

4.4. Niveles de aislamiento

Las transacciones especifican un nivel de aislamiento que define el grado en que se debe aislar una transacción de las modificaciones de recursos o datos realizadas por otras transacciones. Los niveles de aislamiento se describen en cuanto a los efectos secundarios de la simultaneidad que se permiten, como las lecturas desfasadas o ficticias.

Control de los niveles de aislamiento de transacción:

Controla si se realizan bloqueos cuando se leen los datos y qué tipos de bloqueos se solicitan.

Duración de los bloqueos de lectura.

Si una operación de lectura que hace referencia a filas modificadas por otra transacción:

Se bloquea hasta que se libera el bloqueo exclusivo de la fila.

Recupera la versión confirmada de la fila que existía en el momento en el que empezó la instrucción o la transacción.

Lee la modificación de los datos no confirmados.

El nivel de aislamiento para una sesión SQL establece el comportamiento de los bloqueos para las instrucciones SQL.

El estándar ANSI/ISO SQL define cuatro niveles de aislamiento transaccional en función de tres eventos que son permitidos o no dependiendo del nivel de aislamiento. Estos eventos son:

Lectura sucia. Las sentencias **SELECT** son ejecutadas sin realizar bloqueos, pero podría usarse una versión anterior de un registro. Por lo tanto, las lecturas **no son consistentes** al usar este nivel de aislamiento.

Lectura norepetible. Una transacción vuelve a leer datos que previamente había leído y encuentra que han sido modificados o eliminados por una transacción cursada.

Lectura fantasma. Una transacción vuelve a ejecutar una consulta, devolviendo un conjunto de registros que satisfacen una condición de búsqueda y encuentra que otros registro que satisfacen la condición han sido insertadas por otra transacción cursada.

Los niveles de aislamiento SQL son definidos basados en si ellos permiten a cada uno de los eventos definidos anteriormente. Es interesante notar que el estándar SQL no impone un esquema de cierre específico o confiere por mandato comportamientos particulares, pero más bien describe estos niveles de aislamiento en términos de estos teniendo muchos mecanismos de cierre/coincidencia, que dependen del evento de lectura.

Niveles de aislamiento: Comportamiento permitido			
Nivel de aislamiento	Lectura		
	Sucia	No repetible	Fantasma
Lectura no comprometida	Sí	Sí	Sí
Lectura comprometida	No	Sí	Sí
Lectura repetible	No	No	Sí
Secuenciable	No	No	No

Según el estándar SQL 1992, **SQL Server y MySQL** permiten todos estos niveles, **Oracle** sólo permite la lectura comprometida y secuenciable. Los niveles se pueden establecer en ambos para cada transacción. Sin embargo esto no es necesariamente cierto.


El estándar SQL trataba de establecer los niveles de aislamiento que permitirían a varios grados de consistencia para **queries** ejecutadas en cada nivel de aislamiento. Las lecturas repetibles "**REPEATABLE READ**" es el nivel de aislamiento que garantiza que un query un resultado consistente.

En la definición SQL estándar, la **lectura comprometida "READ COMMITTED"** no regresa resultados consistentes, en la **lectura no comprometida "READ UNCOMMITTED"** las sentencias **SELECT** son ejecutadas sin realizar bloqueos,

pero podría usarse una versión anterior de un registro. Por lo tanto, las lecturas no son consistentes al usar este nivel de aislamiento.

A mayor grado de aislamiento, mayor precisión, pero a costa de menor concurrencia.

```
SET [SESSION | GLOBAL] TRANSACTION ISOLATION LEVEL
    {READ UNCOMMITTED | READ COMMITTED
    | REPEATABLE READ | SERIALIZABLE}
```



Si usa tablas transaccionales (como [InnoDB](#) o [BDB](#)), puede desactivar el modo autocommit con el siguiente comando:

```
SET AUTOCOMMIT=0;
```

Si quiere deshabilitar el modo autocommit para una serie única de comandos, puede usar el comando [START TRANSACTION](#):

