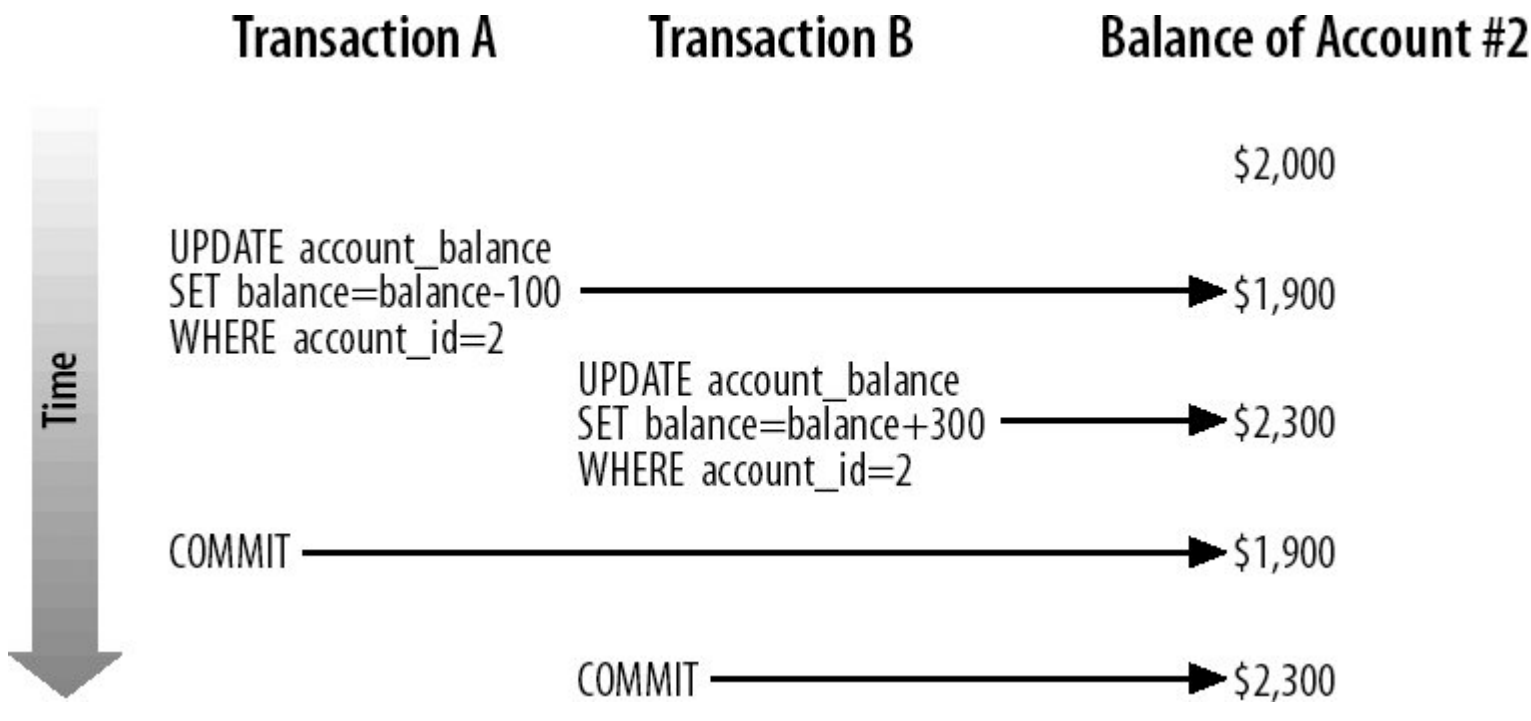


Transactions and Locks

The ACID properties of a transaction can only be implemented by restricting simultaneous changes to the database. This is achieved by placing locks on modified data. These locks persist until the transaction issues a `COMMIT` or `ROLLBACK` statement.

Without locks, a change made by one transaction could be overwritten by another transaction that executes at the same time. Consider, for example, the scenario shown in Figure 8-1, based on the `trf_rnds` procedure of Example 8-2. When two different sessions run this program for the same account number, we encounter some obvious difficulties if locks are not in place.

