T1 → T2 :

  T1 ha de ejecutarse antes que T2 porque la SELECT de T1 lee datos anteriores al UPDATE de T2, y
- Hay un arco T2 → T1:

  T2 ha de ejecutarse antes que T1 porque la SELECT de T2 lee datos anteriores al UPDATE de T1

Es decir, hay un ciclo. Por tanto no son transacciones que ejecutándose intercalando las operaciones en el orden que indica la ilustración admitan una planificación equivalente serializable en cuanto conflictos.

Ahora imagina que la tabla de *cuentas* contiene en cada fila información de una cuenta corriente. Supón que las cuentas 1 y 2 son del mismo cliente. Supón además que antes del UPDATE una sentencia IF verifica que las suma del saldo de las cuentas es suficiente como para extraer los 200 euros de cada UPDATE. Sea 100 euros el saldo de la cuenta 1 y 200 el de la cuenta 2. Entonces al ejecutarlas en serie, por ejemplo T1 antes que T2, al llegar a T2 ese IF se hubiera percatado de que T2 no puede llevarse a cabo porque no hay saldo suficiente, seguramente retrocediendo la transacción y lanzando una excepción. Sin embargo, un aislamiento de instantánea no se hubiera percatado del problema y hubiera permitido ambas transacciones.

Paso	**Transacción T1 serializable**	**Transacción T2 serializable**
1	`select * from cuentas where cta_id = 1 or cta_id=2;`	
2		`select * from cuentas where cta_id = 1 or cta_id=2;`
3	`update cuentas set saldo=saldo-200 where cta_id=1;`	
4		`update cuentas set saldo=saldo-200 where cta_id=2;`
5	`commit;`	
6		`commit;`

Ilustración 20: Debilidades del aislamiento de instantánea: Atasco de Escritura.

La importancia viene precisamente del IF que provee el mantenimiento de una restricción y en última instancia de la consistencia. Si no importa que el cliente se quede con un saldo negativo, esta debilidad del aislamiento de instantánea es permisible, pero en otros casos como en el que no se permita un saldo negativo, no lo es.

Se puede encontrar otro ejemplo en la misma línea en [Ports & Grittner 2012] sección 2.1.1.

Paso	**Transacción T1 serializable**	**Transacción T2 serializable**	**Transacción T3 serializable**
1	`select * from cuentas where cta_id=1;`		
2		`select * from cuentas where cta_id = 1 or cta_id=2;`	
3	`update cuentas set saldo=saldo+200 where cta_id=1;`		
4		`update cuentas set saldo=saldo-200`	

		`where cta_id=2;`	
5	`commit;`		
6			`select * from cuentas where cta_id = 1 or cta_id=2;`
7		`commit;`	

Ilustración 21: Debilidades del aislamiento de instantánea. Dos transacciones de actualización y una de lectura.

Caso 2 ([Silberschatz et al. 2015] sección 15.7.2): En este caso tenemos tres transacciones. Las dos primeras son actualizaciones, y la tercera es de solo lectura (ver Ilustración 21). Basándonos en el mismo ejemplo del "caso 1", la transacción T1 hace un ingreso de 200 euros en la cuenta 1, como había ya 100 euros, en la cuenta 1 quedan 300. La transacción T2 es la misma que en el caso 1, consulta el saldo en todas las cuentas y si es suficiente, permite que se extraigan los 200 euros. Como al principio del ejemplo la cuenta 1 tenía 100 euros, y la cuenta 2 tenía 200 euros, el saldo es suficiente, y la actualización provocará que la cuenta 2 se quede con 0 euros. La transacción T3 consulta el saldo de las cuentas del cliente a las 12 de la noche de final de mes, quizás porque si tiene un saldo superior a 400 euros entre ambas cuentas le regala una vajilla o no le penaliza en las condiciones de un préstamo, etc …
Claramente las transacciones T1 y T2 bloquean filas distintas, y nuevamente no hay ningún problema para que las tres transacciones se ejecuten concurrentemente. T3 mostraría la actualización cometida por T1, la cuenta 1 tendría un saldo de 300 euros, que con los 200 en la cuenta T2 supone un total de 500 y el cliente tendría su vajilla. ¿Realmente el cliente tiene 500 euros a las 12 de la noche de final de mes? … Seguramente el banco piense que no los debería de tener, especialmente si en lugar de una cazuela lo que está en juego es penalizar su préstamo.
Si analizamos el grafo resultante de T1 con T2, la planificación es serializable en cuanto a conflictos, ya que:

- No hay un arco T1 → T2:
  T1 NO ha de ejecutarse antes que T2 porque la SELECT de T1 lee solo la fila 2, NO lee datos anteriores al UPDATE de T2, y
- Hay un arco T2 → T1 :
  T2 ha de ejecutarse antes que T1 porque la SELECT de T2 lee datos anteriores al UPDATE de T1

Sin embargo al añadir T3 aparecen nuevos arcos en el grafo:

- Hay un arco T3 → T2 :
  T3 ha de ejecutarse antes que T2 porque la SELECT de T3 lee datos anteriores al UPDATE de T2
- Hay un arco T1 → T3 :
  T1 ha de ejecutarse antes que T3 porque la SELECT de T3 lee datos posteriores al UPDATE de T1

Por lo que se genera el ciclo T2 → T1 → T3 → T2, y la planificación concurrente de las tres transacciones intercaladas en ese orden deja de ser serializable en cuanto a conflictos.
Hemos visto que una planificación T1, T3, T2; que es la que se obtiene del aislamiento de instantánea, provoca que el cliente no sea penalizado en su préstamo, pero claramente, una planificación T2, T1, T3 (la que hubiera hecho 2PL) le hubiera dejado sin premio. Sin embargo, si el interés de T3 hubiera sido regalarle una cazuela o un cenicero, quizás que la planificación fuese T1, T3, T2 (o la T2, T1, T3) hubiera provocado inconsistencias también, pero en este caso sin mayores consecuencias probablemente.
Caso 3: ([Ports & Grittner 2012] sección 2.1.2.). Es en realidad otro ejemplo del caso 2 como explicaremos más adelante. Sea una aplicación que emite remesas de recibos, tiene 3 transacciones:

1. INSERTAR-RECIBO:
   1. Lee de la tabla de remesas cuál es la remesa actual (la tabla de remesas tiene una única fila con el número de remesa actual).

2. Inserta un recibo en la tabla de recibos con el número de remesa leído en el paso anterior.

2. CERRAR-REMESA:
   1. Incrementa en uno el número de remesa una unidad en la tabla de remesas
3. INFORME-ULTIMA-REMESA-CERRADA:
   1. Lee el número de remesa actual de la tabla de remesas.
   2. Lee la suma del importe de todos los recibos cuyo número de remesa, sea igual al número de remesa leído "menos uno".

Si disponemos la ejecución concurrente de las tres transacciones como en la Ilustración 22, vemos que el recibo que inserta T2 en la remesa "x", antes de que T3 cierre la remesa "x", no es computado por T1 que reporta el total de esa remesa "x". El grafo correspondiente a este ejemplo tiene los arcos:

- T2 → T3 porque T2 ha de leer la remesa antes de que T3 la cambie
- T3 → T1 porque T1 ha de leer la remesa después de que T3 la cambie
- T1 → T2 porque T1 ha de leer los recibos antes de que T2 lo inserte

Así se genera el ciclo T2 → T3 → T1 → T2, por lo que nuevamente la planificación concurrente de las tres transacciones intercaladas en ese orden deja de ser serializable en cuanto a conflictos.

Paso	**Transacción T1 serializable** INFORME-ULTIMA-REMESA-CERRADA	**Transacción T2 serializable** INSERTAR-RECIBO	**Transacción T3 serializable** CERRAR-REMESA

1		`SELECT nroRemesa`
		`INTO x`
		`FROM remesas;`	

2		`/*`	`UPDATE remesas`
			`SER nroRemesa = nroRemesa+1;`
			
			`COMMIT;`

3	`SELECT nroRemesa`
	`INTO x`
	`FROM remesas;`
	
	`SELECT SUM(importe)`
	`FROM recibos`
	`WHERE nroRemesa=x-1;`
	
	`COMMIT;`
		Varios cálculos para estimar el importe y guardarlo en una variable
		`v_importe`
		
		`*/`	

4		`INSERT INTO recibos`
		`( nroremesa, importe)`
		`VALUES`
		`( X, v_importe);`
		
		`COMMIT;`	

Ilustración 22: Debilidades del aislamiento de instantánea. [Ports & Grittner 2012] Sección 2.1.2. (SELECT INTO en Oracle PL/SQL permite almacenar el resultado de una SELECT que devuelve una sola fila en una variable).

Comentario: El caso 2 y el 3 son prácticamente el mismo (T1 del caso 3 hace el papel de T3 en el caso 2, T2 en el caso 3 hace el mismo papel de T2 en el caso 2, y T3 del caso 3 hace el papel de T1 en el caso 2). Hay una diferencia muy sutil entre ambos casos que consiste en que, en el caso 3, el INSERT supone una escritura de un dato que ni T2 ni T3 pueden leer porque aún no existe al comenzar ambas transacciones.

Caso 4 ([Kyte 2010], capítulo 7, sección *Lost Updates: Another Portability Issue*).

Paso	Transacción T1 serializable	Transacción T2 serializable

1	`INSERT INTO A`
	`  SELECT COUNT(*)`
	`  FROM B;`	

2		`INSERT INTO B`
		`  SELECT COUNT(*)`
		`  FROM A;`

3	`Commit;`	
4		`Commit;`

Ilustración 23: Debilidades del aislamiento de instantánea. ([Kyte 2010], capítulo 7, sección Lost Updates: Another Portability Issue)

Sean dos tablas vacías A y B:

```
create table a ( x integer primary key );
create table b ( x integer primary key );
```

Si ejecutamos las transacciones concurrentes de la Ilustración 23, es claro que con el aislamiento serializable de instantánea, tras los *commits,* ambas transacciones tienen una fila con el valor cero. Sin embargo, si se ejecutaran en serie una de las dos tablas tendría un cero y otra un uno, lo cual nos hace vislumbrar que el comportamiento no es totalmente serializable.

Si obtenemos el grafo de precedencia, hay un arco T1 → T2, debido a que T1 ha de ejecutarse antes que T2 para que cuente las cero filas que hay antes de T2. De manera similar hay un arco T2 → T1 porque T2 necesita ejecutarse antes de que T1 haga la inserción, y por lo tanto se produce un ciclo que nos indica que no hay una planificación serializable en cuanto a conflictos equivalente.□

En cuanto a los SGBD que implementan el aislamiento serializable mediante instantáneas, *Oracle* no detecta ningún conflicto en estos casos. Por el contrario, *PostgreSQL* si que los detecta y emite un mensaje de error debido a que hace un chequeo de posibles conflictos entre versiones basado en grafos. Lo hace en el momento del *commit* de cada transacción [Ports & Grittner 2012], pues en ese momento se sabe qué operaciones de lectura y escritura ha hecho dicha transacción,. En los casos 1, 2, 3 y 4, al hacer *commit* de T2 *PostgreSQL* detecta el problema y devuelve el error:

> *ERROR: no se pudo serializar el acceso debido a dependencias read/write entre transacciones. Estado SQL:40001. Detalle:Reason code: Canceled on identification as a pivot, during commit attempt. Sugerencias:La transacción podría tener éxito si es reintentada.*"

La solución para *Oracle* es prevenirlo con la clausula FOR UPDATE del comando SELECT. "*SELECT FOR UPDATE*" para provocar un bloqueo de escritura sobre las filas a las que accede la SELECT, sea cual sea el nivel de aislamiento, tal y como se explicó en la sección "Bloqueo Pesimista".

En el caso 1, hubiera bastado con forzar un bloqueo de escritura con SELECT FOR UPDATE en la SELECT que lee los saldos:

```
select * from cuentas
where cta_id = 1 or cta_id=2
FOR UPDATE;
```

T1 hubiera ganado los bloqueos y T2 hubiera tenido que esperar a que acabase T1. De esta manera en el fondo estamos consiguiendo emular el mismo comportamiento que con 2PL. En el caso 2 hubiera bastado con añadir el SELECT FOR UPDATE a T3 o alternativamente a T2, de manera que se hubiera quedando esperando a que el cambio de T2 se hubiera cometido. El SELECT FOR UPDATE del caso 3 sería en la SELECT de T2 o bien la primera SELECT de T1, por ejemplo. El caso 4 no tiene solución con SELECT FOR UPDATE.

6.4. Liberación de las versiones obsoletas

Finalmente, en los protocolos de concurrencia multiversión (y por lo tanto en el de instantánea también), no es necesario mantener para siempre todas las versiones de un dato, sino que se puede hacer una limpieza de versiones obsoletas. Una versión está obsoleta en un instante *t* si:

1. Corresponde a una transacción **retrocedida,** o
2. **No** siendo la versión **más reciente y** correspondiendo a una transacción **cometida** T**,** la cual terminó antes de que comenzara la transacción activa que lleve más tiempo ejecutándose.
   Es decir, si ts_fin(T) < ts_inicio(T_oldest); donde T_oldest es el ts_inicio de la transacción más antigua de entre las que no hayan terminado en ese instante *t* (i.e.; de entre las que teniendo un estado "activa" tiene un valor de ts_inicio mínimo) .

Para ver con cierta profundidad cómo implementan la limpieza de versiones obsoletas los SGBD considerados en este tema , se recomienda la lectura del anexo 3.

7. OLTP vs. OLAP

Una base de datos trabaja como OLTP (*On-Line Transaction Processing*) cuando está haciendo constantes actualizaciones, normalmente concurrentes, de la base de datos mediante transacciones. Es el caso de cualquier base de datos con la que trabaje de cualquier aplicación que está haciendo altas, bajas y modificaciones. Por el contrario, un SGBD trabaja como OLAP (*On-Line Analytical Processing*), cuando principalmente se dedica a hacer grandes consultas con fines normalmente orientados a la minería de datos y a la inteligencia empresarial.
Si nuestra base de datos necesita compaginar ambas funcionalidades, hay que crear dos instancias de bases de datos, una para OLTP, y otra para OLAP; esta última se conoce normalmente como ***datawarehouse***, y toma los datos que vuelca en ella la instancia OTLP periódicamente (e.g; diariamente, semanalmente, etc …). Es habitual que este proceso de volcado haga algunas transformaciones en los datos (e.g.; típicamente al *datawarehouse* van a parar datos agregados, como por ejemplo los que se obtienen de consultas con GROUP BY: sumatorios, medias, recuentos etc…).
Si utilizáramos la misma instancia de la base de datos para trabajar simultáneamente como OLTP y OLAP de forma intensiva, el sistema daría problemas de rendimiento:

- Por un lado, es claro que si el SGBD implementa el control de concurrencia mediante bloqueos (i.e.; 2PL), se producirían un sin fin de esperas entre los bloqueos de lectura que necesitan las aplicaciones OLAP y los de escritura que necesitan las aplicaciones OTLP.
- Por otro lado, si el SGBD implementa el control de concurrencia mediante instantáneas, si que es verdad que se reducen los bloqueos y las consultas OLAP no entorpecen con sus bloqueos de lectura a las transacciones OLTP; ni las transacciones OLTP entorpecen a las consultas OLAP con sus bloqueos de escritura. Sin embargo, las consultas OLAP sobre grandes bases de datos es normal que tengan tiempos de respuesta de varios minutos. Durante ese tiempo pueden haber ocurrido cientos o miles de transacciones OLTP, y por ello pueden haberse generado gran cantidad de versiones de filas. Esto tiene dos problemas:
  1. Por un lado reconstruir la instantánea para esa consulta OLAP es muy costoso, tendrá que leer quizás cientos de bloques de disco donde estén las versiones originales de las filas modificadas.
  2. Al final de la sección 6.2 decíamos que una versión cometida se consideraba obsoleta en un instante *t* si no siendo la versión cometida más reciente, fue cometida antes de que empezara la transacción activa más antigua. Esto supone que, por ejemplo, si una consulta tuviera un tiempo de respuesta de 20 minutos, normalmente habría que guardar todas las versiones originales de datos modificados que se

han generado durante los últimos 20 minutos; y eso puede suponer gran cantidad de espacio en disco.

Caso de estudio: *Oracle snapshot too old*

En este contexto merece la pena comentar el caso concreto de *Oracle*. *Oracle* guarda las versiones originales de los datos modificados en una zona especial llamada segmento UNDO (ver anexos 4.3 y 3.1). Cuando comienza una transacción, a ésta se le asigna un primer bloque en el segmento UNDO (si fuese una transacción muy larga podría necesitar más bloques que se irían agregando a modo de lista enlazada). En ese instante en el que se asigna el bloque UNDO, *Oracle* tiene que buscar uno que no haya sido utilizado, pero lo normal, es que salvo en los momentos iniciales al arranque de la base de datos, llegue un momento en que todos los bloques UNDO hayan sido utilizados en algún momento por alguna transacción. *Oracle* es capaz de hacer crecer el segmento UNDO si fuera necesario (ver anexo 4.3), pero para evitar que crezca indefinidamente tiene un parámetro UNDO_RETENTION que por defecto tiene el valor 900, que es el número de segundos que hay en 15 minutos ([Lewis 2011] Capítulo 3, sección *Transaction Table Rollback*). Pasado ese tiempo, un bloque del segmento UNDO que solo contenga información de transacciones finalizadas puede ser reutilizado para ser asignado a una nueva transacción. *Oracle* en ese caso "sacrificará" los bloques correspondientes a transacciones finalizadas más viejas.