Si realiza un comando [ROLLBACK](#) tras actualizar una tabla no transaccional dentro de una transacción, ocurre una advertencia [ER_WARNING_NOT_COMPLETE_ROLLBACK](#). Los cambios en tablas transaccionales se deshacen, pero no los cambios en tablas no

Sintaxis de [SAVEPOINT](#) y [ROLLBACK TO SAVEPOINT](#)

```
SAVEPOINT identifier
ROLLBACK TO SAVEPOINT identifier
```

En MySQL 5.0, [InnoDB](#) soporta los comandos SQL [SAVEPOINT](#) y [ROLLBACK TO SAVEPOINT](#).

El comando [SAVEPOINT](#) crea un punto dentro de una transacción con un nombre [identifier](#). Si la transacción actual tiene un punto con el mismo nombre, el antiguo se borra y se crea el nuevo.

El comando [ROLLBACK TO SAVEPOINT](#) deshace una transacción hasta el punto nombrado. Las modificaciones que la transacción actual hace al registro tras el punto se deshacen en el rollback,

Sintaxis de [LOCK TABLES](#) y [UNLOCK TABLES](#)

```
LOCK TABLES
    tbl\_name [AS alias] {READ [LOCAL] | [LOW_PRIORITY] WRITE}
    [, tbl\_name [AS alias] {READ [LOCAL] | [LOW_PRIORITY] WRITE}] ...
UNLOCK TABLES
```

LOCK TABLES bloquea tablas para el flujo actual. Si alguna de las tablas la bloquea otro flujo, bloquea hasta que pueden adquirirse todos los bloqueos. **UNLOCK TABLES** libera cualquier bloqueo realizado por el flujo actual. Todas las tablas bloqueadas por el flujo actual se liberan implícitamente cuando el flujo realiza otro **LOCK TABLES**, o cuando la conexión con el servidor se cierra.

Un bloqueo de tabla protege sólo contra lecturas inapropiadas o escrituras de otros clientes. El cliente que tenga el bloqueo, incluso un bloqueo de lectura, puede realizar operaciones a nivel de tabla tales como **DROP TABLE**.

Tenga en cuenta lo siguiente a pesar del uso de **LOCK TABLES** con tablas transaccionales:

- **LOCK TABLES** no es una operación transaccional y hace un commit implícito de cualquier transacción activa antes de tratar de bloquear las tablas. También, comenzar una transacción (por ejemplo, con **START TRANSACTION**) realiza un **UNLOCK TABLES** implícito.
- La forma correcta de usar **LOCK TABLES** con tablas transaccionales, como **InnoDB**, es poner **AUTOCOMMIT = 0** y no llamar a **UNLOCK TABLES** hasta que hace un commit de la transacción explícitamente. Cuando llama a **LOCK TABLES**, **InnoDB** internamente realiza su propio bloqueo de tabla, y MySQL realiza su propio bloqueo de tabla. **InnoDB** libera su bloqueo de tabla en el siguiente commit, pero para que MySQL libere su bloqueo de tabla, debe llamar a **UNLOCK TABLES**. No debe tener **AUTOCOMMIT = 1**, porque entonces **InnoDB** libera su bloqueo de tabla inmediatamente tras la llamada de **LOCK TABLES**, y los deadlocks pueden ocurrir fácilmente. (Tenga en cuenta que en MySQL 5.0, no adquirimos el bloqueo de tabla **InnoDB** en absoluto si **AUTOCOMMIT=1**, para ayudar a aplicaciones antiguas a evitar deadlocks.)
- **ROLLBACK** no libera bloqueos de tablas no transaccionales de MySQL.

Para usar **LOCK TABLES** en MySQL 5.0, debe tener el permiso **LOCK TABLES** y el permiso **SELECT** para las tablas involucradas.

La razón principal para usar **LOCK TABLES** es para emular transacciones o para obtener más velocidad al actualizar tablas. Esto se explica con más detalle posteriormente.