Figure 8-1. Illustration of a transaction without locking

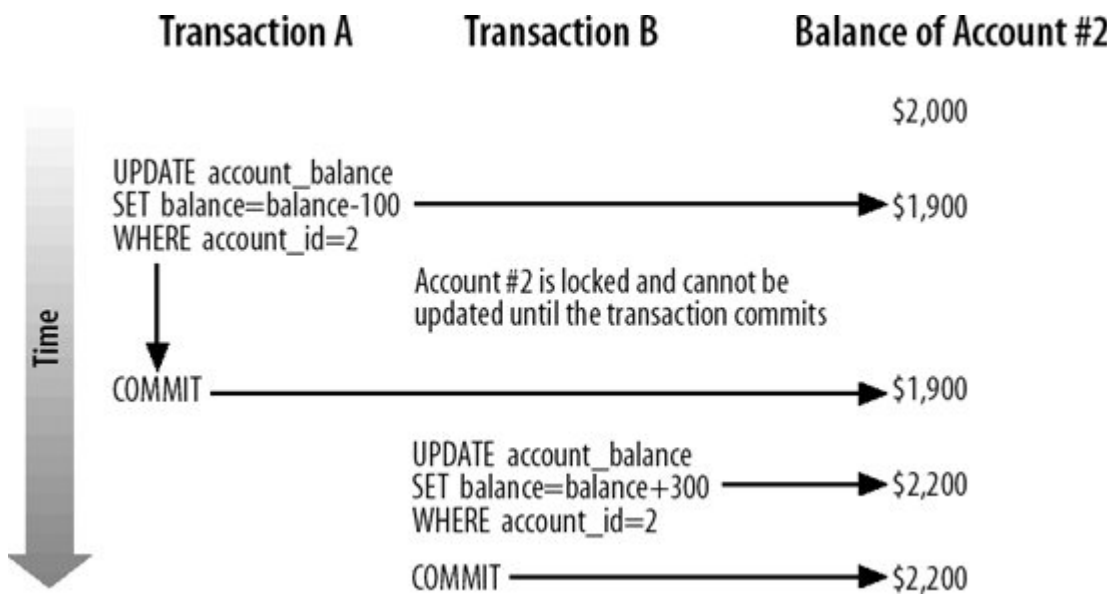


In this scenario, account 2 starts with a balance of \$2,000. Transaction A reduces the balance of the account by \$100. Before transaction A commits, transaction B increases the account value by \$300. Because transaction B cannot see the uncommitted updates made by transaction A, it increases the balance to \$2,300. Because we allowed two transactions to simultaneously modify the same row, the database is now in an inconsistent state. The end balance for the account will be the value set by whichever transaction commits last. If transaction B is the last to commit, then the owner of account 2 will have \$100 more than she should. On the other hand, if transaction A commits first, the account owner will be \$300 out of pocket!

This clearly unacceptable result is completely avoidable when locks are placed on rows that have been changed, as is illustrated in Figure 8-2.

Now, when transaction A updates account 2, the relevant row is locked and cannot be updated by another transaction. Transaction B must wait for transaction A to be committed before its update can proceed. When transaction A commits, transaction B applies its update to the modified account balance, and the integrity of the account balance is maintained.

Figure 8-2. Illustration of a transaction with locking



The downside of this locking strategy is that transaction B must wait for transaction A to complete. The more programs you have waiting for locks to clear, the less throughput your transactional system will be able to support.

MySQL/InnoDB minimizes the amount of lock contention by locking at the row level only. In our example, updates to other rows in the `ACCOUNT_BALANCE` table are able to proceed without restriction. Furthermore, with InnoDB, reads do not normally cause locks to occur, and readers do not need to wait for locks to be released before accessing data. Other transactional storage engines and other RDBMS systems may behave differently.

You can, however, place locks on rows that have only been read by using the `FOR UPDATE OF LOCK IN SHARE MODE` clause in the `SELECT` statement, and this is sometimes required to implement a specific locking strategy (see "Optimistic and Pessimistic Locking Strategies," later in this chapter).

In the following subsections we'll look at various types of locking situations, problems, and strategies.