Supongamos una transacción T1 que tiene una SELECT que tarda un tiempo superior al UNDO_RETENTION en ejecutarse. Sea cual sea su nivel de aislamiento, debe de recuperar la información tal y como estaba en el instante ***t*** en el que comenzó la SELECT (i.e., aunque esté en READ COMMITTED no debería de poder ver los cambios cometidos durante la ejecución de la SELECT).

Naturalmente, las transacciones concurrentes que están modificando datos mientras se ejecuta la SELECT estarán demandando bloques UNDO, y no es improbable que a algunas de ellas se les haya asignado un bloque UNDO que necesita la propia SELECT para reconstruir la información consultada tal y como estaba en ese instante ***t***. La consecuencia es que *Oracle* lanza el error "***ORA-01555 snapshot too old***", indicando que es incapaz de reconstruir la información que necesita la SELECT ([Lewis 2011] Capítulo 3, sección *Transaction Table Rollback*).

En definitiva, este caso ilustra que ni si quiera un SGBD tan reputado como Oracle está pensado para que conviva un modo de uso OLTP de la base de datos con un modo de uso OLAP.

8. Esperas por posible violación de restricciones SQL

Supongamos que dos transacciones intentan insertar una misma fila con el mismo valor en la clave primaria. Este caso provoca una espera y a veces una excepción. Este comportamiento es independiente del nivel aislamiento elegido (i.e., READ COMMITTED vs. SERIALIZABLE) y también de si la concurrencia se implementa mediante bloqueos o mediante instantánea.

Supongamos la tabla *mi_tabla* que hemos utilizado en ocasiones anteriores:

```
CREATE TABLE mi_tabla (
      id        integer constraint pk_mi_tabla primary key,
      conn      char(6) );
```

supongamos también, que aún no existe ninguna fila con *id=100*; por lo cual sería lícito poder insertar una fila con ese *id.* Abrimos 2 sesiones concurrentes , por ejemplo, de *SQL*Plus* y desde cada una de las 2 sesiones intentamos hacer esa inserción. En la sesión 1 la inserción se realiza sin problemas, pero la segunda se queda esperando a que acabe la transacción de la sesión 1:

SESION 1

```
SQL> insert into mi_tabla values (100,'con1');
1 row created
SQL>
```

SESION 2

```
SQL> insert into mi_tabla values (100,'con2');
```

La razón de esa espera es que no se sabe si se va a producir una violación de clave primaria o no, depende de como acabe la transacción 1.

Podrían pasar 2 cosas:

1. Que la transacción 1 acabe con *commit.* En ese caso la inserción de la sesión 2 no se ejecutaría y se lanzaría una excepción:

SESION 2

```
SQL> insert into mi_tabla values (100,'con2');
insert into mi_tabla values (100, 'con2')
*
ERROR at line 1:
ORA-00001: unique constraint (HR.pk_mi_tabla) violated
```

2. Que la transacción 1 acabe con *rollback*. En ese caso la inserción de la sesió 2 se ejecutaría normalmente.

Este tipo de comportamiento se da igualmente:

1. En el caso de haber sido una restricción UNIQUE, no tiene que ser forzosamente solo el caso de la clave primaria.
2. También con las claves ajenas, cualquier caso en el que la violación de la clave ajena no se puede dirimir hasta que no acabe una de las transacciones. He aquí algunos ejemplos, pero hay más casos. Supongamos que la tabla que denominamos "hija" mantiene una clave ajena referenciando a la clave primaria de otra tabla que llamaremos "padre":
    1. Si una transacción borra la fila padre en la que la clave primaria vale 100 y la otra inserta una fila hija de la 100; la inserción se queda a la espera de ver si el borrado de la fila padre es cometida o retrocedida.
    2. Si una transacción borra la última fila hija del padre con clave primaria igual a 100 y la otra intenta borrar esa fila padre; la transacción que borra la fila padre se queda esperando la confirmación de que a la fila padre ya no le quedan filas hijas.

        Nota: Esto último los SGBD testados lo hacen incluso cuando la clave ajena está definida con ON DELETE CASCADE, lo cual a priori podría parecer poco lógico.
    3. Si una transacción cambia el valor de la clave primaria de la fila de la tabla padre, por ejemplo de 100 a 101, supuesto no tenga filas hijas asociadas a 100, y la otra transacción intenta insertar una fila hija con clave ajena igual a 100 (o a 101, pues ambos casos dan lugar a espera por parte de la inserción).

Buena parte de los escenarios en los que se producen esperas por una posible violación de una restricción tienen que ver con que sea el usuario el que dé valor a las claves primarias, y en consecuencia a las claves ajenas que referencian a esas claves primarias. Una forma de evitar estas esperas es utilizar valores automáticos para las claves primarias. En el próximo capítulo veremos como dar valores automáticos a través de las secuencias.

Las claves primarias, UNIQUEs y claves ajenas se implementan, de forma transparente al usuario, indexando los campos que las forman para así acelerar la búsqueda de, por ejemplo, valores repetidos, o filas hijas potencialmente "huérfanas". Es por tanto lógico que la implementación interna de estas esperas por posible violación de restricciones estén basadas en técnicas de bloqueo de índice como las que se expusieron en la subsección "Operaciones de escritura" de la sección 5.1.3.a, donde, para el caso de un control de concurrencia basado en bloqueos, ya se menciona que en el caso del INSERT se bloquean las hojas del índice correspondientes a los nuevos valores insertados.

Por tanto, desde una transacción T:

- Para mantener la unicidad en claves primarias o UNIQUEs:
  - Ante una operación INSERT en T basta bloquear la hoja del índice correspondiente al valor que se va a insertar para esa clave primaria o UNIQUE, evitando que otra transacción concurrente use para otra fila ese mismo valor, lo cual no es posible si T acabase con *commit*.
  - Ante una operación UPDATE en T que modifique esa clave primaria o UNIQUE:
    - Hay que bloquear igualmente la hoja correspondiente al valor nuevo para esa clave primaria o UNIQUE, evitando que otra transacción concurrente T' use para otra fila ese mismo valor, lo cual no es posible si T acabase con *commit*.
    - Pero también hay que bloquear la hoja correspondiente al valor original para esa clave primaria o UNIQUE, evitando que otra transacción concurrente T' use para otra fila ese mismo valor, lo cual no es posible si T acabase con *rollback*.
- Para mantener la integridad referencial:
  - Cuando se están haciendo operaciones con las filas hijas hay que bloquear la hoja del índice de la clave primaria de la fila padre para prevenir que una transacción concurrente elimine o cambie la identidad (i.e.; el valor de la clave primaria) del padre.
  - Cuando se están haciendo operaciones con una fila de la tabla padre hay que bloquear los valores de la hoja de índice de la clave ajena de sus filas hijas para prevenir que una transacción concurrente añada hijos a un padre que está siendo eliminando o cambiando su identidad (i.e.; cambiando el valor de la clave primaria)

  La Tabla 5 muestra los bloqueos que se han detectado experimentalmente en *Oracle, PostgreSQL* y *SQL-Server* y con qué bloqueo de índice se podría implementar cada caso.

Operación	**¿Dónde se produce bloqueo?**	**Finalidad**
INSERT en tabla hija desde una transacción T	Hoja de la PK de la tabla padre	Impedir que ese padre sea borrado o cambie de valor su PK en una transacción concurrente T' si T es cometida. En ese caso, T' devolvería error cuando T acabe con *commit*
INSERT en tabla padre desde una transacción T	Hoja del índice de la FK de los hijos	Impedir que desde otra transacción concurrente T' se le añada un hijo a ese padre[13] (con INSERT/UPDATE) si T retrocediese. En ese caso, T' devolvería error cuando T acabe con *rollback.*
`UPDATE hijo SET FK = V2 WHERE FK = V1;` (Ver nota [14])	Hoja del índice de la PK del padre correspondiente a V2	Impedir que ese futuro padre con PK=V2 sea borrado o cambie de valor su PK en una transacción concurrente T' si T es cometida. En ese caso, T' devolvería error cuando T acabe con *commit.*

13 En *PostgreSQL*, sin embargo, no se produce el bloqueo y directamente da error de violación de clave ajena al añadir el hijo.

14 Si en lugar de hacer UPDATE de todos los hijos de V1, se hiciese UPDATE de algunos de ellos, se observa que el SGBD no comprueba que no se actualizan todos los hijos, sino que bloquea de manera mecánica al padre aunque

Operación	**¿Dónde se produce bloqueo?**	**Finalidad**
	Hoja del índice de la PK del padre correspondiente a V1	Impedir que ese ese antiguo padre con PK=V1 sea borrado o cambie de valor su PK[15] en una transacción concurrente T', si T retrocediese. En ese caso, T' devolvería error cuando T acabe con *rollback.*
`UPDATE padre SET PK = V2 WHERE PK = V1;`	Hoja del índice de la FK de los hijos correspondiente a V2	Impedir que desde otra transacción concurrente T' se le añada un hijo con FK=V2 a ese padre[13] (con INSERT/UPDATE) si T retrocediese. En ese caso, T' devolvería error cuando T acabe con *rollback.*
	Hoja del índice de la FK de los hijos correspondiente a V1[x]	Impedir que desde otra transacción concurrente T' se le añada un hijo con FK=V1 a ese padre[13] (con INSERT/UPDATE) si T retrocediese. En ese caso, T' devolvería error cuando T acabe con *commit.*
DELETE de todos los hijos desde una transacción T[16]	Hoja del índice de la PK del padre	Impedir que si ese padre es borrado o cambie de valor, su PK en una transacción concurrente T', T devuelva un error cuando acabe con *rollback.*
DELETE en tabla padre desde una transacción T	Hoja del índice de la FK de los hijos	Impedir que desde otra transacción concurrente T' se le añada un hijo (con INSERT/UPDATE) cuando T acabe con *commit.*

Tabla 5. Posibles acciones de bloqueo de índice para prevenir violaciones de integridad referencial (PK=Clave Primaria, FK=Clave Ajena, T = transacción que realiza la primeramente la operación en la primera columna de la tabla, T' = transacción concurrente que acto seguido realiza otra operación que amenaza la integridad referencial). Las excepciones a esta tabla que se han detectado experimentalmente aparecen en las notas al pie.

IV. Resumen

Las transacciones son el mecanismo que permite agrupar cambios que han de ejecutarse conjuntamente para que la base de datos refleje siempre un estado consistente. Las transacciones además son segmentos de tiempo en los que cada sesión concurrente se comporta como si estuviera aislada de las demás, de manera que el estado de la base de datos que percibe cada sesión fuera el mismo que si dicha sesión fuese la única que accediese a la base de datos en ese momento.

hubiera más hijos. Es más, bloquea al padre incluso si hay una declaración ON UPDATE CASCADE, aunque en ese caso, cambiar el valor original de la clave primaria del padre desde T' no da error al hacer *rollback* T (por el ON UPDATE CASCADE).

15 En *PostgreSQL,* sin embargo, no se produce el bloqueo y directamente da error de violación de clave ajena al intentar borrar o cambiar el valor de la clave primaria del padre.

16 Si en lugar de hacer DELETE de todos los hijos, se hiciese DELETE de algunos de ellos, se observa que el SGBD no comprueba que se estén borrando todos los hijos, sino que bloquea de manera mecánica al padre aunque hubiera más hijos. Es más, bloquea al padre incluso si hay una declaración ON DELETE CASCADE, aunque en ese caso, borrar el padre no da error al hacer *rollback* T (por el ON DELETE CASCADE).

Para crear esa ilusión de aislamiento, la base de datos se basa en mecanismos que consumen recursos de tiempo (bloqueos) y espacio de almacenamiento (sistemas de concurrencia multiversión), por lo que hay que tener en cuenta si merece la pena relajar esa exigencia de máximo aislamiento (aislamiento SERIALIZABLE) a otro nivel que consuma menos recursos, como es el aislamiento READ COMMITTED, que al final es el que usan por defecto la mayoría de sistemas.

V. Glosario

ACID: acrónimo que resume las propiedades que deben de cumplir las transacciones en un SGBD, por la cual han de ser **A**tómicas, **C**onsistentes, **I**(*solation*) = dar una visión que aisle cada sesión de las demás y ***D(****urability)* persistentes.

Aislamiento de instantánea: técnica de implementación del aislamiento que permite que las transacciones puedan reconstruir un estado (una instantánea en el sentido de instantánea fotográfica) anterior de la base de datos, de manera que pueden tener el valor de un dato antes de que otra transacción lo hubiera cambiado. El aislamiento de instantánea garantiza que las lecturas de un dato no tengan que esperar por bloqueo a las escrituras concurrentes que ocurran sobre el mismo dato.

Autocommit: propiedad de la sesión de la base de datos por la cual se ignoran las ordenes *commit* y *rollback* haciendo implícitamente un comando *commit* detrás de cada orden SQL SELECT/INSERT/DELETE/UPDATE.

Commit: orden SQL por la cual finaliza la transacción en curso y da comienzo la siguiente, y por la que se registran y validan todos los cambios que hayan podido producirse por sentencias INSERT, DELETE y/o UPDATE dentro de dicha transacción en curso.

MVCC: *Multi-Version Concurrency Control*, es el control de concurrencia a través del mantenimiento de versiones de la misma fila cada vez que ésta sufre un cambio por parte de una transacción. El aislamiento de instantánea es el caso más común de MVCC.

Rollback: orden SQL por la cual finaliza la transacción en curso y da comienzo la siguiente, y por la que se deshacen o retroceden todos los cambios que hayan podido producirse por sentencias INSERT, DELETE y/o UPDATE dentro de dicha transacción en curso.

Serializable: se dice que dos transacciones se pueden ejecutar concurrentemente a través de una planificación serializable equivalente, cuando el resultado final de esa ejecución concurrente es el mismo que el que se obtiene de la ejecución en serie de ambas transacciones.

Segmento UNDO o de anulación, es el área que *Oracle* dedica a guardar las versiones viejas de los datos en su implementación del aislamiento de instantánea.

Transacción: agrupamiento de operaciones sobre la base de datos de manera que respetando el principio de atomicidad, o se cometen todas o ninguna.

VI. Bibliografía

Referencias:

[Bernstein et al. 1987] Bernstein P.A., Hadzilacos V., Goodman N. ***Concurrency Control and Recovery in Database Systems*.** Addison-Wesley. 1987.

[Bolton et al. 2010] Bolton C., Langford J., Ozar B., Rowland-Jones J., Kehayias J., Gross C., Wort S. ***SQL Server® 2008 Internals and Troubleshooting***, Wiley Publishing 2010. (Capítulo 6)

[Connolly & Begg. 2005] Connolly, Thomas M. y Begg, Carolyn E. ***Sistemas de Bases de Datos 4ª ed***. Pearson, 2005. (Capítulo 20)
(Disponible en UBUvirtual: http://0-www.ingebook.com.ubucat.ubu.es/ib/NPcd/IB_BooksVis?cod_primaria=1000187&codigo_libro=2155)

[García-Molina et al. 2009] Garcia-Molina H, Ullman D. y Widom J. ***DATABASE SYSTEMS. The Complete Book 2nd ed.*** Pearson, 2009. (Capítulos 17, 18 y 19).

[Kriegel 2011] Kriegel A. ***Discovering SQL: Hands-On Guide for Beginners***. John Wiley & Sons, 2011.

[Kyte 2010] Kyte T. ***Expert Oracle Database Architecture: Oracle Database 9i, 10g, and 11g Programming Techniques and Solutions, 2nd Ed.*** Apress 2010.

[Lewis 2011] Lewis J. ***Oracle Core: Essential Internals for DBAs and Developers***. Apress 2011.

[Melton y Simon 2002] Melton, Jim y Simon, Alan R. ***SQL:1999: Understanding Relational Language Components***. Morgan Kaufmann, 2002.

[Murali 2004] Murali V. ***Oracle Real Application Clusters.*** Digital Press, 2004.

[Oracle 2003] ***Oracle® Database Application Developer's Guide - Fundamentals 10g Release 1 (10.1)*** December 2003. https://docs.oracle.com/cd/B14117_01/appdev.101/b10795.pdf

[Oracle 2014] ***Oracle® Pro*C/C++ Programmer's Guide 12c Release 1 (12.1)***. Mayo 2014. https://docs.oracle.com/database/121/LNPCC/E53432-01.pdf

[Oracle 2015] ***Oracle® Database Concepts 11g Release 2 (11.2)***, Mayo 2015 https://docs.oracle.com/cd/E11882_01/server.112/e40540.pdf

[Ports & Grittner 2012] Ports, D.R., Grittner K., ***"Serializable Snapshot Isolation in PostgreSQL".*** In Procs VLDB 2012, pp 1850 – 1861.

[Silberschatz et al. 2015] Silberschatz, Abraham, Korth, Henry F. y Sudarshan, S. ***Fundamentos de Bases de Datos 6ªEd.*** McGraw-Hill, 2015. (Capítulos 14, 15 y 16)
(Disponible en UBUvirtual: http://0-www.ingebook.com.ubucat.ubu.es/ib/NPcd/IB_Escritorio_Visualizar?cod_primaria=1000193&libro=5935).

[SQL-Server 2016] SQL2000. © Microsoft Corp. 2016 https://www.microsoft.com/en-us/download/details.aspx?id=51958 (último acceso 10-07-2019)

[Suzuki 2019] Suziki, Hironobu. "***The Internals of PostgreSQL for database administrators and system developers***". Interdb.jp (2019). (Capítulo 5) http://www.interdb.jp/pg/pgsql05.html (último acceso 19-07-2019).