Si un flujo obtiene un bloqueo **READ** en una tabla, ese flujo (y todos los otros) sólo pueden leer de la tabla. Si un flujo obtiene un bloqueo **WRITE** en una tabla, sólo el flujo con el bloqueo puede escribir a la tabla. El resto de flujos se bloquean hasta que se libera el bloqueo.

La diferencia entre **READ LOCAL** y **READ** es que **READ LOCAL** permite comandos **INSERT** no conflictivos (inserciones concurrentes) se ejecuten mientras se mantiene el bloqueo. Sin embargo, esto no puede usarse si va a manipular los ficheros de base de datos fuera de MySQL mientras mantiene el bloqueo. Para tablas **InnoDB**, **READ LOCAL** esencialmente no hace nada: No bloquea la tabla. Para tablas **InnoDB**, el uso de **READ LOCAL** está obsoleto ya que una **SELECT** consistente hace lo mismo, y no se necesitan bloqueos.

Cuando usa **LOCK TABLES**, debe bloquear todas las tablas que va a usar en sus consultas. Mientras los bloqueos obtenidos con un comando **LOCK TABLES** están en efecto, no puede acceder a ninguna tabla que no estuviera bloqueada por el comando. Además, no puede usar

una tabla bloqueada varias veces en una consulta --- use alias para ello. Tenga en cuenta que en este caso, debe tener un bloqueo separado para cada alias.

```
mysql> LOCK TABLE t WRITE, t AS t1 WRITE;
mysql> INSERT INTO t SELECT * FROM t;
ERROR 1100: Table 't' was not locked with LOCK TABLES
mysql> INSERT INTO t SELECT * FROM t AS t1;
```

Si sus consultas se refieren a una tabla que use un alias, debe bloquear la tabla que usa el mismo alias. No funciona bloquear la tabla sin especificar el alias:

```
mysql> LOCK TABLE t READ;
mysql> SELECT * FROM t AS myalias;
ERROR 1100: Table 'myalias' was not locked with LOCK TABLES
```

Si bloquea una tabla usando un alias, debe referirse a ella en sus consultas usando este alias:

```
mysql> LOCK TABLE t AS myalias READ;
mysql> SELECT * FROM t;
ERROR 1100: Table 't' was not locked with LOCK TABLES
mysql> SELECT * FROM t AS myalias;
```

WRITE bloquea normalmente teniendo una prioridad superior que **READ** al bloquear para asegurar que las actualizaciones se procesan en cuanto se puede. Esto significa que si un flujo obtiene un bloqueo **READ** y luego otro flujo pide un bloqueo **WRITE**, las peticiones de bloqueo **READ** posteriores esperan hasta que el flujo **WRITE** quita el bloqueo. Puede usar bloqueos **LOW_PRIORITY WRITE** para permitir a otros flujos que obtengan bloqueos **READ** mientras el flujo está en espera para el bloqueo **WRITE**. Debe usar bloqueos **LOW_PRIORITY WRITE** sólo si está seguro que habrá un momento sin flujos con bloqueos **READ**.

LOCK TABLES funciona como sigue:

1. Ordena todas las tablas a ser bloqueadas en un orden definido internamente. Desde el punto de vista del usuario, este orden es indefinido.
2. Si una tabla se bloquea con bloqueo de lectura y escritura, pone el bloqueo de escritura antes del de lectura.
3. Bloquea una tabla a la vez hasta que la sesión obtiene todos los bloqueos.

Esta política asegura un bloqueo de tablas libre de deadlocks. Sin embargo hay otros puntos que debe tener en cuenta respecto a esta política:

Si está usando un bloqueo **LOW_PRIORITY WRITE** para una tabla, sólo significa que MySQL espera para este bloqueo hasta que no haya flujos que quieren un bloqueo **READ**. Cuando el flujo ha obtenido el bloqueo **WRITE** y está esperando para obtener un bloqueo para la siguiente tabla en la lista, todos los otros flujos esperan hasta que el bloqueo **WRITE** se libera. Si esto es un problema con su aplicación, debe considerar convertir algunas de sus tablas a transaccionales.

Puede usar **KILL** para terminar un flujo que está esperando para un bloqueo de tabla..