8.4.1. Situations in Which Locks Arise

While it is possible for you to lock rows explicitly, you will generally rely on the storage engine to lock rows (or an entire table) implicitly, which it will do under the following circumstances:

- When an `UPDATE` statement is executed, all rows modified will be locked.
- An `INSERT` statement will cause any primary or unique key records to be locked. This will prevent a concurrent insert of a statement with an identical primary key.
- You can lock entire tables with the `LOCK TABLES` statement. This is not generally recommended, because it not only reduces concurrency, it operates above the storage engine layer, which might mean that any storage engine deadlock resolution mechanisms may be ineffectual.
- If you use the `FOR UPDATE OF LOCK IN SHARE MODE` clauses in a `SELECT` statement, all of the rows returned by that `SELECT` statement will be locked.

Locking rows as they are read is an important technique that we'll demonstrate in subsequent examples. To read and simultaneously lock a row, you include the `FOR UPDATE OF LOCK IN SHARE MODE` clause in the `SELECT` statement, as follows:

```
SELECT SELECT_statement options  
[FOR UPDATE] [LOCK IN SHARE MODE]
```

The two locking options differ in the following ways:

FOR UPDATE

When you use this clause, you acquire an exclusive lock on the row with the same characteristics as an `UPDATE` on that row. Only one `SELECT` statement can simultaneously hold a `FOR UPDATE` lock on a given row; other `SELECT` statements (or `DML` statements) will have to wait until the transaction ends.

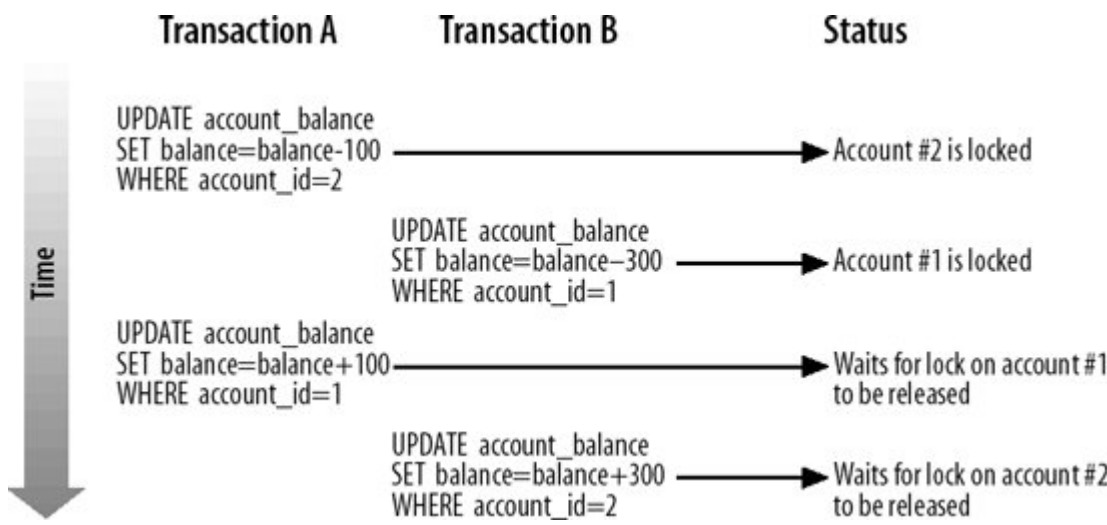
LOCK IN SHARE MODE

When you use this clause, it prevents any `DML` from being applied to the row you have locked. However, unlike `FOR UPDATE`, any number of `SHARE MODE` locks can be applied to a single row simultaneously.

8.4.2. Deadlocks

A deadlock occurs when two transactions are each waiting for the other to release a lock they each block each other, and neither can proceed. For instance, consider the situation in which one transaction attempts to transfer \$100 from account 2 to account 1. Simultaneously, another transaction attempts to transfer \$300 from account 1 to account 2. If the timing of the two transactions is sufficiently unfortunate, then each may end up waiting for the other to release a lock, resulting in a stalemate that will never end. Figure 8-3 shows the sequence of events.