VII. Anexos

1. Versiones de SGBD utilizadas en los ejemplos

Las versiones utilizadas en esta versión de los apuntes fueron:

1. *Oracle 11g*
2. *PostgreSQL 9.6*
3. *SQL-Server 2008 R2*

2. Estrategias de recuperación mediante logs

En todo SGBD existe un módulo que llamaremos gestor del *buffer* es el encargado de gestionar la paginación o

swapping, de manera que en memoria están las páginas de disco correspondientes a las últimas operaciones que realiza el SGBD, ya sean consultas o modificaciones. Cuando se produce una modificación, normalmente, ésta se realiza en la versión de la página o bloque que reside en memoria, pero en general, no se traslada de forma inmediata a disco hasta que el gestor del *buffer* considera que es necesario. Diremos que un bloque o página, es un bloque *sucio (dirty block)* cuando su contenido en memoria no es el mismo que en disco.

Esto nos lleva a un problema: ¿qué pasa si la base de datos se cae y en ese momento existiesen bloques sucios?. Claramente, si no dispusiésemos de un mecanismo de recuperación, la base de datos tendería a quedar en un estado inconsistente, ya que por ejemplo, si una transacción ha cambiado dos bloques de disco, y solo uno de ellos fue persistido en disco antes de la caída de la base de datos, al volver a levantar la base de datos habría cambios de esa transacción que sí que perdurarían, mientras que otros se perderían; violándose así los principios de consistencia y atomicidad de las propiedades ACID.

Por ello, todas las técnicas de recuperación que hay en la actualidad mantienen un fichero *log* que permite restablecer la base de datos con la misma información cometida que contendría justo antes de la caída.

Las primitivas con las que trabaja el gestor del *buffer* del SGBD son ([García-Molina et al. 2009] sección 17.1):

1. INPUT(X): Copia el bloque de disco que contiene el dato X en memoria.
2. READ(X, t): Copia el dato X en una variable local t de la transacción. Si el bloque conteniendo el dato X no estuviese en memoria, entonces se ejecutará un INPUT(X) de manera implícita antes de asignar el dato X a t. READ es una operación abstracta de lectura que podemos asumir es independiente del lenguaje de bases de datos utilizado, en el caso del DML de SQL podríamos asumir que se trata del comando SELECT.
3. WRITE(X, t): Copia el valor de la variable local t en la versión del dato X de memoria. Si el bloque correspondiente al dato X no estuviese en memoria, entonces se ejecutará un INPUT(X) de manera implícita antes de asignar el valor de t al dato X.

   Nuevamente, WRITE es una operación abstracta, en este caso de escritura, también independiente del lenguaje de bases de datos utilizado y del tipo de operación DML considerada. Si considerásemos que el lenguaje DML que se utiliza es SQL, claramente podemos ver que:

   - UPDATE es una escritura o WRITE de un valor en una serie de filas.
   - DELETE es también una escritura si asumimos que la fila tiene un campo “invisible” para los usuarios con información que indica si la fila está borrada o no. Por tanto, borrar es hacer WRITE sobre ese campo.
   - INSERT es una escritura de una fila nueva.
4. OUTPUT(X): Copia el bloque en memoria que contiene el dato t en el disco.
5. FLUSH LOG: Copia en disco todos los bloques **del *log*** en memoria que aún no hayan sido copiados desde la última operación FLUSH LOG. Nota que un bloque de *log* solo hay operaciones de inserción de nuevas entradas al *log* (no hay ni borrados ni modificaciones), por lo que
   1. Una vez que se hace *flush* de un bloque, éste queda inalterable.
   2. Los bloques que vuelca la operación FLUSH LOG contienen únicamente las entradas del *log* que aún no se han grabado en disco.

El *log* es un fichero de disco, y como ya sabemos, las operaciones de escritura sobre disco (y las de lectura también) se hacen bloque a bloque (antiguamente los bloques de la base de datos se correspondían con los de disco del sistema operativo, por lo que lo normal es que un bloque ocupase 512 bytes. Los sistemas operativos actualmente tienen unidades de asignación de mayor tamaño (e.g.;varios Kbs e incluso Mbs), y los bloques de los SGBD también han crecido en tamaño. 8 Kb es un tamaño muy usual actualmente, pero el tamaño de los bloque suele ser configurable (e.g., Oracle actualmente admite tamaños de 2Kb, 4Kb, 8Kb y 16Kb, el tamaño por defecto de Oracle son 8Kb). Por

esa razón, las entradas del *log* inicialmente se escriben en un bloque en memoria, para más tarde escribir ese bloque de memoria a disco con FLUSH LOG haciendo *append.* Esas páginas o bloques de memoria que se persisten mediante *flush* vuelven a quedar tras el *flush* disponibles para registrar nuevas entradas en el *log.* Una operación de *flush,* por tanto, vuelca un número indeterminado de entradas de *log* en el disco. Por ejemplo, en un momento se podría hacer un *flush* correspondiente a 3 páginas, la última de las cuales aún está incompleta (i.e., tiene espacio libre como para haber albergado aún más entradas del *log);* y en otro determinado momento el *flush* podría ser de una sola página, la cual también podría estar incompleta.

El *log* contendrá, en principio, registros de los siguientes tipos ([García-Molina et al. 2009] sección 17.2.1):

- <START T>: que indica que ha comenzado la transacción T.
- <COMMIT T>: que indica que la transacción T se ha completado con *commit*.
- <ABORT T>: que indica que la transacción T se ha terminado con *rollback*.
- <T, X, v>: que indican que en la transacción T el dato X ha cambiado. Según el tipo de estrategia de *logging* usada v puede ser una cosa u otra (el valor original/antiguo del dato X o el valor actualizado/nuevo del dato X debido a la acción de la transacción T).
- <CKPT> o CHECKPOINT, que explicaremos unas líneas más adelante.

Las estrategias para recuperación mediante *logging* son:

1. ***Undo Logging***: En esta estrategia el *log* guarda el valor viejo/original de cada dato que cambia, y se van guardando en disco todos los cambios cada vez que hace *commit* la transacción (i.e.; no hay apenas tiempo de que ningún bloque esté sucio). Cuando se hace la recuperación del sistema se ignoran las entradas en el *log* correspondientes a transacciones cometidas, y se utilizan esos valores viejos para deshacer las transacciones que hayan quedado "a medias" durante la caída y cuyos cambios eventualmente ya hayan sido reflejados en disco.
2. ***Redo Logging***: En esta estrategia el *log* guarda el valor nuevo de los datos cambiados. Cuando se hace la recuperación se ignoran las transacciones no terminadas y se repiten todos los cambios que hayan podido hacer las transacciones cometidas.
3. ***Undo/Redo Logging:*** En esta estrategia el *log* guarda tanto los valores viejos de los datos cambiados como los nuevos. Cuando se hace la recuperación, por un lado se utilizan los valores nuevos para rehacer los efectos de las transacciones cometidas, y por otro lado se utilizan los valores viejos para deshacer los posibles efectos de las transacciones que hayan quedado "a medias" durante la caída y cuyos cambios eventualmente ya hayan sido reflejados en disco.

Si se produce una caída de la base de datos, el *gestor de recuperación* del SGBD leerá el *log* y aplicando una de esas tres técnicas logrará reconstruirla a un estado consistente.

En estos algoritmos, no se procesan todas las entradas del *log* desde el principio de los tiempos, sino que periódicamente se hace una operación de CHECKPOINT que permite descartar las entradas antiguas del *log.*, con lo que en caso de caída el gestor de recuperación ignorará las entradas del *log* anteriores al CHECKPOINT. De hecho, tras un CHECKPOINT el *log* podría reiniciarse para que así no creciese indefinidamente. Nota que en el instante que acabe el CHECKPOINT la base de datos queda sin bloques sucios.

2.1. Undo Logging

Este tipo de estrategia obedece dos reglas U1 y U2 ([García-Molina et al. 2009] sección 17.2.2):

- **U1**: Si una transacción T modifica el elemento X de la base de datos, entonces el registro del *log* < T, X, v > debe de escribirse en el disco ***antes*** de que el nuevo valor de X sea escrito en el disco.
- **U2:** Si la transacción acaba con *commit* el registro de *log* <COMMIT T> debe de escribirse en el disco

después de que el cambio en disco haya sido efectivo, pero tan pronto como sea posible.

Por tanto, ante cada cambio en la base de datos realizado por la transacción:

1. Primero se registra el cambio en el *log* (se guarda el valor antes del cambio). Por ello, en los registros de la forma <T, X, v> de esta estrategia de *logging* la variable v contiene el valor original del dato X (el valor anterior al cambio que hace T).
2. Posteriormente el valor nuevo se guarda de manera persistente en la base de datos.

Una vez se han efectuado estos dos pasos para todos los cambios en la transacción y se hace *commit* de la misma, y se persiste el *commit* en el *log.*

Nota: **¿Cuál es la versión antigua del dato en SQL si la operación no es un UPDATE?.** En el caso de un DELETE la versión antigua del dato es obviamente la fila entera. El caso del INSERT es especial. Por ejemplo Oracle, guarda es el identificador de fila interno que tenga la base de datos para la fila conocido como ROWID. También se consignaría de alguna forma que la operación a deshacer es un INSERT de la fila con dicho ROWID. De esta forma, cuando más adelante (ver sección 2.1.2 Recuperación con Undo Logging) se utilice la información del *log* para restaurar la base de datos, se borrará la fila con ese ROWID.

Ejemplo: Tenemos una transacción que:

1. Marca un asiento de un avión como ocupado
2. Decrementa en uno el número de asientos disponibles

Supongamos que el asiento a ocupar es el 10 y hay 20 asientos disponibles

Paso		t	M-E	M-N	D-E	D-N	Log
1							<START T>
2	READ(estado asiento 10, t)	Libre	Libre		Libre	20	
3	t:= Ocupado	Ocupado	Libre		Libre	20	
4	WRITE(estado asiento 10, t)	Ocupado	Ocupado				<T, estado asiento 10, Libre>
5	READ(nº asientos libres, t)	20	Ocupado	20	Libre	20	
6	t := t -1	19	Ocupado	20	Libre	20	
7	WRITE(nº asientos libres, t)	19	Ocupado	19	Libre	20	<T, nº asientos libres, 20>
8	FLUSH LOG						
9	OUTPUT(estado asiento 10)	19	Ocupado	19	Ocupado	20	
10	OUTPUT(nº asientos libres)	19	Ocupado	19	Ocupado	19	
11							<COMMIT T>
12	FLUSH LOG						

Ilustración 24: Estado de los distintos niveles del gestor del buffer del SGBD con Undo-Logging. M-E = Estado asiento 10 en Memoria;M-N = nº asientos libres en Memoria; D-E = Estado asiento 10 en Disco;D-N = nº asientos libres en Disco.

El FLUSH LOG del paso 8 garantiza la regla U1 haciendo que los registros < T, X, v > sean escritos en disco. El FLUSH LOG del paso 12 garantiza la regla U2 registrando el COMMIT en el *log,* tras las operaciones de OUTPUT. Nota que como pueden transcurrir *n* transacciones simultáneamente, todas podrían estar escribiendo sus registros de *log* en el mismo bloque de memoria, por lo que ese mismo bloque contendrá de forma intercalada, registros de las *n* transacciones. Una consecuencia de esta situación es que una cualquiera de esas *n* transacciones podría realizar una operación FLUSH LOG, haciendo que se volcaran en disco antes de tiempo los registros de *log* de las otras transacciones. Sin embargo, si algunos registros se guardasen de forma anticipada "en principio" no tendría ningún efecto pernicioso, ya que seguiría respetándose la regla U1.

Nota: decimos "en principio" porque si que existe un caso especial con el que el sistema tendrá que ser

cauto. Si dos transacciones concurrentes T1 y T2 modificasen el mismo dato X, y T1 hiciese FLUSH LOG la regla U1 podría violarse para T2, ya que se está registrando el cambio de T2 en el *log "antes"* de que ese cambio de T2 se registre en el disco. En este caso, la solución pasa por utilizar bloqueos para evitar que varias transacciones registren < T1, X, v1 > y < T2, X, v2 >. No obstante, en la práctica, sabemos que el gestor de concurrencia si que se produce un bloqueo de escritura del dato X para preservar el aislamiento (ver secciones 5 y 6.2) con independencia de que se utilice o no *undo logging*, por lo que ese mismo bloqueo sobre el dato X puede extender su acción a un bloqueo adicional que además impida insertar registros < T2, X, v2 > en el bloque de memoria del *log.*

Es importante notar que con esta estrategia de *logging* los bloques apenas tienen tiempo de estar *sucios* en memoria, pues las operaciones OUTPUT(X) de esa transacción han de realizarse antes del <COMMIT T>. De hecho, **una vez se hace *commit* no hay bloques sucios provocados por esa transacción.**

2.1.1 Problema del Undo Logging

El problema principal de esta técnica es que obliga a hacer las operaciones de OUTPUT con anterioridad al *commit*, con lo que el número de operaciones lectura-escritura a disco no puede ser optimizado de forma que solo hiciésemos OUTPUT cuando realmente necesitáramos desalojar una página de memoria por problemas de espacio; sino que cada *commit* obliga a hacer OUTPUT de todos los bloques sucios correspondientes a sus cambios.

Más adelante veremos que tanto la técnica de *Redo – Logging* como *Undo/Redo - Logging* evitan este problema.

2.1.2 Recuperación con Undo Logging

El algoritmo de recuperación de la base de datos en caso de que haya una caída consistiría en examinar el *log* comenzando por el final y por cada <START T> ([García-Molina et al. 2009] sección 17.2.3):

1. Si existe el correspondiente <COMMIT T> no hay que hacer nada, ya que la regla U2 garantiza que los cambios que se han producido en la transacción, ya han sido registrados en el disco mediante operaciones OUTPUT.
2. Si no existe el <COMMIT T> significa
    - O bien que la transacción ha retrocedido (en ese caso existirá una entrada en el *log* <ABORT T>)
    - O bien se ha quedado a medias (no habrá en el *log* ni *commit* ni *abort* para esa transacción).

    En ambos casos habrá que deshacer los cambios que se hayan producidos por las operaciones OUTPUT que hayan registrado cambios correspondientes a esa transacción. Afortunadamente, la regla U1 garantiza que en el *log* estén disponibles los registros *undo* de la forma <T, X, v> que permiten restaurar los valores originales. En el caso de que la transacción se haya quedado a medias, además hay que escribir en el *log* <ABORT T>, y hacer FLUSH LOG para que ese *abort* quede registrado por si eventualmente se produce otra caída durante el propio proceso de restauración.

Por ejemplo:

- CASO 1: Supongamos que en la transacción anterior (Ilustración 24) la base de datos se cae tras el paso 12. Entonces el registro <COMMIT T> ya está en disco, por lo que todos los registros de esa transacción T serían ignorados en el proceso de recuperación.
- CASO 2: Supongamos que en la transacción anterior la base de datos se cae entre los instantes 11 y 12. Es decir, el *commit* tuvo lugar en bloque del *log* residente en memoria, pero no dio tiempo a que persistiera en disco.

    Nota que podría ser que, aún ocurriendo la caída entre los instantes 11 y 12, el disco contuviese casualmente <COMMIT T> debido a que a que el *commit* ya estaba en ese momento en el bloque del *log* en memoria y el FLUSH LOG de otra transacción concurrente lo hubiera reflejado en disco. Pero

en ese caso estaríamos en la misma situación de CASO 1.

Suponemos que en este caso 2 el registro <COMMIT T> no está en disco. Entonces el gestor de recuperación de la base de datos tendría que leer de atrás hacia delante el *log* e ir restaurando los valores antiguos mediante la información de los registros *undo* (<T, X, v>). Cuando en ese recorrido hacia delante encontrase <START T>, escribiría <ABORT T> y haría un FLUSH LOG para persistir el *abort*.

- CASO 3: La caída sucede entre los pasos 10 y 11. Es decir el <COMMIT T> todavía no ha tenido lugar ni siquiera en el bloque del *log* en memoria. En este caso se operaría de igual forma que en el caso 2 deshaciendo los cambios.
- CASO 4: La caída tiene lugar entre los pasos 8 y 10. Es la misma situación que los casos 2 y 3, pero puede que no se hayan producido alguna o todas las operaciones de OUTPUT, con lo que los valores nuevos (*Libre* y *20*) puede que no estén en disco. No obstante, si procedemos como en los casos 2 y 3 el resultado seguiría siendo correcto, pues no tiene ningún efecto restaurar el valor de *Libre* a *Libre*, o el *20* a *20.*
- CASO 5: La caída tiene lugar antes del paso 8. En principio ninguno de los registros de *log* de la transacción T parece que estén en disco; aunque podría ser que algunos si lo estuvieran porque otra transacción concurrente hubiese hecho FLUSH LOG de bloques de *log* que casualmente tuvieran registros de T. No obstante, la regla U1 garantiza que si los cambios se han producido en disco (i.e.; operaciones OUTPUT) debería de existir en disco el correspondiente registro *undo* (<T, X, v>). Por tanto:
  - Si no existen registros (<T, X, v>) no hay nada que restaurar; y
  - Si casualmente existieran registros (<T, X, v>) porque otra transacción hiciera *flush* estaríamos en una situación similar al caso 4, en el que la restauración realmente no tiene efecto porque el valor que contiene la base de datos y el valor del registro *undo* son el mismo.