Tenga en cuenta que *no* debe bloquear ninguna tabla que esté usando con **INSERT DELAYED** ya que en tal caso el **INSERT** lo realiza un flujo separado.

Normalmente, no tiene que bloquear tablas, ya que todos los comandos **UPDATE** son atómicos, ningún otro flujo puede interferir con ningún otro que está ejecutando comandos SQL . Hay algunos casos en que no debe bloquear tablas de ningún modo:

- Si va a ejecutar varias operaciones en un conjunto de tablas **MyISAM** , es mucho más rápido bloquear las tablas que va a usar. Bloquear tablas **MyISAM** acelera la inserción, las actualizaciones, y los borrados. Por contra, ningún flujo puede actualizar una tabla con un bloqueo **READ** (incluyendo el que tiene el bloqueo) y ningún flujo puede acceder a una tabla con un bloqueo **WRITE** distinto al que tiene el bloqueo.

La razón que algunas operaciones **MyISAM** sean más rápidas bajo **LOCK TABLES** es que MySQL no vuelca la caché de claves para la tabla bloqueada hasta que se llama a **UNLOCK TABLES**. Normalmente, la caché de claves se vuelca tras cada comando SQL.

- Si usa un motor de almacenamiento en MySQL que no soporta transacciones, debe usar **LOCK TABLES** si quiere asegurarse que ningún otro flujo se ejecute entre un **SELECT** y un **UPDATE**. El ejemplo mostrado necesita **LOCK TABLES** para ejecutarse sin problemas:

```
mysql> LOCK TABLES trans READ, customer WRITE;
mysql> SELECT SUM(value) FROM trans WHERE customer_id=some_id;
mysql> UPDATE customer
->     SET total_value=sum_from_previous_statement
->     WHERE customer_id=some_id;
mysql> UNLOCK TABLES;
```

Sin **LOCK TABLES**, es posible que otro flujo pueda insertar un nuevo registro en la tabla **trans** entre la ejecución del comando **SELECT** y **UPDATE**.

Puede evitar usar **LOCK TABLES** en varios casos usando actualizaciones relativas (**UPDATE customer SET value=value+new_value**) o la función **LAST_INSERT_ID()** ,

Puede evitar bloquear tablas en algunos casos usando las funciones de bloqueo de nivel de usuario **GET_LOCK()** y **RELEASE_LOCK()** . Estos bloqueos se guardan en una tabla hash en el servidor e implementa **pthread_mutex_lock()** y **pthread_mutex_unlock()** para alta velocidad.

Puede bloquear todas las tablas en todas las bases de datos con bloqueos de lectura con el comando **FLUSH TABLES WITH READ LOCK** . Consulte [Sección 13.5.5.2, “Sintaxis de FLUSH”](#). Esta es una forma muy conveniente para obtener copias de seguridad si tiene un sistema de ficheros como Veritas que puede obtener el estado en un punto temporal.

Nota: Si usa **ALTER TABLE** en una tabla bloqueada, puede desbloquearse.

Sintaxis de **SET TRANSACTION**

```
SET [GLOBAL | SESSION] TRANSACTION ISOLATION LEVEL  
{ READ UNCOMMITTED | READ COMMITTED | REPEATABLE READ | SERIALIZABLE }
```

Este comando prepara el nivel de aislamiento de transacción para la siguiente transacción, globalmente, o para la sesión actual.

El comportamiento por defecto de **SET TRANSACTION** es poner el nivel de aislamiento para la siguiente transacción (que no ha empezado todavía). Si usa la palabra clave **GLOBAL** el comando pone el nivel de aislamiento de transacción por defecto globalmente para todas las transacciones creadas desde ese momento. Las conexiones existentes no se ven afectadas. Necesita el permiso **SUPER** para hacerlo. Usar la palabra clave **SESSION** determina el nivel de transacción para todas las transacciones futuras realizadas en la conexión actual.

Puede inicializar el nivel de aislamiento global por defecto para **mysqld** con la opción **--transaction-isolation**