Figure 8-3. Sequence of events that leads to a deadlock condition



Part I: Stored Programming Fundamentals

Introduction to MySQL Stored Programs

- [Introduction to MySQL Stored Programs](#)
- [What Is a Stored Program?](#)
- [A Quick Tour](#)
- [Resources for Developers Using Stored Programs](#)
- [Some Words of Advice to Developers](#)
- [Conclusion](#)

MySQL Stored Programming Tutorial

- [MySQL Stored Programming Tutorial](#)
- [What You Will Need](#)
- [Our First Stored Procedure](#)
- [Variables](#)
- [Parameters](#)
- [Conditional Execution](#)
- [Locks](#)
- [Dealing with Errors](#)
- [Interacting with the Database](#)
- [Calling Stored Programs from Stored Programs](#)
- [Putting It All Together](#)
- [Stored Functions](#)
- [Triggers](#)
- [Calling a Stored Procedure from PHP](#)
- [Conclusion](#)

Language Fundamentals

- [Language Fundamentals](#)
- [Variables, Literals, Parameters, and Comments](#)
- [Operators](#)
- [Expressions](#)
- [Built-in Functions](#)
- [Data Types](#)
- [MySQL 5 Strict Mode](#)
- [Conclusion](#)

Blocks, Conditional Statements, and Iterative Programming

- [Blocks, Conditional Statements, and Iterative Programming](#)
- [Block Structure of Stored Programs](#)
- [Conditional Control](#)
- [Iterative Processing with Loops](#)
- [Conclusion](#)

Using SQL in Stored Programming

- [Using SQL in Stored Programming](#)
- [Using Non-SELECT SQL in Stored Programs](#)
- [Using SELECT Statements with an INTO Clause](#)
- [Creating and Using Cursors](#)
- [Using Unbounded SELECT Statements](#)
- [Performing Dynamic SQL with Prepared Statements](#)
- [Handling SQL Errors: A Preview](#)
- [Conclusion](#)

Error Handling

- [Error Handling](#)
- [Introduction to Error Handling](#)
- [Condition Handlers](#)
- [Named Conditions](#)
- [Missing SQL-2003 Features](#)
- [Putting It All Together](#)
- [Handling Stored Program Errors in the Calling Application](#)
- [Conclusion](#)

Part II: Stored Program Construction

Creating and Maintaining Stored Programs

- [Creating and Maintaining Stored Programs](#)
- [Creating Stored Programs](#)
- [Editing an Existing Stored Program](#)
- [SQL Statements for Managing Stored Programs](#)
- [Gathering Information About Stored Programs](#)
- [Conclusion](#)

Transaction Management

- [Transaction Management](#)
- [Transactional Support in MySQL](#)
- [Defining a Transaction](#)
- [Working with Savepoints](#)
- [Transactions and Locks](#)
- [Transaction Design Guidelines](#)
- [Conclusion](#)

MySQL Built-in Functions

- [MySQL Built-in Functions](#)
- [Stored Functions](#)
- [Numeric Functions](#)
- [Date and Time Functions](#)
- [Other Functions](#)
- [Conclusion](#)

Stored Functions

- [Stored Functions](#)
- [Creating Stored Functions](#)
- [SQL Statements in Stored Functions](#)
- [Calling Stored Functions](#)
- [Using Stored Functions in SQL](#)
- [Conclusion](#)

Triggers

- [Triggers](#)
- [Creating Triggers](#)
- [Using Triggers](#)
- [Trigger Overhead](#)
- [Conclusion](#)

Part III: Using MySQL Stored Programs in Applications

Using MySQL Stored Programs in Applications

- [Using MySQL Stored Programs in Applications](#)
- [The Pros and Cons of Stored Programs in Modern Applications](#)
 - [Advantages of Stored Programs](#)
 - [Disadvantages of Stored Programs](#)
- [Calling Stored Programs from Application Code](#)
- [Conclusion](#)

Using MySQL Stored Programs with PHP

- [Using MySQL Stored Programs with PHP](#)
- [Options for Using MySQL with PHP](#)
- [Using PHP with the mysql Extension](#)
- [Using MySQL with PHP Data Objects](#)
- [Conclusion](#)