2.1.3 Checkpointing en undo logging

En su versión más simple, un CHECKPOINT con *undo logging* seguiría los siguientes pasos ([García-Molina et al. 2009] sección 17.2.4):

1. El SGBD deja en espera todas las nuevas transacciones que intenten comenzar
2. La operación CHECKPOINT queda esperando el *commit* o *rollback* de todas las transacciones que hubieran empezado antes del CHECKPOINT.
3. FLUSH LOG
4. Escribir un registro de *log* <CKPT> o registro de CHECKPOINT (en memoria)
5. Hacer un nuevo FLUSH LOG para guardar en el log de disco el registro <CKPT>.

Nota que no hay que volcar los bloques sucios en memoria, ya que las operaciones OUTPUT (X) ya han tenido lugar antes de cada <COMMIT, T> no dejando bloques sucios en memoria.

En la práctica esta operación tiene demasiado coste porque dejamos la base de datos inoperativa durante el tiempo que tarden en cerrarse las transacciones que estén en ese momento en curso, lo que puede ser inaceptable en el caso de que por ejemplo haya transacciones largas o que por descuido de los usuarios no acaban de cerrarse. Por ello, se suele aplicar la siguiente variante (i.e., *nonquiscent checkpoint,* [García-Molina et al. 2009] sección 17.2.5):

1. Escribir un registro especial de *log* <START CKPT (T1 ... TK)> donde T1 … TK son las transacciones activas, aún no cerradas, en ese momento.
2. Hacer FLUSH LOG para guardar en disco ese registro especial.
3. Esperar a que terminen todas las transacciones T1 … TK pero no prohibir que comiencen nuevas transacciones
4. Cuando acaben T1 … TK escribir un registro de *log* especial <END CKPT> y hacer FLUSH LOG.

De esta manera, si al procesar el *log* de atrás hacia delante:

1. Encontramos <END CKPT>, es decir, la base de datos se ha caído después de terminar el CHECKPOINT: las únicas transacciones que hay que deshacer son aquellas que habiendo empezado después de <START CKPT (T1 ... TK)> no hayan registrado un *commit*; ya que las transacciones T1 … TK acabaron.
2. No encontramos <END CKPT>, es decir, la base de datos se ha caído mientras se hacía el CHECKPOINT: habrá que deshacer
    1. De entre las transacciones que hayan comenzado después del <START CKPT T1 … TK>, las que no hayan acabado con *commit*.
    2. De entre las transacciones T1 … TK aquellas que no hubieran acabado con *commit* (esto supone que habría que leer el *log* hasta el comienzo de esas transacciones, el cual es anterior al <START CKPT T1 … TK>).

2.2. Redo Logging

Este tipo de estrategia obedece una única regla ([García-Molina et al. 2009] sección 17.3.1):

- **R1:** Antes de modificar cualquier dato X en disco habrá que hacer persistentes todas las entradas en el *log* correspondientes a esa transacción, incluyendo < T, X, v> y < COMMIT T>.

En esta estrategia el valor v de <T, X, v> representa el valor nuevo (i.e., tras la actualización) del dato X.

> ***Nota:* ¿Cuál es la versión nueva del dato en SQL si la operación no es un UPDATE?.** En el caso de un INSERT la versión nueva del dato es obviamente la fila entera. El caso del DELETE es especial. Por ejemplo Oracle lo que se guarda es el identificador de fila interno que tenga la base de datos para la fila (i.e.; el ROWID). También se consignaría de alguna forma que la operación a rehacer es un DELETE. De esta forma, cuando más adelante (ver sección 2.2.1 Recuperación con Redo Logging) se utilice la información del *log* para restaurar la base de datos, se borrará la fila con ese ROWID.

La Ilustración 25 representa el ejemplo de transacción en el que se marca el asiento 10 de un avión como ocupado, y se decrementa en uno el número de asientos disponibles utilizando *redo-log*. Como diferencias respecto *undo-log*:

1. En la columna *log* vemos que se guardan los valores nuevos (*Ocupado* y *19*).
2. Que el registro de <COMMIT T> en el *log* se adelanta al mismo momento en que finaliza la transacción, sin necesidad de que se realicen previamente las operaciones OUTPUT. Vemos que se hace un único FLUSH LOG que guarda tanto el *commit* como los registros <T, X, v>.
3. Las operaciones OUTPUT se realizarán en cualquier momento posterior al FLUSH LOG (aunque en la tabla se consigna a los instantes 10 y 11 podrían ser instantes muy posteriores en el tiempo), por lo que el gestor del *buffer* tiene libertad de decidir cuando hacer OUTPUT, y hacerlo en el momento más óptimo para reducir escrituras en disco (típicamente cuando el *buffer* del SGBD esté lleno y necesite subir a memoria un bloque intercambiándolo por otro bloque que ya esté en memoria y que fuese un bloque sucio).

Paso		t	M-E	M-N	D-E	D-N	Log
1							<START T>
2	READ(estado asiento 10, t)	Libre	Libre		Libre	20	
3	t:= Ocupado	Ocupado	Libre		Libre	20	
4	WRITE(estado asiento 10, t)	Ocupado	Ocupado				<T, estado asiento 10, Ocupado>
5	READ(nº asientos libres, t)	20	Ocupado	20	Libre	20	
6	t := t -1	19	Ocupado	20	Libre	20	

7	WRITE(nº asientos libres, t)	19	Ocupado	19	Libre	20	<T, nº asientos libres, 19>
8							<COMMIT T>
9	FLUSH LOG						
10	OUTPUT(estado asiento 10)	19	Ocupado	19	Ocupado	20	
11	OUTPUT(nº asientos libres)	19	Ocupado	19	Ocupado	19	

Ilustración 25: Estado de los distintos niveles del gestor del buffer del SGBD con Redo-Logging. M-E = Estado asiento 10 en Memoria;M-N = nº asientos libres en Memoria; D-E = Estado asiento 10 en Disco;D-N = nº asientos libres en Disco.

2.2.1 Recuperación con Redo Logging

Debido a la regla R1 nunca puede haber un cambio reflejado en el disco si no existe en el *log* la entrada <COMMIT, T> correspondiente a esa transacción. Por lo tanto, el proceso de recuperación ignora todas las entradas correspondientes a transacciones que no tengan <COMMIT T>, lo cual incluye las transacciones que han quedado inacabadas al caerse la base de datos.

El algoritmo de recuperación de la base de datos en caso de que haya una caída consistiría en ([García-Molina et al. 2009] sección 17.3.2):

1. Identificar las transacciones con <COMMIT T>
2. Leer el *log* de delante hacia atrás y por cada entrada <T, X, v>
    1. Si T está cometida hacer OUTPUT(X, v)
    2. Sino está cometida, no hacer nada, salvo que esté inacabada (no tenga tampoco <ABORT T>), en cuyo caso se escribe <ABORT T> en el *log* haciendo FLUSH LOG.

2.2.2 Checkpointing en redo logging

A diferencia del *undo-logging*, podría haber bloques sucios que trascendiesen a transacciones anteriores a las que están corriendo en ese momento. Por tanto hay que hacer OUTPUT(X) de todos esos bloques. El algoritmo sería ([García-Molina et al. 2009] sección 17.3.3):

1. Escribir un registro especial de *log* <START CKPT (T1 ... TK)> donde T1 … TK son las transacciones activas, aún no cerradas, en ese momento. Hacer FLUSH LOG para guardar en disco ese registro especial
2. Volcar en disco todos los bloques sucios modificados por transacciones que hayan hecho *commit* antes del <START CKPT (T1 ... TK)>
3. Escribir la entrada <END CKPT> en el *log*. Hacer FLUSH LOG para guardar en disco esa entrada especial.

En caso de procesar el *log* para reestablecer la base de datos pueden ocurrir las siguientes situaciones:

1. El *log* contiene <END CKPT>, es decir el CHECKPOINT acabó con éxito. En ese caso las transacciones anteriores a T1 … TK que acabaron con *commit*, ya han actualizado sus cambios en disco, por lo que solo hay que procesar las entradas <T, X, v> de las transacciones cometidas de T1 … TK y posteriores a TK, con objeto de restaurarlas.
2. El *log* no contiene <END CKPT>, pero contiene <START CKPT (T1 ... TK)>, lo que significa que la base de datos se cayó durante el proceso de CHECKPOINT. En ese caso hay que rehacer todas las transacciones cometidas a partir del último <END CKPT> (o desde el principio del *log* si no existiera un <END CKPT>.

2.3. Undo/Redo Logging

Este tipo de estrategia obedece una única regla ([García-Molina et al. 2009] sección 17.4.1):

- **UR1:** Antes de modificar en disco un elemento X de la base de datos como efecto de la transacción T, es

necesario que el registro < T, X, v, w> se registre en el disco. En este caso la entrada en el *log* registra tanto el valor original de X, antes del cambio, representado por v; como el valor tras el cambio, representado por w.

La Ilustración 26 muestra el ejemplo de transacción en el que se marca el asiento 10 de un avión como ocupado, y se decrementa en uno el número de asientos disponibles utilizando *undo*/*redo-log*. Como diferencias respecto a las estrategias anteriores:

1. En la columna *log* vemos que se guardan tanto los valores originales de los datos, como los valores actualizados.
2. Como en el caso del *undo logging*, se hace un FLUSH LOG antes de cualquier OUTPUT(X). Esto aparentemente es una limitación pero:
3. El <COMMIT T> y los OUTPUT se puede hacer en cualquier orden y en cualquier momento a partir del FLUSH LOG (no necesariamente en los instantes y en el orden reflejado en la Ilustración 26), por lo que esta estrategia es al final la más flexible, pues se tiene la libertad de paginar los bloques sucios cuando el gestor del *buffer* crea que es el momento óptimo (como en *redo-logging*) pero la persistencia <COMMIT T> puede posponerse y aprovechar un FLUSH LOG que haga otra transacción concurrente (lo cual no lo permitía el *redo-logging*).

Paso		t	M-E	M-N	D-E	D-N	Log
1							<START T>
2	READ(estado asiento 10, t)	Libre	Libre		Libre	20	
3	t:= Ocupado	Ocupado	Libre		Libre	20	
4	WRITE(estado asiento 10, t)	Ocupado	Ocupado				<T, estado asiento 10, Libre, Ocupado>
5	READ(nº asientos libres, t)	20	Ocupado	20	Libre	20	
6	t := t -1	19	Ocupado	20	Libre	20	
7	WRITE(nº asientos libres, t)	19	Ocupado	19	Libre	20	<T, nº asientos libres, 20, 19>
8	FLUSH LOG						
9	OUTPUT(estado asiento 10)	19	Ocupado	19	Ocupado	20	
10							<COMMIT T>
11	OUTPUT(nº asientos libres)	19	Ocupado	19	Ocupado	19	

Ilustración 26: Estado de los distintos niveles del gestor del buffer del SGBD con Undo/Redo-Logging. M-E = Estado asiento 10 en Memoria;M-N = nº asientos libres en Memoria; D-E = Estado asiento 10 en Disco;D-N = nº asientos libres en Disco.

***Nota**:* El hecho de que el *commit* no persista en disco tan pronto acabe la transacción puede producir efectos indeseables. Si la base de datos se cayese en el paso 9, no habría constancia de que la transacción estuviera cometida, y en la recuperación el asiento 10 aparecería como libre, lo que generaría un conflicto si otro usuario logra reservarlo tras la restauración de la base de datos. Por ello, podría añadirse la regla ([García-Molina et al. 2009] sección 17.4.1):

- **UR2:** Las entradas <COMMIT T> deben de volcarse con FLUSH LOG a disco tan pronto como tengan lugar en memoria.

2.3.1 Recuperación con Undo/Redo Logging

El algoritmo de recuperación tiene en este caso 2 pasos ([García-Molina et al. 2009] sección 17.4.2)

1. Hacer *redo* de las transacciones cometidas empezando por las primeras hasta las últimas (como se hacía en *redo logging*). Esto permite, por ejemplo en el caso de la Ilustración 26 que el OUTPUT del paso 12 fuera restaurado, y en general todos los OUTPUT que se hubieran hecho con posterioridad a un <COMMIT, T>,

o que no se hubieran hecho, a pesar de que si se hubiera hecho el <COMMIT, T> en disco.

2. Hacer *undo* de las transacciones inacabadas empezando por las últimas hasta llegar a las primeras (como se hacía en *undo logging*). Si en el ejemplo anterior la base de datos se hubiese caído justo después del OUTPUT del paso 9, habría que haber deshecho ese OUTPUT, y de esta cuestión se encargaría este paso.

2.3.2 Checkpointing en Undo/Redo

El algoritmo sería ([García-Molina et al. 2009] sección 17.4.3):

1. Escribir un registro especial de *log* <START CKPT (T1 ... TK)> donde T1 … TK son las transacciones activas, aún no cerradas, en ese momento. Hacer FLUSH LOG para guardar en disco ese registro especial
2. Volcar en disco todos los bloques sucios modificados por *"todas"* las transacciones.
   Nota que en el *checkpoint* con *redo* solo se volcaban en disco bloques sucios que hayan hecho *commit* antes del <START CKPT (T1 ... TK)>, mientras que en este caso se vuelcan todos. Esto es una simplificación respecto al caso del *redo* debido a que en *undo/redo* podemos hacer OUTPUT de partes de la transacción antes del *commit,* como en el paso 9 de la Ilustración 25, y no tiene efectos indeseables.
3. Escribir la entrada <END CKPT> en el *log.* Hacer FLUSH LOG para guardar en disco esa entrada especial.

2.4. Ejemplo de implementación en un SGBD comercial: Logging en Oracle

Oracle implementa una estrategia de *logging* que es una mezcla entre *Redo* y *UndoRedo*. A primera vista es una estrategia *Redo* en la que la que se van rellenando una serie de ficheros REDO LOG. Cuando acaba de llenar uno, comienza a llenar el siguiente. Estos ficheros contienen los valores nuevos tras el cambio; es decir, la tuplas < T, X, v > que veíamos en la sección 2.2 de los anexos para la estrategia *redo*.

Sin embargo, Oracle mantiene un *tablespace undo* (ver anexos 3.1 y 4.3) en el que se guardan las versiones antiguas de los datos para las transacciones más recientes a fin de implementar el aislamiento de instantánea. Cuando una transacción cambia una fila, la información de la versión antigua de la fila se guarda en el *tablespace undo*. Pero el *tablespace undo*, es un *tablespace* más, por lo que a efectos de recuperación de la base de datos se gestiona como cualquier otro *tablespace* de usuario. Esto significa que cada vez que modificamos una fila de una tabla cualquiera, se hace una especie de inserción de la versión antigua de la fila en el *tablespace undo*, y esa especie de inserción es también llevada al REDO LOG como una tupla < T, X, v > especial (i.e.; se marca internamente que es del *tablespace undo*). Como la variable v representa la información nueva, y la información nueva del *tablespace undo* es la información vieja de la transacción, estamos guardando "por la puerta de atrás" la información *undo* en el mismo fichero REDO LOG que guarda el resto de entradas *redo.*

En recuperación, en lugar de aplicar el algoritmo *redo log* visto en el anexo 2.2.1, se puede aplicar una variante del algoritmo *undo/redo* del anexo 2.3.1, ya que tenemos la información vieja de las transacciones incompletas. Con lo cual en lugar de someter la estrategia de *logging* a la regla R1, podemos someterla a la UR1, que es más flexible (i.e.; nos deja retrasar el momento de hacer persistente el <COMMIT T>).

Sin embargo, en la práctica ese retraso no suele darse. Como veíamos en el anexo 2.3 convenía añadir la regla UR2 en *undo/redo* que hacía que el <COMMIT T> persistiera tan pronto como tuviera lugar el comando *commit.*

Para ello, la sintaxis del comando COMMIT en Oracle es más extensa[17]. Podemos hacer:

- `"COMMIT WRITE IMMEDIATE;"` que es la opción por defecto[18], que fuerza a que se ejecuten inmediatamente las operaciones <COMMIT T> y FLUSH LOG con la ejecución del comando SQL *commit*. Es más, `"COMMIT WRITE IMMEDIATE WAIT;"` (WAIT también es la opción por defecto[18])

17 Ver https://docs.oracle.com/cd/B19306_01/server.102/b14200/statements_4010.htm#i2060233

18 El parámetro de inicialización de la base de datos *COMMIT_WRITE* permite alterar este valor por defecto.

forzaría que el gestor del *log* (el proceso en *background* LGWR en Oracle) esperara la confirmación de que la escritura ha tenido lugar antes de seguir atendiendo nuevas peticiones.

- "`COMMIT WRITE BATCH;`" que permite que se retrasen las operaciones <COMMIT T> y FLUSH LOG correspondientes a la ejecución del comando SQL *commit* a otro momento, minimizando las escrituras en disco (en la misma escritura se haría *flush* de varios *commits*). Esto puede ser interesante para optimizar el rendimiento si no tenemos la exigencia del ejemplo de la reserva del avión que ilustraba la necesidad de la regla UR2, pero no deja de ser un escenario raro en la práctica.

Por lo tanto, el algoritmo de recuperación de Oracle aunque se podría comportar como en *Undo/Redo logging*, en la práctica se comporta como en *Redo Logging* (no hay que deshacer OUTPUTs no cometidos).

Nota: No obstante esa información *undo* en el REDO LOG permite hacer otras cosas, como por ejemplo restaurar la base de datos a un instante de tiempo concreto (tecnología *fashback* de Oracle), con fines de análisis o auditoría. La idea es ir haciendo *undo* de las transacciones cometidas hasta regresar a un punto pretérito de la base de datos y en el hacer consultas (i.e. *flashback queries*) o por ejemplo hacer *rollback* de una transacción pasada y de todas las que se basaron en ella, por ejemplo.