Using MySQL Stored Programs with Java

- [Using MySQL Stored Programs with Java](#)
- [Review of JDBC Basics](#)
- [Using Stored Programs in JDBC](#)
- [Stored Programs and JDBC Applications](#)
- [Using Stored Procedures with Hibernate](#)
- [Using Stored Procedures with Spring](#)
- [Conclusion](#)

Using MySQL Stored Programs with Perl

- [Using MySQL Stored Programs with Perl](#)
- [Review of Perl DBD/mysql Basics](#)
- [Executing Stored Programs with DBD::mysql](#)
- [Conclusion](#)

Using MySQL Stored Programs with Python

When MySQL/InnoDB detects a deadlock situation, it will force one of the transactions to roll back and issue an error message, as shown in Example 8-6. In the case of InnoDB, the transaction thus selected will be the transaction that has done the least work (in terms of rows modified).

Example 8-6. Example of a deadlock error

```
mysql> CALL tfer_funds(1,2,300);
ERROR 1213 (40001): Deadlock found when trying to get lock; try restarting transaction
```

Deadlocks can occur in any database system, but in row-level locking databases like MySQL/InnoDB, the possibility of a deadlock is usually low. You can further reduce the frequency of deadlocks by locking rows or tables in a consistent order, and by keeping your transactions as short as possible.

If you are building (or debugging) an application in which deadlocks seem likely to occur, and you cannot reorganize your transactions to avoid them, you can add logic to your programs to handle deadlocks and retry the transaction.

Example 8-7 shows a modified version of the stored procedure in Example 8-2 that will retry its transaction up to three times in the event of a deadlock.

Example 8-7. Stored procedure with deadlock-handling logic

```
1 CREATE PROCEDURE tfer_funds2
2 (from_account INT, to_account INT,
3 tfer_amount numeric(10,2), OUT out_status INT,
4 OUT out_message VARCHAR(30))
5 BEGIN
6
7 DECLARE deadlock INT DEFAULT 0;
8 DECLARE attempts INT DEFAULT 0;
9
10 tfer_loop:WHILE (attempts<3) DO
11 BEGIN
12 DECLARE deadlock_detected CONDITION FOR 1213;
13 DECLARE EXIT HANDLER FOR deadlock_detected
14 BEGIN
15 ROLLBACK;
16 SET deadlock=1;
17 END;
18 SET deadlock=0;
19
20 START TRANSACTION;
21
22 UPDATE account_balance
23 SET balance=balance-tfer_amount
24 WHERE account_id=from_account;
25
26 UPDATE account_balance
27 SET balance=balance+tfer_amount
28 WHERE account_id=to_account;
29
30 COMMIT;
31
32 END;
33 IF deadlock=0 THEN
34 LEAVE tfer_loop;
35 ELSE
36 SET attempts=attempts+1;
37 END IF;
38 END WHILE tfer_loop;
39
40 IF deadlock=1 THEN
41 SET out_status=-1;
42 SET out_message="Failed with deadlock for 3 attempts";
43
44 ELSE
45 SET out_status=0;
46 SET out_message=CONCAT("OK (",attempts," deadlocks)");
47 END IF;
48
49 END;
```

The error-handling techniques in Example 8-7 rely on statements introduced in Chapter 6. Here is a line-by-line explanation of the code:

Line(s)	Explanation
10	Commence a <code>WHILE</code> loop that will control attempts to execute (and possibly re-execute) the transaction. The <code>WHILE</code> loop condition of <code>(attempts<3)</code> ensures that we will try no more than three times to complete this task.
11	Define an anonymous <code>BEGIN</code> block within the loop to contain the transaction. The <code>END</code> statement for this block appears on line 32. The block allows us to trap an error within the body of the loop, but not exit the loop itself.
12-18	Prepare the block for the execution of the transaction. Define an <code>EXIT</code> handler and associate it with the deadlock error. When a deadlock occurs, the handler will set a variable indicating failure, issue a <code>ROLLBACK</code> , and then terminate the block, while remaining within the loop.
20-30	The SQL statements that make up the transaction for this program.
33-37	Determine if it is time to leave the loop or increment the counter. If a deadlock did not occur, the value of the <code>deadlock</code> variable is 0, so we use the <code>LEAVE</code> statement to terminate the <code>WHILE</code> loop. If <code>deadlock</code> equals 1, then the <code>BEGIN-END</code> block has terminated because of a deadlock, so we increment the <code>attempts</code> variable and (provided that <code>attempts</code> has not yet reached 3) allow the loop to re-execute the SQL statements and thereby retry the transaction.
40-47	On these lines we examine the <code>deadlock</code> and <code>attempts</code> variables to determine the final state of the transaction. If <code>deadlock=1</code> , then our most recent attempt to execute the transaction failed with a deadlock, andsince we have tried three timeswe terminate with an error. Otherwise, we signal a successful end to the transaction, although we note how many times we encountered a deadlock in the process.

Going to this much effort to handle deadlocks will be overkill for most applications. Unless your application design is particularly vulnerable to deadlocks, you will encounter deadlocks so infrequently that you actually weaken your application by including so much hard-to-maintain deadlock-handling code.

As noted above, there are usually other ways to avoid deadlock scenarios. For instance, in Example 8-8 we lock the rows to be updated in numerical order before issuing any `UPDATES`. Because the rows are always locked in the same order, one instance of this transaction should not cause a deadlock if another session runs the same program.

Example 8-8. Locking rows in order to avoid deadlock conditions

```
CREATE PROCEDURE tfer_funds3
(from_account INT, to_account INT,tfer_amount NUMERIC(10,2))
BEGIN
DECLARE local_account_id INT;
DECLARE lock_cursor CURSOR FOR
SELECT account_id
FROM account_balance
WHERE account_id IN (from_account,to_account)
ORDER BY account_id
FOR UPDATE;

START TRANSACTION;

OPEN lock_cursor;
FETCH lock_cursor INTO local_account_id;

UPDATE account_balance
SET balance=balance-tfer_amount
WHERE account_id=from_account;

UPDATE account_balance
SET balance=balance+tfer_amount
WHERE account_id=to_account;
```

- [Using MySQL Stored Programs with Python](#)
- [Installing the MySQL db Extension](#)
- [MySQL db Basics](#)
- [Using Stored Programs with MySQLdb](#)
- [A Complete Example](#)
- [Conclusion](#)

Using MySQL Stored Programs with .NET

- [Using MySQL Stored Programs with .NET](#)
- [Review of ADO.NET Basics](#)
- [Using Stored Programs in ADO.NET](#)
- [Using Stored Programs in ASP.NET](#)
- [Conclusion](#)

Part IV: Optimizing Stored Programs

Stored Program Security

- [Stored Program Security](#)
- [Permissions Required for Stored Programs](#)
- [Execution Mode Options for Stored Programs](#)
- [Stored Programs and Code Injection](#)
- [Conclusion](#)

Tuning Stored Programs and Their SQL

- [Tuning Stored Programs and Their SQL](#)
- [Why SQL Tuning Is So Important](#)
- [How MySQL Processes SQL](#)
- [SQL Tuning Statements and Practices](#)
- [About the Upcoming Examples](#)
- [Conclusion](#)

Basic SQL Tuning

- [Basic SQL Tuning](#)
- [Tuning Table Access](#)
- [Tuning Joins](#)
- [Conclusion](#)

Advanced SQL Tuning

- [Advanced SQL Tuning](#)
- [Tuning Subqueries](#)
- [Tuning Anti-Joins Using Subqueries](#)
- [Tuning Subqueries in the FROM Clause](#)
- [Tuning ORDER and GROUP BY](#)
- [Tuning DML \(INSERT, UPDATE, DELETE\)](#)
- [Conclusion](#)

Optimizing Stored Program Code

- [Optimizing Stored Program Code](#)
- [Performance Characteristics of Stored Programs](#)
- [How Fast Is the Stored Program Language?](#)
- [Reducing Network Traffic with Stored Programs](#)
- [Stored Programs as an Alternative to Expensive SQL](#)
- [Optimizing Loops](#)
- [IF and CASE Statements](#)
- [Recursion](#)
- [Cursors](#)
- [Trapper Overhead](#)
- [Conclusion](#)