2.5. Logging y almacenamiento de estado sólido

Los discos de estado sólido (SSD), aún siendo más rápidos que los discos duros tradicionales, siguen teniendo tiempos de acceso más lentos que la memoria RAM. Luego, con la tecnología SSD actual, aunque la base de datos utilice un SSD, podemos tener también datos en el disco que no se corresponden con las transacciones cometidas en memoria RAM, y por tanto la necesidad de seguir haciendo *logging*. Es más, los SSD son más rápidos, pero también tienen una vida útil más corta, dando con el uso errores de E/S con más probabilidad que los discos tradicionales, por lo que es más interesante, si cabe, tener un mecanismo de recuperación.

Una tendencia reciente son las llamadas *Bases de Datos en Memoria* o *IMDb (In Memory Databases)* que almacenan "todos" los datos en memoria principal para acelerar los tiempos de acceso. Las IMDb en la actualidad se utilizan principalmente para hacer consultas (i.e.; SELECTs) en entornos de minería de datos e inteligencia empresarial, por lo que no hay inconsistencias entre los datos en memoria y disco, y por tanto no hay necesidad de *log* (e.g., *SQL Server In-Memory OLTP*).

Sin embargo, recientemente han surgido sistemas IMDb que soportan transacciones (e.g. Oracle *TimesTen*); en estos sistemas las alternativas para mantener la durabilidad pueden pasar por:

1. Seguir aplicando una técnica de *logging* con *checkpoints*. De hecho, aunque el *log* residiera en un disco duro tradicional no constituiría un cuello de botella, el aumento de velocidad de un IMDb frente un SGBD tradicional seguiría siendo muy significativo, porque el *log* se escribe con operaciones *append* que no necesitan desplazamiento de la cabeza del disco.
2. Utilizar nuevas tecnologías NVRAM (RAM no volátil) que hicieran persistente la memoria. Estas tecnologías son muy incipientes, pero se puede intuir que a la larga harán caer en desuso la necesidad de hacer *logging*.

3. Aislamiento de instantánea en algunos sistemas comerciales

Los sistemas comerciales que implementan aislamiento de instantánea utilizan la estrategia de *primera actualización gana.* Cada uno utiliza aproximaciones diferentes para crear la instantánea a partir de las versiones de fila, limpiar las versiones obsoletas etc … A continuación se esboza como lo hacen *Oracle*, *PostgreSQL* y *SQL-Server*.

3.1. Aislamiento de instantánea en Oracle

Oracle para implementar las versiones organiza el espacio de datos en dos áreas:

1. Un área de la base de datos que contiene la versión más reciente del dato. No hay un término concreto para designar a este área, quizás en el caso de *Oracle* podríamos identificarla como el ***tablespace users***.
2. Otro área de la base de datos donde ir guardando las versiones viejas de los datos, en las versiones 9i y posteriores *Oracle* se la da el nombre de ***UNDO SEGMENT*** o segmento *UNDO* (ver anexo 4.3).
   La información que contiene el segmento UNDO no se guarda como filas o versiones de filas, sino en forma de ***change vectors***, que relatan las operaciones físicas que hay que realizar para restaurar un cambio. Por ejemplo, si hemos modificado el campo VARCHAR *nombre* que contenía *Rober* al nuevo valor *Roberto,* el *chage vector* nos indicará que hay que ampliar 2 bytes el tamaño utilizado para el campo y que además hay que escribir en ese campo el contenido *Roberto.* Este formato de *change vector* es el mismo en el que se registran los cambios en el REDO *log* (ver anexo 2).

Nota que en el caso de que hubiera que hacer un *rollback* que afectase a un dato modificado, habría que llevar la versión anterior a ese cambio guardada en el segmento *undo,* al *tablespace users.* Sin embargo, en el caso de que hubiera que hacer *commit*, no hay que copiar la versión correspondiente a esa transacción en el *tablespace users*, porque ya está en él. De esta forma, *commit* se convierte en general en una operación mucho más ligera que *rollback*, lo cual es deseable porque lo normal es que la inmensa mayoría de las transacciones acaben con *commit.*

La forma de asignar a las transacciones una marca de tiempo es a través del *SCN* o *System Change Number*. El SCN se incrementa, salvo raras excepciones, únicamente cada vez que se produce un *commit* ([Lewis 2011] capítulo 1, sección *Oracle in Processes*). Si observamos como va cambiando el SCN, aunque solo haya una sesión utilizando el sistema, veremos que los SCN que se obtienen de ir cometiendo las transacciones normalmente no son consecutivos porque entre medias se producen otros "*commits"* de operaciones internas de Oracle.

Una base de datos *Oracle* tiene típicamente un solo *tablespace UNDO* y cada tablespace UNDO puede contener varios segmentos UNDO (ver anexos 4.2 y 4.3). En la cabecera de un segmento UNDO (i.e.; en la primera página del segmento) existe diversa información de control, quizás las más importantes sean:

1. La ***tabla de transacciones*** ([Lewis 2011] capítulo 3, sección *Transactions and Undo*). Cada entrada en la tabla de transacciones representa una transacción reciente. La tabla de transacciones tiene un tamaño limitado (i.e.; 34 entradas en *Oracle 11g* para una base de datos con tamaño de página de 8Kb, pudiendo haber más entradas si aumentamos el tamaño de página). Por tanto, si el número de segmentos UNDO es limitado, y el número de entradas en la tabla también, no es posible mantener la información histórica de todas las transacciones, sino únicamente de la más recientes. Es decir, las entradas en la tabla de transacciones se van reutilizando cíclicamente.
   La tabla de transacciones consta múltiples campos. Los siguientes son los más importantes para entender cómo se genera la instantánea ([Lewis 2011] capítulo 3, sección *The Transaction Table*):
   1. *index:* es el número de entrada en la tabla de transacciones, típicamente entre 1 y 34.
   2. *state:* que puede valer activa o inactiva
   3. *cflags*: en el caso de que la transacción esté inactiva indica si acabado con *commit* o con *rollback*.
   4. *wrap#*: Es el número de veces que esa entrada ha sido reutilizada por distintas transacciones desde que se instaló la base de datos. Recuerda que el número de entradas es limitado y por tanto se van reutilizando. El criterio es reutilizar aquella transacción cometida hace más tiempo (i.e., la que teniendo estado inactivo tiene un SCN más bajo).
   5. *scn:* Es el SCN de la transacción asociada a esa entrada de la tabla. Caso de ser una transacción inactiva, este campo marca el momento en que termina. Si acabase con *rolback* internamente tiene

lugar una especie de "transacción interna" que acaba siempre con *commit* y que consiste precisamente en restaurar los valores originales. Por tanto, las transacciones con *rollback* también tienen un SCN correspondiente a un *commit* (recuerda que solo se obtiene un nuevo SCN cada vez que se produce un *commit).*

6. *dba:* o *Data Block Address* Indica dónde está la información del segmento UNDO correspondiente a esa transacción. El campo *dba* contiene la dirección del registro del último cambio hecho por la transacción; y a su vez, dicho registro, en su caso, podría contener la dirección del registro anterior (i.e.; correspondiente al penúltimo cambio hecho por la transacción), y así sucesivamente, formando una lista enlazada de cambios a deshacer en caso de *rollback* o en caso de construcción de una instantánea. Nota que como la forma de deshacerlos es siguiendo un orden cronológico inverso, resulta muy conveniente apuntar al último cambio, que éste apunte al penúltimo etc …

2. El ***transaction control***, que es un registro con una serie de campos que contienen información de control de ese segmento UNDO, entre la que destaca:
   1. *scn:* Es el SCN más antiguo en ese momento en la tabla de transacciones.
   2. *uba* o UNDO *Block Adress*:, contiene una dirección del segmento UNDO correspondiente a el campo *dba* de la tabla de transacciones de la última entrada que fue reutilizada. Es decir, cuando una entrada de la tabla de transacciones es reutilizada, el valor de su *dba* no se pierde, porque se guarda en el campo *uba* de *transaction control*.

Al comenzar una transacción, lo primero que registra en su información UNDO es:

1. El valor que tenían todos los campos de la entrada de la tabla de transacciones que va a ocupar antes de ser reutilizada.
2. El valor que tenía *uba* en *transaction control,* que recordemos apunta a la información UNDO de la transacción que se reutilizó antes de ésta, y por tanto, también está a punto de cambiar.

Recordemos que la información UNDO se registra en una lista de cambios en orden cronológico inverso, y que la dirección donde comienza la lista figura en el campo *dba* de la tabla de transacciones; de manera que a través del campo *uba* de *transaction control* podríamos restaurar por completo la última entrada que se perdió en la tabla de transacciones, así como el *uba* que tenía *transaction table* cuando esta desapareció (i.e.; es posible hacer una especie de *rollback* para recuperar la última entrada que se perdió en la tabla de transacciones ([Lewis 2011] capítulo 3, sección *Transaction Table Rollback*)). Pero si repetimos este procedimiento recursivamente, podríamos restablecer la penúltima entrada en perderse, la antepenúltima, etc... hasta regresar al estado de la tabla de transacciones en un determinado momento pretérito.

Por tanto, cuando una transacción comienza:

1. Se le asigna un bloque UNDO con espacio libre.
2. Se le asigna una entrada en su tabla de transacciones,
3. Se registra como primeras operaciones de ese bloque UNDO los valores originales correspondientes a esa entrada en la tabla de transacciones, y el valor anterior del campo *uba* en *transaction control*.
4. Se modifica el campo *uba* de la *transaction table* con el *dba* que había anteriormente en esa entrada de la tabla de transacciones,
5. A esa entrada se la incrementando el *wrap#*, se marca *state* como activa, y *scn* se pone a cero.
6. En el campo *dba* se consigna la dirección del bloque UNDO asignado.

Ahora la transacción es identificable mediante su ***Xid*** o *transaction Id*, el cual se forma a través de la concatenación del número de segmento UNDO donde está la tabla de transacciones que le corresponde+ el número de entrada en la tabla de transacciones + *wrap#*. Por ejemplo el *Xid* "`0x0004.009.000007ec`" (nota que está en hexadecimal)

significa que es la transacción que está en la tabla de transacciones del segmento UNDO 4, en la entrada de su tabla de transacciones número 9, y que es la vez número "`000007ec`" (que es 2028 en decimal) que dicha entrada ha sido reutilizada para alojar la información de una transacción.

En *Oracle* la información de correspondiente a los bloqueos está distribuida en cada cabecera cada página del *tablespace users,* en la llamada ITL o *Interested Transaction List*. El distribuir en cada página esta información de control hace que la gestión de bloqueos sea escalable, evitando un cuello de botella por mucho que crezca el número de bloqueos.

La ITL es una pequeña tabla en la que cada fila representa a una transacción que ha logrado bloquear al menos una fila en esa página del *tablespace users.* Nota que cada entrada en la lista representa una transacción que está "interesada" en actualizar una determinada fila, de ahí el nombre *Interested Transaction List*. La ITL consta de los siguientes campos ([Lewis 2011] capítulo 3, sección *The Interested Transaction List*):

1. *Itl:* Es el número de entrada en la ITL que la identifica unívocamente dentro de cada ITL (i.e.; dentro de cada página del *tablespace users*). Cada fila de una tabla tiene un campo interno llamado *lock byte.* Cuando una fila está bloqueada almacena este número *Itl* en el *lock byte* para indicar por qué transacción de la ITL está bloqueada esa fila (i.e; el *lock byte* vale "0" si la fila no está bloqueada, vale "1" si está bloqueada por la primera transacción en la ITL, "2" si por la segunda etc …). Nota que varias filas de la misma tabla pueden tener el mismo valor mayor que cero para su *lock byte*, lo que indicaría que están siendo bloqueadas por la misma transacción.
2. *Xid:* Es el identificador de la transacción que está ejerciendo un bloqueo sobre datos de esa página. Por tanto, funciona como un apuntador a la entrada correspondiente de la tabla de transacciones.
3. *Uba* o UNDO *Block Adress:* Se corresponde con el campo *Uba* de la tabla de transacciones y debería tener el mismo valor.
4. *Flag:* Es una bandera que identifica el estado de la transacción. Los estados más importantes son:
    1. "`----`": Significa *Activa* o bien que es una entrada de la ITL que todavía no ha sido utilizada (en este último caso *Xid* debería valer cero).
    2. "`C---`": Significa cometida
5. *Lck:* Es un contador con el número de filas bloqueadas por esa transacción en esa página.
6. *Scn/Fsc:* Es un campo cuyo significado varía en función del valor de *flag*. Si el estado es "C" representaría el SCN del momento en que fue cometida la transacción (i.e.; el ts_fin en la sección 6.2).

Inicialmente la ITL comienza con dos entradas (si bien, si se intuye un elevado nivel de concurrencia sobre una tabla, se puede especificar un tamaño inicial mayor de la ITL con el parámetro INITRANS de la lista de atributos físicos del CREATE/ALTER TABLE de *Oracle*[19]), y si necesita más trata de reutilizar alguna entrada correspondiente a una transacción terminada (las entradas de la ITL cometidas son reutilizadas por orden de antigüedad de su SCN ([Lewis 2011] capítulo 3, sección *Concurrent Action*). En caso de que no encuentre ninguna transacción terminada, la lista crece dinámicamente hasta un tamaño máximo que se puede calcular teniendo en cuenta que cada entrada de la ITL ocupa 24 bytes y que la ITL no puede exceder el 50% del tamaño del bloque, aunque rebasar ese tamaño es muy improbable[20]. Por ejemplo, en *Oracle* 10g y 11g para bloques de 8Kb el tamaño máximo de la ITL son 169 entradas ([Lewis 2011] sección *The Interested Transaction List*). Por tanto, el tamaño de la ITL depende del tamaño de bloque especificado para esa base de datos[21].

19 Conviene especificarlo mejor en el CREATE, porque el ALTER solo lo cambia para los bloques nuevos que vaya adquiriendo el objeto; si bien, en el caso de las tablas ALTER TABLE con la clausula STORAGE MOVE realoja los bloques viejos en nuevos bloques con el nuevo INITRANS.

20 Si se diera ese improbable caso se produciría una espera (i.e; *ITL wait*) a que quedara alguna entrada de la ITL reutilizable debido a que finalice una transacción.

21 Existe otro parámetro físico llamado MAXTRANS del CREATE/ALTER TABLE para especificar el tamaño

Al reutilizar una entrada de la ITL correspondiente a una transacción terminada, los valores de esa entrada no se pierden, sino que se almacenan en el segmento UNDO ([Kyte 2010] capítulo 2, sección *Read Consitency*). Es decir, desde una entrada ITL su *Uba* nos lleva a los pasos necesarios para deshacer los cambios que produzca esa transacción, entre ellos, restaurar los valores anteriores de dicha entrada de la ITL. Aplicando este procedimiento de forma recurrente sobre una entrada restaurada de la ITL podríamos ir navegando hacia atrás en el tiempo, de transacción cometida en transacción cometida, y reconstruir el estado del bloque, no solo revirtiendo los cambios sobre las filas de la tabla, sino los propios cambios en la ITL.

Una vez se ha hecho *commit* es necesario tener un procedimiento eficiente que permita actualizar el estado de las entradas en las ITL de cada bloque que ha sido modificado por la transacción. Pero revisitar todos esos bloques y modificar en la correspondiente entrada de la ITL el *flag , SCN* etc … es costoso, especialmente en transacciones que hayan modificado cientos de bloques. Por ello, *Oracle* implementa una estrategia "perezosa" de manera que no actualiza esas entradas de la ITL (i.e.; no hace *cleanout* en la terminología *Oracle*) hasta el momento de volver a reutilizarlas con otra transacción nueva. Esto es lo que se conoce como *Delayed Block Cleanout* ([Lewis 2011] capítulo 3, sección *Delayed Block Cleanout*).

Por tanto, cuando una transacción termina

1. En la tabla de transacciones:
   1. En su entrada de la tabla de transacciones su *state* pasa a inactivo.
   2. El campo *scn* toma el valor del siguiente SCN.
2. Las ITL no las cambia, las cambiarán las siguientes transacciones que pidan entradas en la ITL.

   Nota: En realidad, desde la versión 7.3, a veces si que se cambian algunas entradas de la ITL. Esto es debido a que en un *cluster* de bases de datos (i.e.; *Oracle Parallel Server – OPS* – en tiempos de la 7.3, *Oracle Real Application Cluster – RAC* – a partir de la 9.1), cuando una de las instancias al pedir una entrada en la ITL de un bloque tiene que hacer el *Deyed Block Cleanuot*, puede que tenga que pedir a otra instancia que le pase la cabecera de un segmento UNDO para poder averiguar qué valor tiene que dar al campo *scn* (en un *cluster* cada instancia tiene un *tablespace UNDO* propio). Para pasar esa información de una instancia a otra *Oracle* necesita que la instancia que tiene la información escriba el bloque en memoria con dicha información (i.e.; el bloque con la cabecera del segmento UNDO) en disco, y que la instancia que la necesita lo lea de disco ([Lewis 2011] capítulo 3, sección *Commit SCN*), perdiéndose velocidad precisamente en el tipo de instalaciones que se suponen más críticas.