Best Practices in MySQL Stored Program Development

- [Best Practices in MySQL Stored Program Development](#)
- [The Development Process](#)
- [Coding Style and Conventions](#)
- [Variables](#)
- [Conditional Logic](#)
- [Loop Processing](#)
- [Exception Handling](#)
- [SQL in Stored Programs](#)
- [Dynamic SQL](#)
- [Program Construction](#)
- [Performance](#)

```
    CLOSE lock_cursor;

    COMMIT;

END;
```

8.4.3. Lock Timeouts

A deadlock is the most severe result of locking. Yet, in many other situations, a program in one session may be unable to read or write a particular row, because it is locked by another session. In this case, the program can and by default will wait for a certain period of time for the lock to be released. It will then either acquire the lock or time out. You can set the length of time a session will wait for an InnoDB lock to be released by setting the value of the `innodb_lock_wait_timeout` configuration value, which has a default of 50 seconds.

When a timeout occurs, MySQL/InnoDB will roll back the transaction and issue an error code 1205, as shown in Example 8-9.

Example 8-9. Lock timeout error

```
mysql> SELECT * FROM account_balance FOR UPDATE;
ERROR 1205 (HY000): Lock wait timeout exceeded; try restarting transaction
```

So if you have very long-running transactions, you may want to increase the value of `innodb_lock_wait_timeout` or introduce error-handling code to cope with the occasional 1205 error.

In some circumstances, particularly when you mix MySQL/InnoDB and non-InnoDB tables in the same transaction (a practice we do not normally recommend), MySQL/InnoDB may be unable to detect a deadlock. In such cases, the "lock wait timeout" error will eventually occur. If you are mixing MySQL/InnoDB and non-InnoDB tables, and you are particularly concerned about deadlocks, you may want to implement error-handling logic for lock timeouts similar to that implemented for deadlocks in Example 8-7.

8.4.4. Optimistic and Pessimistic Locking Strategies

If your transaction reads data that subsequently participates in an `UPDATE`, `INSERT`, or `DELETE`, you need to take steps to ensure that the integrity of your transaction is not jeopardized by the possibility of another transaction changing the relevant data between the time you read it and the time you update it.

For instance, consider the transaction in Example 8-10. This variation on our funds transfer transaction makes sure that there are sufficient funds in the "from" account before executing the transaction. It first queries the account balance, and then takes an action depending on that value (the balance must be greater than the transfer amount).

Example 8-10. Funds transfer program without locking strategy

```
CREATE PROCEDURE tfer_funds4
  (from_account INT, to_account INT, tfer_amount NUMERIC(10,2),
  OUT status INT, OUT message VARCHAR(30))
BEGIN
  DECLARE from_account_balance NUMERIC(10,2);

  SELECT balance
  INTO from_account_balance
  FROM account_balance
  WHERE account_id=from_account;

  IF from_account_balance >= tfer_amount THEN

    START TRANSACTION;

    UPDATE account_balance
    SET balance=balance-tfer_amount
    WHERE account_id=from_account;

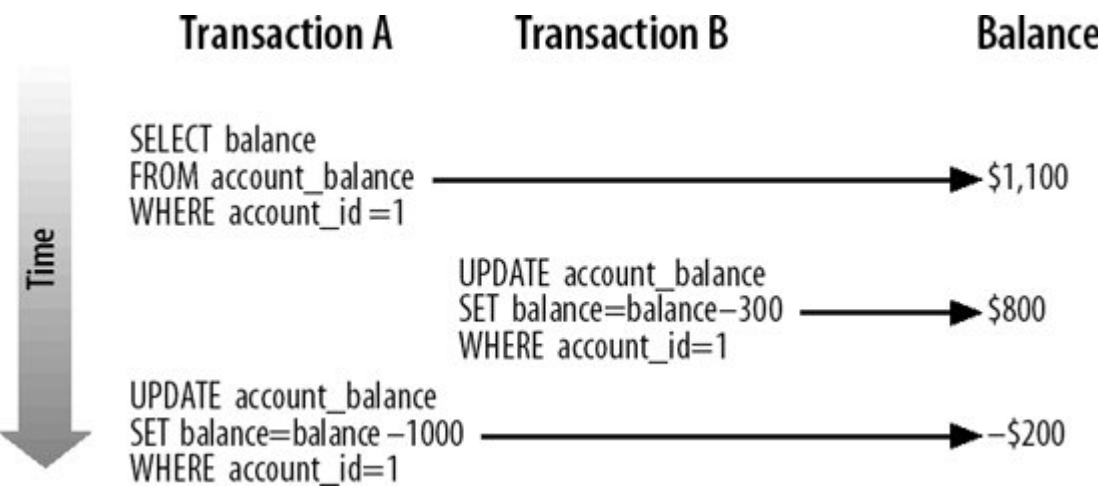
    UPDATE account_balance
    SET balance=balance+tfer_amount
    WHERE account_id=to_account;
    COMMIT;

    SET status=0;
    SET message='OK';
  ELSE
    SET status=-1;
    SET message='Insufficient funds';
  END IF;
END;
```