   Por esa razón, mientras se están cambiando los datos, la sesión crea en memoria una lista, que no puede exceder un determinado tamaño máximo, con los bloques que ha "tocado" la transacción. En el momento de hacer *commit* toma aquellos bloques de la lista que aun estén en memoria (para evitar lecturas de disco) y actualiza en sus ITL el valor de *flag* correspondiente a esa transacción a "-U--", así como el campo *fsc/scn* al valor correspondiente (i.e.; el mismo que acaba de escribir en la tabla de transacciones). Sin embargo, no hace la limpieza de los bloqueos causados por esa transacción, que aún quedaría pendientes de limpiar ([Lewis 2011] capítulo 3, sección *Commit Cleanout*). De esta manera, ya no es necesario que estas entradas de la ITL tengan que leer la tabla de transacciones de un segmento UNDO suyo o de cualquier otra instancia para poder averiguar el valor del campo *scn*.

El procedimiento para crear la instantánea se basa en crear una instantánea de cada bloque de la base de datos que contenga datos que esté leyendo el correspondiente comando SQL, supongamos por ejemplo, que es a causa de una SELECT.

Por tanto, para cada uno de esos bloques se crea en memoria una instantánea de ese bloque; de manera que por

máximo de la ITL, pero este límite es puramente sintáctico y lo ignora a partir de la versión 9i.

ejemplo en el caso serializable contuviera la información más reciente cometida antes de que se iniciara la transacción actual, y en el caso *read committed* contuviera la información más reciente cometida en ese instante. Para lo cual sigue estos pasos ([Lewis 2011] capítulo 3, sección *Creating Consistency*):

1. Clona el bloque en un *buffer*. Este clon se modifica en los siguientes pasos hasta conseguir la instantánea del mismo.

2. Aplica *cleanout* a todas las transacciones cometidas que lo necesiten (i.e.; que aún no estén en estado "C---"). Para ello,:

2.1. Si el *flag* tiene el valor "-U--" solo es necesario hacer limpieza de los bloqueos (i.e.; el contador de filas bloqueadas *lock* se pone a cero, así como el *lock byte* de cada una de esas filas bloqueadas). Además se pone el *flag* a "C---".

2.2. Si no, en principio basta usar su *Xid* para consultar la entrada correspondiente en la tabla de transacciones, saber si está activa o no, y si está cometida obtener la información de su SCN.
Sin embargo, podría darse el caso de que esa entrada en la tabla de transacciones hubiera sido reutilizada. En el *Xid* el campo correspondiente al *wrap#*, nos indicaría esta circunstancia. Si la tabla de transacciones contiene en la entrada marcada por el *Xid* el mismo *wrap#* que está indicando el propio *Xid*, es que la transacción sigue aún registrada en la tabla de transacciones, mientras que si en esa entrada de la tabla de transacciones nos encontramos un valor de *wrap#* superior al del *Xid*, es que ha sido reutilizada. Esto nos lleva a varias formas de actualizar (hacer *cleaout*) de las entradas de la ITL de los bloques visitados por la SELECT:

2.2.1. Si no ha sido reutilizada se pasa el estado del *flag* a "C---", el campo *fsc*/*scn* toma el valor del campo *scn* en la tabla de transacciones, el contador de filas bloqueadas *lock* se pone a cero, así como el *lock byte* de cada una de esas filas bloqueadas.

2.2.2. Si ha sido reutilizada, en principio no tenemos en la tabla de transacciones el valor de SCN para poder asignárselo al campo *fsc/scn* de la ITL. Hay dos situaciones:

2.2.2.1. Que la transacción que hace la SELECT esté en modo de aislamiento *read-committed*, en cuyo caso la información relevante no es tanto ¿cuándo ha sido cometida esa transacción?, sino simplemente, si ha sido o no cometida. Por ello, nos basta con una aproximación grosera del momento en el que se hizo el *commit* que se conoce como *Upper Bound Commit* ([Lewis 2011]capítulo 3, sección *Delayed Block Cleanout*). Esta aproximación utiliza el valor del campo *scn* en *transaction control*, que recordemos que contiene el SCN más bajo para la transacciones cometidas que actualmente están en la tabla de transacciones.
En este caso, como el SCN es aproximado, el valor del *flag* en la ITL no se pone a "C--" sino a "C-U-" (la U es de *Upper Bound Commit*), así si más adelante una transacción serializable lee el bloque puede saber que no debe tomar ese SCN como exacto, con lo que tendría averiguar el SCN exacto con el procedimiento que se detalla en el siguiente punto.

2.2.2.2. Pero si el nivel de aislamiento es serializable (o *read-only,* ver anexo 4.4) es necesario un valor exacto del SCN, porque es la marca de tiempo ts_fin que va a decidir si esa transacción hizo *commit* antes de que empezara la transacción que tiene la SELECT.
Para averiguar ese SCN exacto, sabemos que *Oracle* es capaz de retrotraer la tabla de transacciones a un punto en el tiempo deseado, que en este caso es el punto en el tiempo en el que la entrada marcada por el *Xid* tenga el valor esperado de *wrap#*.
Por tanto, en el caso serializable puede ser más costoso realizar la operación de *cleanout*.

Una vez averiguado el *scn* se actualiza en el campo *fsc/scn* de la ITL, y el campo *flag* que pasa a tomar el valor "C---".

Cualquiera que sea el nivel de aislamiento, se limpiarán los bloqueos (i.e.; el contador de filas bloqueadas *lock* se pone a cero, así como el *lock byte* de cada una de esas filas bloqueadas).

3. Se deshacen en orden cronológico inverso todos los cambios de las transacciones no cometidas (i.e.; activas) con excepción de la propia transacción que está haciendo esa SELECT (podría darse el caso de que la propia transacción que hace la lectura/SELECT, primero hiciera algún UPDATE/INSERT/DELETE y después la propia SELECT). Estas transacciones activas están en la ITL con *flag* "----", y por tanto basta seguir el *Uba* en la ITL para encontrar en forma de lista enlazada esos cambios en orden cronológico inverso, que es el que nos interesa.
4. Si hay cambios producidos por transacciones con SCN superior al SCN objetivo (i.e.; el SCN objetivo se corresponderá con el SCN del comienzo de la transacción en el caso serializable, o con el SCN de comienzo del comando DML – en este caso la SELECT – en el caso *read-committed*), se deshacen en orden cronológico inverso todos los cambios de dichas transacciones.

Nota que, debido a que los cambios en la ITL también han sido registrados en el segmento UNDO, al revertir esos cambios también se va a ir restaurando la ITL. Así, al revertir una transacción activa o una con un SCN superior al SCN objetivo, pueden aparecer antiguas entradas en la ITL con su correspondiente *scn* y *uba*, y si ese *scn* vuelve a ser superior al SCN objetivo, se utilizaría el *uba* para deshacer sus cambios, incluidos los cambios en la ITL; con lo que recurrentement podría volver a aparecer otra entrada antigua de la ITL sobre la que aplicar el mismo procedimiento hasta que el campo *scn* de esa entrada de la ITL fuese inferior al SCN objetivo. Es decir, hasta llegar a una transacción cuyos cambios hays que visibilizarlos, y ya no hay que ir más hacia atrás en el tiempo.

En cuanto al SCN objetivo (i.e.; el ts_inicio de la sección 6.2) es una información que en este esquema que implementa *Oracle,* solo es relevante para la propia transacción en curso, por eso no es necesario que esté disponible al resto de transacciones en, por ejemplo, la ITL o la tabla de transacciones; sino que es una información que guarda la sesión como propia.

En cuanto a la reutilización de las versiones obsoletas en el caso de *Oracle*, ya ha sido comentada en la sección Caso de estudio: Oracle snapshot too old.

3.2. Aislamiento de instantánea en PostgreSQL

En *PostgreSQL,* a diferencia de *Oracle,* no hay un segmento UNDO donde guardar las versiones viejas de los datos, sino que cada vez que se modifica una fila, se crea una nueva versión de la misma en un mismo área común para todos los datos. Otra diferencia es que aparentemente no hay tampoco marcas de tiempo, aunque los identificadores de las transacciones (i.e., *txid* o *transaction ids*), al ser números consecutivos, hacen el papel de marcas de tiempo.

PostgreSQL llama *tupla* a cada versión de una fila. Cada vez que insertamos una fila se genera la versión inicial, cada vez que hacemos UPDATE de la fila se genera una nueva versión; así podríamos por ejemplo hacer *n* UPDATEs sobre la misma fila generando *n+1* versiones o tuplas de la misma.

Para implementar la concurrencia, cada tupla contiene una serie de campos ocultos que son las llamadas ***"columnas del sistema"*** que, entre otras cosas, permiten saber qué transacción ha creado esa versión como consecuencia de un INSERT o UPDATE (columna del sistema *xmin)* y qué transacción ha dejado como obsoleta esa versión como consecuencia de un UPDATE o DELETE (columna del sistema *xmax*).

Por ejemplo:

1. Cuando la transacción con *txid* 100 haga un INSERT de una fila se crea una versión con xmin=100 y xmax=0

2. Cuando la transacción con *txid* 101 haga un UPDATE sobre la fila anterior
   1. La versión existente mantiene xmin=100 pero cambia xmax=101
   2. Se crea una nueva version con xmin=101 y xmax=0
3. Cuando la transacción con txid 102 haga DELETE sobre la fila anterior, la última versión mantendrá xmin=101, pero xmax cambia a 102 (además de marcarse dicha tupla como borrada).

Además existe una tabla de transacciones llamada *commit log* o *clog*, que tiene una entrada por transacción y la información sobre su estado (i.e., cometida, retrocedida o activa). *PostgreSQL* mantiene bits de estado en cada tupla que contienen información del estado de la transacción, a fin de evitar consultar *clog* constantemente. La información de las columnas del sistema se combina con la del estado de las transacciones, y sirve para decidir, según qué nivel de aislamiento qué versión de la fila devolver en el momento de crear la instantánea (ver anexo 3.2.1 para ver las 10 reglas que hay que aplicar para dirimir si una versión de fila es visible o no a una instantánea).

Las versiones obsoletas pueden limpiarse a través de un proceso conocido como *VACUUM* que el administrador puede invocar periódicamente.

Para más información sobre el aislamiento de instantánea en *PostgreSQL* ver el capítulo 5 de [Suzuki 2019].

3.2.1 Reglas de visibilidad de las versiones de fila en PostgreSQL

(Adaptado a partir del capítulo 5 de [Suzuki 2019]).

Regla 1: If Status(t_xmin) = ABORTED ⇒ Invisible

Si la transacción que crea la versión aborta (e.g.; hace *rollback*) la versión es descartada

Regla 2: If Status(t_xmin) = IN_PROGRESS ∧ t_xmin = current_txid ∧ t_xmax = INVAILD ⇒ Visible

Si la transacción que crea la versión está aún en progreso y
la transacción que crea la versión es la propia transacción actual y
además xmax es nulo/cero,:

Es que es una **inserción** o una **modificación** generada en la transacción actual; por lo que hay que hacerla visible.

Regla 3: If Status(t_xmin) = IN_PROGRESS ∧ t_xmin = current_txid ∧ t_xmax ≠ INVAILD ⇒ Invisible

Si la transacción que crea la versión está aún en progreso y
la transacción que crea la versión es la propia transacción actual y
además xmax NO es nulo/cero,:

Como es una versión creada en la transacción actual (t_xmin = current_txid) que está en progreso, la versión solo ha podido ser vista por la transacción actual, y por tanto solo ha podido ser modificada o borrada por la propia transacción actual (de hecho xmax debiera ser igual a xmin en este caso). Si la ha borrado, ya no es visible, y si la ha modificado tampoco porque habrá una versión más reciente en esa misma transacción (i.e.; la de xmax=0 (inválido)).

Regla 4: If Status(t_xmin) = IN_PROGRESS ∧ t_xmin ≠ current_txid ⇒ Invisible

Si la transacción que crea la versión está aún en progreso y
la transacción que crea la versión NO es la propia transacción actual

No es visible, por ser un cambio NO cometido de otra transacción (evitamos lectura sucia)

Regla 5: If Status(t_xmin) = COMMITTED ∧ Snapshot(t_xmin) = active ⇒ Invisible

En la terminología de *PostgreSQL, snapshot* de una transacción T en un instante t son las transacciones que estuvieron activas en ese instante t ([Suzuki 2019] sección 5.5). Es decir, las transacciones concurrentes a no tener en cuenta a la hora de formar la instantánea; lo cual significa:

- En el caso SERIALIZABLE las transacciones que estuvieron en progreso en el momento de comenzar T o que todavía no comenzaron en ese momento.
- En el caso READ COMMITTED las transacciones que estuvieron en progreso en el instante actual t en el que formamos la instantánea (e.g.; como consecuencia de ejecutar una SELECT en ese instante t), o que aún no han comenzado (es decir, se excluyen las del caso serializable que se cometieron antes del instante t).

Por tanto: **Snapshot(t_xmin) = active en el caso serializable** significa que la transacción x_min está en progreso o aún no comenzó cuando comenzó la **transacción** actual.
Snapshot(t_xmin) = active en el caso read committed significa que la transacción x_min está en progreso o aún no comenzó en el momento de generar la instantánea para la **consulta** actual.

Si la transacción que creo la versión ya está cometida y

estuvo activa al comenzar la transacción actual o después de comenzar la transacción actual
En el caso serializable no debiera poder verse para evitar una lectura no repetible
En el caso *read-committed* nota que se verían las cometidas, pues no las considerarlas activas.

Regla 6: If Status(t_xmin) = COMMITTED ∧ (t_xmax = INVALID ∨ Status(t_xmax) = ABORTED) ⇒ Visible

Si la transacción que creo la versión ya está cometida y
no la ha cambiado/borrado nadie, o en todo caso, la transacción que hizo el cambio/borrado ha abortado
Esa versión es la más actual de entre las cometidas antes de que la transacción actual empezara, luego la hacemos visible.

Regla 7:

If Status(t_xmin) = COMMITTED ∧ Status(t_xmax) = IN_PROGRESS ∧ t_xmax = current_txid ⇒ Invisible

Si la transacción que creo la versión ya está cometida y
la ha cambiado/borrado una transacción en progreso y
esa transacción es la actual
Entonces esa versión ha sido cambiada/borrada por la transacción actual; luego si ha sido cambiada hay otra versión mas reciente, y si ha sido borrada tampoco es visible.

Regla 8:

If Status(t_xmin) = COMMITTED ∧ Status(t_xmax) = IN_PROGRESS ∧ t_xmax ≠ current_txid ⇒ Visible

Si la transacción que creo la versión ya está cometida y
la ha cambiado/borrado una transacción en progreso y
esa transacción NO es la actual
Si esa transacción que hizo UPDATE/DELETE no ha terminado y no es la actual, en el caso DELETE no es efectivo el borrado por ser *in progress*, en el caso UPDATE, seria una actulización de una transacción *in progress* que no es la actual, tampoco deberíamos ver el cambio para evitar lecturas sucias, pues la transacción t_xmax podría hacer *rollback*.

Regla 9:

If Status(t_xmin) = COMMITTED ∧ Status(t_xmax) = COMMITTED ∧ Snapshot(t_xmax) = active ⇒ Visible

Snapshot(t_xmax) = active en el caso seralizable significa que la transacción x_max está en progreso o aún no comenzó cuando comenzó la transacción actual.
Snapshot(t_xmax) = active en el caso read committed significa que la transacción x_max está en progreso o aún no comenzó en el momento de generar la instantánea para la consulta actual.

Si la transacción que creo la versión ya está cometida y
posteriormente (en el instante xmax) la ha cambiado/borrado una transacción cometida, la cual:

- En el caso serializable no había terminado cuando comenzó la transacción actual
  => La modificación o borrado posterior es una transacción aún no cometida en el momento de comenzar la transacción actual
- En el caso *read-committed* no había terminado cuando comenzó la consulta actual
  => La modificación o borrado posterior es una transacción aún no cometida en el momento de comenzar la consulta actual

Entonces, la modificación o borrado en el instante xmax no debería de ser aún efectivo para la transacción que efectúa la consulta, luego la versión en xmin es visible.

Regla 10:

If Status(t_xmin) = COMMITTED ∧ Status(t_xmax) = COMMITTED ∧ Snapshot(t_xmax) ≠ active ⇒

Invisible

Snapshot(t_xmax) = active en el caso seralizable significa que la transacción x_max está en progreso o aún no comenzó cuando comenzó la transacción actual.
Snapshot(t_xmax) = active en el caso read committed significa que la transacción x_max está en progreso o aún no comenzó en el momento de generar la instantánea para la consulta actual.
Entonces, la modificación o borrado en el instante xmax no debería de ser aún efectivo, luego la versión en xmin es visible.

Si la transacción que creo la versión ya está cometida y
posteriormente (en el instante xmax) la ha cambiado/borrado una transacción cometida, pero que:

- En el caso serializable sí había terminado cuando comenzó la transacción actual
  => La modificación o borrado posterior es una transacción cometida en el momento de comenzar la transacción actual
- En el caso *read-committed* sí había terminado cuando comenzó la consulta actual
  => La modificación o borrado posterior es una transacción cometida en el momento de comenzar la

consulta actual

Entonces, la modificación o borrado en el instante xmax debería de ser efectivo para la transacción que efectúa la consulta, luego la versión en xmin es invisible.

3.3. Aislamiento de instantánea en SQL-Server

SQL-Server, por defecto implementa la concurrencia con 2PL, pero es posible forzar a que trabaje con aislamiento de instantánea. Para ello, el primer paso es hacer

```
ALTER DATABASE MyDatabase SET ALLOW_SNAPSHOT_ISOLATION ON;
```

ALLOW_SNAPSHOT_ISOLATION habilita dentro de la base de datos temporal ***tempdb***, un conjunto de páginas de la base de datos conocido como *version store*, donde se guardan las versiones de las filas originales (similar al segmento UNDO de *Oracle*, con la diferencia que en *Oracle* se guardan vectores de cambio, y en *SQL-Server* se guardan por entero las versiones antiguas de cada fila, como hacía *PostgreSQL*).

Si se quiere utilizar aislamiento READ COMMITTED basado en versiones, estableceremos el nivel de aislamiento como SNAPSHOT:

```
SET TRANSACTION ISOLATION LEVEL SNAPSHOT;
```

(En *SQL-Server* si hacemos simplemente SET TRANSACTION ISOLATION LEVEL SERIALIZABLE se utiliza el protocolo de bloqueos 2PL, en lugar de las versiones).

Si por el contrario queremos utilizar versiones para implementar aislamiento READ COMMITTED, en principio, no basta con configurar la transacción con:

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
```

pues por defecto, *SQL-Server* intenta implementar también este nivel de aislamiento mediante bloqueos (ver sección 5.1.4), por lo que hay que cambiar esta estrategia de implementación del nivel READ_COMMITTED a nivel de toda la instancia de base de datos con:

```
ALTER DATABASE MyDatabase SET READ_COMMITTED_SNAPSHOT ON;
```

A partir de aquí el funcionamiento interno recuerda a *PostgreSQL.* Por un lado, en *SQL-Server* las transacciones se pueden identifican por un número secuencial *(XSN=Transaction Sequence Number*), que guarda un orden cronológico de cuando empiezan las transacciones, como en *PostgreSQL.* Por cada operación DELETE/UPDATE se guarda la versión original de cada fila en el área *version store* de *tempdb*. Estas versiones tienen 14 bits adicionales pues:

- En cada versión de fila se guarda el *XSN* que creó la versión (similar a *xmin* en *PostgreSQL).*
- Un puntero a la versión anterior de esa fila. En la versión inicial, la generada por el INSERT, ese puntero vale cero. Este puntero recuerda mucho a *xmax* en *PostgreSQL,* ya que navegando hacia la versión anterior y obteniendo su *XSN* obtendríamos el número de transacción que provocó la modificación (UPDATE/DELETE) de una versión de fila, que es en definitiva el valor que tomaba *xmax* en *PostgreSQL.* Una diferencia con *PostgreSQL* es que si una misma transacción hace *n* UPDATEs de la misma fila en *PostgreSQL,* estas actualizaciones generan *n* versiones de fila, mientras que *SQL-Server* ser genera una única versión cuyo contenido es el más reciente (el del último UPDATE de esa transacción), lo cual permite eliminar prematuramente versiones que son innecesarias ([Bolton et al. 2010] capítulo 6).

Hasta donde sabemos, el algoritmo que implementa *SQL-Server* para construir las instantáneas no aparece publicado. Sin embargo, es fácil intuir que no debería ser muy distinto al de *PosgreSQL*, en tanto mantiene estrategias e informaciones similares en el manejo de las versiones. Lógicamente, *SQL-Server* tiene que mantener alguna estructura de memoria en el gestor de concurrencia con las transacciones en curso y su estado, que permita entre otras cosas, reconstruir la instantánea recuperando las versiones adecuadas en cada consulta.

Un recolector de basura que se dispara aproximadamente cada minuto eliminando definitivamente aquellas versiones

que tienen un *XSN* tan bajo que ya no pueden ser utilizadas nunca más.
Para más información de cómo funciona el aislamiento de instantánea en *SQL-Server*, ver la sección *Optimistic Concurrency,* pgs 240- 251, en [Bolton et al. 2010] capítulo 6.

4. Cuestiones avanzadas sobre Oracle

4.1. Más sobre el comienzo y fin de las transacciones

Nota 1: Existe alguna controversia sobre si las operaciones DDL provocan el fin de la transacción en curso con *commit*. En [Oracle 2015] capítulo 10-3 da a entender que una sentencia DDL provoca el comienzo de una transacción. Pero las pruebas que hemos hecho con Oracle 11 son conformes a lo que, por el contrario, podemos encontrar en [Oracle 2014] capítulo 3, sección *"How to Begin and End Transactions"* o [Kyte 2010] *"DDL and Atomicity",* donde dice que los comandos DDL nunca forman parte de una transacción, y que para ello, Oracle hace un *commit* implícito anterior a cada comando DDL si se los encuentra a lo largo de una sesión.

Nota 2: Para experimentar por ti mismo cuando comienzan o terminan las transacciones en Oracle puedes probar a abrir 2 sesiones concurrentes. Una sesión por ejemplo con el usuario HR y otra con el administrador de base de datos SYS. En la sesión HR puedes ejecutar comandos SQL que sospeches hacen empezar o terminar una transacción. Cada vez que ejecutes uno de esos comandos puedes hacer desde la sesión SYS:

```
select XID, start_time
from v$transaction;

select current_timestamp from dual;
```

v$transacion te mostrará un listado en el que cada fila representa una transacción que estará activa. Es probable que determinadas operaciones "al principio" no se reflejen en v$*transaction* (e.g., un comienzo de transacción con una SELECT o con SET TRANSACTION), pero tan pronto ejecutes un INSERT/DELETE/UPDATE dentro de la transacción, veras en *v$transaction* la fila correspondiente a esa transacción identificada por el campo *XID* (i.e., *transaction ID*). Si te fijas en el campo *start_time* y lo comparas con la salida de la segunda SELECT, verás que el comienzo real de la transacción en los casos de que SELECT o SET TRANSACTION fueran el primer comando de la transacción, no se corresponde con el *current_timestamp* tras ese INSERT/DELETE/UPDATE, sino tras ese SELECT/SET TRANSACTION confirmando lo explicado en la sección *III.1.1Comienzo y fin de una transacción al detalle*.
Si haces nuevas operaciones y la transacción con ese *XID* sigue mostrándose en *V$TRANSACTION*, es que esas operaciones no cierran la transacción. Si por el contrario la fila con ese *XID* desaparece es que esas operaciones si que cierran la transacción.

4.2. Organización física y lógica de los datos en Oracle

Oracle organiza de manera lógica la información en ***tablespaces***. Cada objeto de la base de datos, por ejemplo cada tabla, pertenece a un único *tablespace*. En la consulta 1 de la Ilustración 27 se puede ver que en el caso de una instalación por defecto de *Oracle Express 11g* hay cinco *tablespaces*:

- **SYSTEM** y **SYSAUX**: SYSTEM debe estar presente en toda base de datos *Oracle* y contiene el diccionario de datos o metabase. SYSAUX es un *tablespace* auxiliar del *tablespace* SYSTEM. A partir de

la versión 10g también es obligatorio para toda base de datos *Oracle.*

- **UNDOTBS1**: Es el *tablespace* ***UNDO*** donde se guarda la información necesaria para hacer *rollback* así como para que el aislamiento de instantánea sea capaz de leer una versión vieja de un dato.
- **TEMP**: Es un *tablespace* temporal. Se utiliza para obtener resultados parciales de las consultas SQL. Por ejemplo, para guardar el resultado de un filtrado por un WHERE antes de aplicar un JOIN, o para guardar el resultado de una consulta antes de aplicar el GROUP BY, o para guardar un resultado parcialmente ordenado durante una ordenación, etc ...
- **USERS**: Es el que se utiliza para guardar los objetos persistentes de los usuarios. El caso más representativo de objeto que se almacena en este *tablespace* son las tablas, pero también contiene los índices.

Los *tablespaces* organizan su información físicamente en **segmentos**. Un segmento puede estar almacenado en uno o varios **ficheros de datos**, y un mismo fichero de datos puede contener en su totalidad, o en parte, varios segmentos. En la consulta 2 de la Ilustración 27 se muestra para la misma instalación, que cada *tablespace* se corresponde con un fichero y viceversa.

Nota a la ilustración: Aunque el resultado de la consulta es real (i.e.; es el que ha obtenido el profesor de su instalación), se puede observar que en la segunda consulta *UNDOTBS1* aparece en el fichero *SYSAUX.DBF* y que *SYSAUX* aparece en *UNDOTBS1.DBF*. Es decir, ambos *tablespaces* han intercambiado entre si el fichero en el que deberían estar, probablemente por algún error del asistente de instalación,

```
select tablespace_name, block_size, contents,
segment_space_management
from dba_tablespaces;
```

```
TABLESPACE_NAME                BLOCK_SIZE CONTENTS  SEGMENT_SPACE_MANAGEMENT
------------------------------ ---------- --------- ------------------------
SYSTEM                               8192 PERMANENT MANUAL
SYSAUX                               8192 PERMANENT AUTO
UNDOTBS1                             8192 UNDO      MANUAL
TEMP                                 8192 TEMPORARY MANUAL
USERS                                8192 PERMANENT AUTO
```

```
select substr(file_name, 1, 50), file_id, tablespace_name
from dba_data_files;
```

```
SUBSTR(FILE_NAME,1,50)                             FILE_ID TABLESPACE_NAME
-------------------------------------------------- ---------- --------------------
C:\ORACLEXE\APP\ORACLE\ORADATA\XE\USERS.DBF                  4 USERS
C:\ORACLEXE\APP\ORACLE\ORADATA\XE\SYSAUX.DBF                3 UNDOTBS1
C:\ORACLEXE\APP\ORACLE\ORADATA\XE\UNDOTBS1.DBF              2 SYSAUX
C:\ORACLEXE\APP\ORACLE\ORADATA\XE\SYSTEM.DBF                1 SYSTEM
```

```
select TABLESPACE_NAME, SEGMENT_TYPE, count(*), sum(extents)
from dba_segments
group by TABLESPACE_NAME, SEGMENT_TYPE
order by TABLESPACE_NAME, SEGMENT_TYPE;
```

```
TABLESPACE_NAME                SEGMENT_TYPE         COUNT(*) SUM(EXTENTS)
------------------------------ ------------------ ---------- ------------
SYSAUX                         CLUSTER                     1           18
SYSAUX                         INDEX                    3148         4451
SYSAUX                         INDEX PARTITION           106          107
SYSAUX                         LOB PARTITION               1            1
SYSAUX                         LOBINDEX                  936          940
SYSAUX                         LOBSEGMENT                936         1528
SYSAUX                         NESTED TABLE               24           24
SYSAUX                         TABLE                    1335         2400
SYSAUX                         TABLE PARTITION            90           92
SYSAUX                         TABLE SUBPARTITION         32           32
SYSTEM                         CLUSTER                     9          101
SYSTEM                         INDEX                     743         1249
SYSTEM                         LOBINDEX                  104          104
SYSTEM                         LOBSEGMENT                104          141
SYSTEM                         NESTED TABLE               10           10
SYSTEM                         ROLLBACK                    1            6
SYSTEM                         TABLE                     583         1100
UNDOTBS1                       TYPE2 UNDO                 10           47
USERS                          INDEX                     800          801
USERS                          TABLE                     783          783

20 filas seleccionadas
```

Ilustración 27: Tablespaces de la instalación Oracle Express 11g. En la segunda consulta se muestran los ficheros en los que reside cada tablespace. La tercera consulta, además calcula cuantos segmentos de cada tipo tiene cada tablespace y cuantas extensiones en total hay en cada tablespace.

Un segmento es una agrupación de **extensiones**. Cada extensión representa un conjunto de páginas o **bloques** de la base de datos contiguas (ver Ilustración 28). El que sean contiguas es importante para minimizar los tiempos de latencia debidos al movimiento de la cabeza del disco duro, ya que como normalmente se hacen operaciones sobre datos en el mismo *tablespace*, conviene tener una forma de forzar a que los datos de ese *tablespace* estén en disco en bloques contiguos.

Una tabla, al ser un objeto,ha de estar en un solo *tablespace*,. Aunque no está reflejado en la Ilustración 29, cada tabla solo tiene bloques *Oracle* de un único *segmento,* de manera que sus bloques están así más o menos contiguos.

Cada vez que se una extensión tiene todos sus bloques llenos de datos (i.e., la extensión está llena), *Oracle* reserva "de golpe" todos los bloques contiguos necesarios para completar una nueva extensión

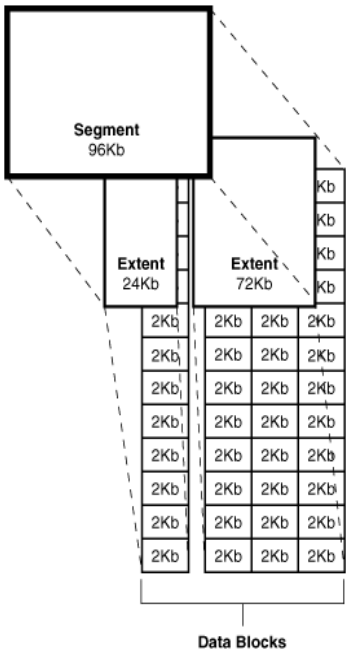


Ilustración 28: Relación entre segmentos, extensiones y bloques de datos (Es casual que el tamaño de los bloques sea 2Kb). Fuente: https://docs.oracle.com/cd/B19306_01/server.102/b14220/logical.htm.

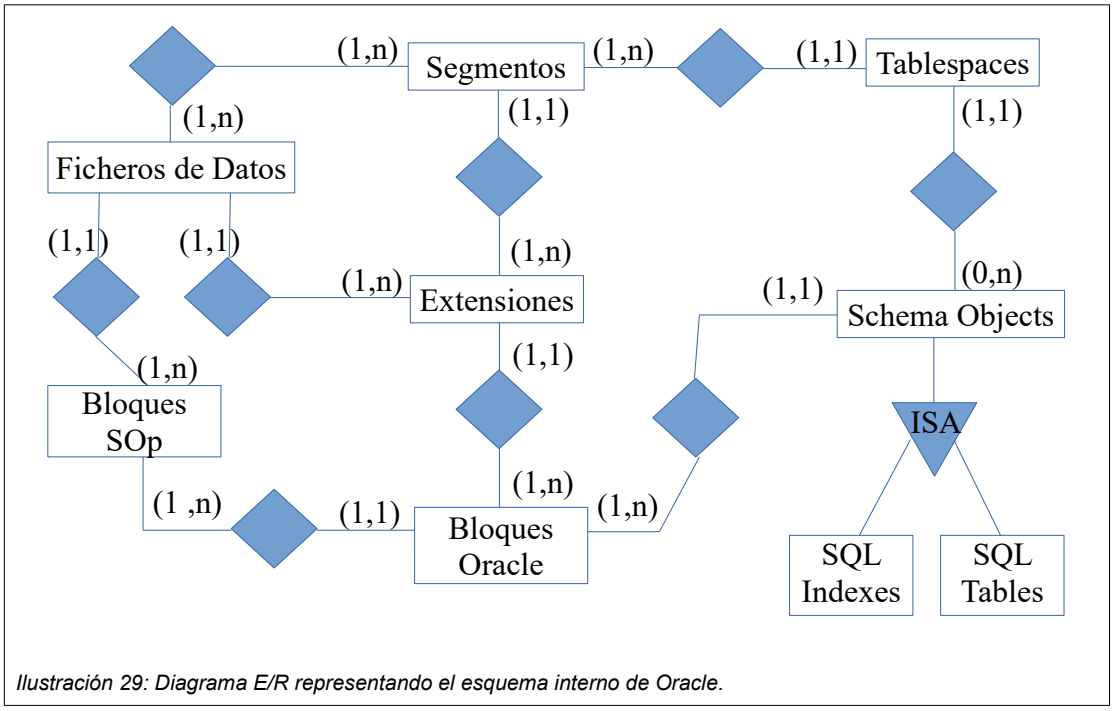


Ilustración 29: Diagrama E/R representando el esquema interno de Oracle.

En la consulta 3 de la Ilustración 27 se muestran para cada *tablespace* cuantos segmentos de cada tipo y extensiones tiene una instalación de ejemplo de *Oracle Express 11g* a modo meramente ilustrativo.

El diagrama E/R de la Ilustración 29 muestra a modo de resumen la relación existente entre todos los elementos presentados en esta sección. Los objetos del esquema de la base de datos (*Schema Objects* en la ilustración) solo están en los *tablespaces* de usuario, y en nuestro ejemplo solo hay uno, que es el *tablespace USERS.* En una situación real podría haber, por ejemplo, dos *tablespaces* de usuario; por ejemplo uno para los desarrolladores y otro para datos

de explotación. Si un bloque de disco contiene información de objetos del esquema, solo puede contenerlo de un solo objeto. Es decir, en el mismo bloque de datos no puede haber, por ejemplo, datos de dos tablas, o de una tabla y un índice simultáneamente, etc …

> ***Nota avanzada***: Podría debatirse que la excepción a esta regla son los *clusters*, que son "a grandes rasgos" objetos del esquema de base de datos que permiten almacenar en disco directamente el resultado de un *join* para que tarde menos en procesarlo. Por ejemplo, una factura con todas sus líneas de factura. Bajo la perspectiva de *Oracle,* los *clusters* son objetos del esquema lógico por si mismos y no violan esta regla. Sin embargo, desde el punto de vista "puro" de SQL, si que da la sensación de que en el fondo el mismo bloque guarda información de varios objetos, esto es, de varias tablas.

La Ilustración 29 solo muestra como objetos del esquema las tablas y los índices, pero hay mas, como por ejemplo los *clusters*, de la nota anterior.

4.3. Notas sobre los segmentos UNDO de Oracle

Los segmentos del *tablespace* UNDO guardan la información de las versiones viejas de los datos, lo cual permite tanto hacer *rollback* como construir la instantánea. Los segmentos UNDO[22] en versiones anteriores a la 9i eran conocidos como *ROLLBACK SEGMENTS.* Con la versión 9i Oracle los gestiona automáticamente por defecto, mientras que en versiones anteriores solo podían gestionarse en modo manual.

Como todo *segmento*, los segmentos UNDO son una agrupación de *bloques* de la base de datos, que a su vez pueden agrupar varios bloques de disco. Los segmentos de *Oracle* se agrupan en un *TABLESPACE.* Para más información acerca de los conceptos de *segmento* y *tablespace,* ver el anexo *4.2Organización física y lógica de los datos en Oracle*.

Cada página de un segmento UNDO de *Oracle* solo contiene información sobre las versiones viejas de los datos de a lo sumo una transacción "activa", pero puede contener además informaciones de otras transacciones ya terminadas. En una instancia de base de datos Oracle debería haber siempre un TABLESPACE UNDO activo que contendrá varios segmentos UNDO.

> **Nota 1:** Si no lo hubiera, *Oracle* tratará de alojar los segmentos UNDO en el tablespace SYSTEM, que no es lo ideal, y además generaría esta alerta en el log: "*ORA-01552: cannot use system rollback segment for non-system tablespace*").
>
> **Nota 2:** En un *cluster* de máquinas compartiendo el mismo SGBD (i.e., Oracle RAC – *Real Application Cluster*) cada instancia o máquina tiene su propio segmento UNDO activo ([Murali 2004] sección 4.4).

En las versiones actuales los segmentos UNDO son gestionados de manera automática por *Oracle* (AUM *Automatic Undo Management*), y se desaconseja el uso de la gestión manual. La AUM permite que la instancia ponga fuera de linea o en línea segmentos UNDO, hacerlos crecer o encoger, mover extensiones disponibles de un segmento a otro o incluso crear y destruir segmentos UNDO. Anteriormente los administradores tenían que monitorizar la base de datos y hacer estas acciones manualmente cuando estimaba oportuno, por ejemplo para evitar que unos segmentos crecieran mucho más que otros, lo que no es óptimo; o para redimensionarlos si no tenían espacio suficiente en disco, o si tenían demasiado espacio ([Lewis 2011] capítulo 3, sección *Transaction Table Rollback*). La única cuestión a vigilar con la AUM es el parámetro UNDO_RETENTION (ver sección "Caso de estudio: Oracle snapshot too old") que controla el tiempo mínimo que hay que conservar la información de transacciones cometidas en el *tablespace* UNDO. Si establecemos un UNDO_RETENTION grande, necesitaremos mucho espacio UNDO, lo cual habrá que tenerlo en cuenta al dimensionar el tamaño máximo del *tablespace* UNDO y/o el espacio libre en disco; pues podría

22 Traducidos como *segmentos de anulación* en [Connolly & Begg. 2005] sección 20.5.2.

disminuir demasiado, ya que es normal y recomendable que el *tablespace* tenga activado el modo *AUTOEXTEND*, el cual permite a *Oracle* que haga crecer automáticamente los ficheros del *tablespace* si necesita más espacio/extensiones.

La información que contienen los bloques de un segmento UNDO consisten en unas pequeñas instrucciones que sirven para restaurar las versiones viejas de las filas cambiadas. Estas instrucciones que sirven para restaurar por ejemplo el valor de un campo, se conocen como *Oracle Change Vectors.* También se utilizan en el REDO LOG, pero en este caso como instrucciones para restaurar los valores nuevos en caso de caída del SGBD.

4.4. Transacciones de solo lectura

El comando SET TRANSACTION de Oracle permite hacer que una transacción solo admita sentencias DML de tipo SELECT[23]. Para especificar una transacción como de solo lectura haremos

```
SET TRANSACTION READ ONLY;
```

al principio de la misma; y tiene los siguientes efectos:

1. Devuelve el error ORA-01456 si intentamos hacer un INSERT, DELETE o UPDATE en dicha transacción.
2. Cada vez que ejecutemos una SELECT en dicha transacción, se verá la base de datos en el mismo estado que estaría al principio de la misma, con independencia de las modificaciones, cometidas o no, que se hayan efectuado desde otras sesiones concurrentes; con lo que en la práctica la transacción transcurre en un nivel de aislamiento serializable.

   Importante: Por tanto READ ONLY, no es un nivel de aislamiento en si mismo. De hecho, recuerda que para especificar un nivel de aislamiento se utiliza:

   ```
   SET TRANSACTION ISOLATION LEVEL {READ COMMITTED|SERIALIZABLE};
   ```

   de hecho, no se puede ejecutar *SET TRANSACTION ISOLATION LEVEL* una vez la transacción está configurada como READ ONLY.

La utilidad de este tipo de transacciones READ ONLY esta relacionada con transacciones en las que se ejecutan una o varias consultas sobre múltiples tablas que:

1. Necesiten bastante tiempo para ser resueltas, por ejemplo decenas de minutos; y
2. Esas múltiples tablas sospechemos que pueden ser cambiadas de manera inconsistente; por ejemplo, porque en lugar de utilizar transacciones para mantener la consistencia, nos conste que puedan estar siendo actualizadas con sesiones con *autocommit;* y además
3. Se necesite que esas consultas trabajen con un estado consistente de la base de datos, pese al riesgo de que las tablas no estén siendo actualizadas de manera consistente (i.e., nada que no se pudiera haber hecho también con un nivel de aislamiento serializable).

¿Qué interés tiene utilizar READ ONLY frente a SERIALIZABLE?. Prácticamente ninguno. La única diferencia es que READ ONLY prohíbe hacer altas, bajas y modificaciones. READ ONLY surgió en *Oracle* antes que SERIALIZABLE que no fue añadida hasta la versión Oracle 7.3.3 (1997), y parece que lo han mantenido únicamente por compatibilidad retroactiva. Otra diferencia muy sutil es que las transacciones READ ONLY no tienen asignadas un segmento UNDO puesto que no hacen modificaciones, mientras que una transacción SERIALIZABLE si lo tendría.

Nota avanzada: En contraposición a "*SET TRANSACTION READ ONLY*", existe "*SET TRANSACTION READ WRITE*", que permite que la transacción tenga tanto operaciones de lectura como de escritura. "*SET*

23 Para ser más correctos, en *Oracle* se pueden utilizar SELECTs para hacer modificaciones y borrados mediante el comando SELECT FOR UPDATE introducido en la sección "Bloqueo Pesimista". Pues bien, las transacciones de solo lectura tampoco admiten SELECTs FOR UPDATE,

TRANSACTION READ WRITE" es el comportamiento por defecto de *Oracle*. Dado que *Oracle* solo permite poner un único *SET TRANSACTION* al principio de la transacción, cuando se hace "*SET TRANSACTION READ WRITE*" *Oracle* toma el nivel de aislamiento que tenga la sesión en ese momento. Es decir, tomará nivel de aislamiento READ COMMITTED a no ser que hayamos hemos hecho "*ALTER SESSION SET ISOLATION_LEVEL=SERIALIZABLE*", en cuyo caso naturalmente tomará el nivel SERIALIZABLE.

4.5. Estoy obteniendo el error ORA-08177 sin motivo justificado aparente

El error *ORA-08177* puede ocurrir eventualmente aún cuando se esté intentando escribir sobre filas distintas. Cuando ocurre esto es porque ambas filas están alojadas en el mismo bloque de datos. En la cabecera de cada bloque *Oracle* registra información de qué transacciones están o bien bloqueando filas de ese bloque en ese momento, o bien esperando en ese momento a causa de que haya filas bloqueadas en ese bloque. Esta información se registra en forma de una lista (*ITL* o *Interested Transaction List* (ver nexo 3.1), que crece dinámicamente a medida que aumenta el número de transacciones que mantienen o piden bloqueos en ese bloque. Cada bloque de datos de *Oracle* se corresponde con una única tabla. El tamaño inicial de la lista se puede especificar con el parámetro de almacenamiento INITRANS del CREATE TABLE de *Oracle.*

Por ejemplo:

```
create table calificaciones (
  nombre varchar(20) primary key,
  nota   numeric(2)
) INITRANS 3;
```

establecería el valor 3 para ese parámetro, haciendo que el tamaño inicial de la ITL para los bloques de esa tabla fuese 3. La lista ITL tiene un tamaño inicial por defecto de 2; lo que significa que la lista solo puede alojar información de 2 transacciones antes de que comience a crecer dinámicamente.

Pues bien, Oracle **requiere como mínimo un valor para INITRANS de 3** cuando el modo de aislamiento es **SERIALIZABLE** (ver sección 5.2 "*How Oracle Database Processes SQL Statements"* en [Oracle 2003]). Si no se da ese valor igual o mayor que 3 a INITRANS el error *ORA-08177* puede aparecer, por ejemplo, en un escenario como el siguiente:

1. Dos transacciones T1 y T2 están intentando actualizar concurrentemente filas distintas del mismo bloque. Al menos T1 está en modo SERIALIZABLE.
2. T1 comienza la transacción y consigue actualizar 1 fila, T2 consigue también, acto seguido, actualizar otra fila del mismo bloque que la que ha actualizado T1.
3. Ahora T1 intenta actualizar otra fila distinta de las dos anteriores de ese mismo bloque, pero no lo consigue devolviendo *Oracle* inmediatamente el error *ORA-08177* en T1 sin que ninguna de las dos transacciones haya terminado aún.

Este comportamiento es chocante dado que:

1. La ITL solo necesita una entrada por transacción y tiene ya 2 entradas para las 2 transacciones, y
2. La ITL es una lista dinámica, debería de ser capaz de crecer si necesitara crecer.

La causa última por qué se necesita ampliar ese tamaño de INITRANS aún cuando aparentemente es innecesario, no aparece documentada en ninguna parte, que nosotros sepamos.

Curiosidad: Si creamos una tabla sin cambiar el valor por defecto de INITRANS y preguntamos al diccionario de *Oracle* por las propiedades de la tabla, nos dirá que INITRANS es 1. Sin embargo, si hacemos un volcado de una página de disco cualquiera de la tabla recién inicializada con una transacción

que inserte unas pocas filas de prueba, veremos que es 2, como anteriormente hemos dicho. Esta incongruencia también está reportada en la literatura ([Lewis 2011] capítulo 3, sección *The Interested Transaction List*).

5. Consideraciones sobre la implementación de 2PL

En el ejemplo de la Ilustración 7 descrito en la sección III.5.1.2, en el paso 5 se T4 entra a ejecutarse por delante de T3 que estaba en espera, porque en aplicación "literal" del algoritmo de la Ilustración 3, T3 no puede aun obtener su bloqueo, por ser de escritura, pero T4 si que puede.

Sin embargo, esta no es la forma habitual en la que se comportan los sistemas comerciales, pues tratan de evitar el problema del "***bloqueo indefinido***" [Connolly & Begg. 2005], pg 540 al final de la sección 20.2.3 *Métodos de Bloqueo,* subsección *Bloqueo de dos fases (2PL)*:

> "También es posible que las transacciones queden en un **bloqueo indefinido**, es decir, que se queden en un estado de espera indefinidamente, incapaces de adquirir un nuevo bloqueo, aunque el propio SGBD no esté experimentando una situación de interbloqueo. Esto puede suceder si el algoritmo de espera de las transacciones no es equitativo y no toma en cuenta el tiempo que las transacciones han estado esperando. Para evitar bloqueos indefinidos, debe de utilizarse un sistema de prioridades en el que la prioridad de una transacción vaya aumentando a medida que lo hace el tiempo de espera; por ejemplo, puede utilizarse una cola de tipo *el primero en llegar es el primero en ser servido* para las transacciones en espera"

Es decir, después de entrar la transacción T4 para leer el dato por delante de T3, podrían entrar más y más transacciones de lectura de distintas sesiones y T3 seguir en espera en ese denominado *bloqueo indefinido*.

Como dice [Connolly & Begg. 2005], la solución más habitual es encolar las peticiones de bloqueos creando una cola por cada fila que esté bloqueada. En la cabeza de la cola estaría la transacción (o transacciones si son bloqueos de lectura) que han adquirido el bloqueo. **Antes de conceder el bloqueo a una transacción habría de verificarse que no hay ninguna antes de ella en la cola esperando**, de esta manera T3 se ejecutaría antes que T4 por haber llegado antes a esa cola.

Esta misma idea de encolar las peticiones de bloqueo está presente en otras referencias de la literatura cuando se adentran en la implementación que hacen los SGBDs del protocolo 2PL. Por ejemplo:[Bernstein et al. 1987], pg 60, sección "*3.6. Implementation Issues*", o [Silberschatz et al. 2015], pg 312, sección "*15.1.4 Implementación de los bloqueos*". La explicación en [Silberschatz et al. 2015] es la más detallada.

SQL-Server utiliza esta técnica, evitando que en el Ejemplo 3, T4 "se cuele" de T3, con lo que finalmente se ejecutaría de la siguiente manera (***S*** representan los bloqueos de lectura y ***X*** los bloqueos de escritura, ***grant*** que el bloqueo ha sido concedido y ***wait*** que queda en espera a que se conceda):

- Paso 1: La transacción T1 pide un bloqueo S sobre la fila con id A1. Como no hay ninguna cola de bloqueos para esa fila se crea una nueva con un único elemento T1-S-grant (i.e.; T1 lo bloquea para lectura y el bloqueo está concedido). Al ser el primer elemento de la cola, obtiene el bloqueo y devuelve el resultado de la consulta.
- Paso 2: La transacción T2 pide un bloqueo S sobre la misma fila con id A1. La cola pasa a ser:
  T1-S grant → T2-S grant
  ya que al ser una petición de lectura, es compatible con el bloqueo anterior, y por ello también se marca como "*grant*". Al obtener el bloqueo, devuelve el resultado de la consulta.
- Paso 3: La transacción T3 pide un bloqueo X sobre la fila, y como hay bloqueos incompatibles con la escritura que quiere hacer T3, en este caso dos bloqueos S; se queda en espera. La cola de bloqueos para

esa fila queda así:

T1-S grant → T2-S grant → T3-X-wait

- Paso 4: La transacción T1 hace *commit* y libera su bloqueo. La cola queda así:

  T2-S grant → T3-X-wait

  Como todavía queda un bloqueo S sobre la fila, la transacción T3 sigue en espera.
- Paso 5: Comienza la transacción T4. Pide un bloqueo S, la cola queda así:

  T2-S grant → T3-X-wait → T4-S-wait

  y, a diferencia de cómo se gestionaba esta situación en el Ejemplo 3 de la la sección III.5.1.2, **T4 queda en espera porque hay una transacción en espera por delante de ella, que es T3**.
- Paso 6: T2 hace *commit* y se libera su bloqueo S. La cola queda así:

  T3-X-wait → T4-S-wait

  Como no hay ninguna transacción en curso que impida que T3 ejerza el bloqueo X sobre la fila, T3 obtiene el bloqueo:

  T3-X-grant → T4-S-wait

  y realiza la escritura. T4 sigue en espera pues ahora el X de T3 es incompatible con su petición S.
- Paso 7: De momento, no puede ejecutarse porque T4 sigue en espera estancada sin poder hacer la SELECT del paso 5.
- Paso 8: Comienza T5, pide un bloqueo de escritura sobre la fila:

  T3-X-grant → T4-S-wait → T5-X-wait

  Como ese bloqueo es incompatible con el bloqueo X de T3 se queda en espera.
- Paso 9: T3 hace *commit* y libera su bloqueo X:

  T4-S-wait → T5-X-wait

  Como estaban esperando T4 y T5 para operar con esa fila, y T4 fue la primera que esta en la cola pidiendo, en este caso, un bloqueo S, y no hay otro bloqueo incompatible por delante en la cola, T4 obtiene el bloqueo:

  T4-S-grant → T5-X-wait

  Sin embargo, T5 sigue en espera porque la escritura que quiere hacer es incompatible con el bloqueo S que está haciendo T4. T4 ejecuta la SELECT que se quedó parada en el paso 5.
- Paso 7: Ahora es cuando realmente se ejecuta el *commit* del paso 7, pues la SELECT de T4 ha finalizado. Al ejecutarse se libera el bloqueo S de T4.

  T5-X-wait
- Paso 8: Una vez terminada T4, T5 ya no encuentra por delante, en la cola, ninguna operación incompatible con su bloqueo X, por lo que obtiene el bloqueo:

  T5-X-grant

  y ejecuta el UPDATE.
- Paso 10: T5 hace *commit* y libera el bloqueo X, dejando vacía la cola de bloqueos para esa fila.

Por ello, la planificación final resultante con SQL-Server, no es la equivalente a (T1, T2, T4, T3, T5) como se sugiere en el Ejemplo 3 de la sección III.5.1.2 cuando se aplica 2PL sin tener en cuenta esta política de colas, sino (T1, T2, **T3, T4**, T5) evitando así ese problema de "bloqueo indefinido" que podría haber ocurrido con el UPDATE de T3. Nota que se ha pagado un pequeño precio frente a como se hacía en la sección III.5.1.2, pues allí T4 no quedaba esperando y si ejecutaba inmediatamente.

Licencia

Autores: Jesús Maudes & Raúl Marticorena & Mario Martínez Abad
Área de Lenguajes y Sistemas Informáticos
Departamento de Ingeniería Civil
Escuela Politécnica Superior
UNIVERSIDAD DE BURGOS
2020