Unfortunately, as currently written, this program might under the right circumstances allow the "from" account to become overdrawn. Since some amount of time elapses between the query that establishes the current balance and the update transaction that reduces that balance, it is possible that another transaction could reduce the balance of the account within that period of time with its own `UPDATE` statement. This program's `UPDATE` would, then, cause a negative balance in the account.

Figure 8-4 shows the business policy violation that can result from a poor locking strategy. Transaction A determines that account 1 has sufficient funds before executing the transfer, but in the meantime transaction B has reduced the available funds by \$300. When transaction A finally executes its update, the result is a negative balance.

Figure 8-4. Error resulting from a poor locking strategy



There are two typical solutions to this kind of scenario:

The pessimistic locking strategy

Assume that concurrent updates are quite likely to occur, and write programs to prevent them from happening. Generally, this means you will need to lock rows as they are read. Other transactions that want to update the row must wait until the "pessimistic transaction" ends.

The optimistic locking strategy

Assume that it is unlikely that anyone will update a row between the time we view it and the time we update it. Since we cannot be sure that this assumption is true, we must then, at the last possible moment, make sure that the row has not been updated. If the row has been updated, the transaction cannot be trusted and will have to be aborted.

8.4.4.1. Pessimistic locking strategy

Let's explore the pessimistic strategy first, with a simple example. We ensure that nobody modifies the balance of the "from" account by locking it with the `FOR UPDATE` clause as we retrieve the balance. We can now rest assured that when we issue our `UPDATE` statement, the balance of the account cannot have been altered. Example 8-11 shows how easy this is; all we needed to do was move the `SELECT` statement inside of the transaction and cause it to lock the rows selected with the `FOR UPDATE` clause.

Example 8-11. Pessimistic locking strategy

```
CREATE PROCEDURE tfer_funds5
  (from_account INT, to_account INT, tfer_amount NUMERIC(10,2),
  OUT status INT, OUT message VARCHAR(30))
BEGIN
  DECLARE from_account_balance NUMERIC(10,2);

  START TRANSACTION;

  SELECT balance
  INTO from_account_balance
  FROM account_balance
  WHERE account_id=from_account
  FOR UPDATE;

  IF from_account_balance >= tfer_amount THEN
```

```
UPDATE account_balance
SET balance=balance-tfer_amount
WHERE account_id=from_account;

UPDATE account_balance
SET balance=balance+tfer_amount
WHERE account_id=to_account;
COMMIT;

SET status=0;
SET message='OK';
ELSE
ROLLBACK;
SET status=-1;
SET message='Insufficient funds';
END IF;
END;
```

The pessimistic locking strategy usually results in the simplest and most robust code code that ensures consistency between `SELECT` and DML statements within your transaction. The pessimistic strategy can, however, lead to long-held locks that degrade performance (forcing a large number of sessions to wait for the locks to be released). For instance, suppose that after you validate the balance of the transaction, you are required to perform some long-running validationperhaps you need to check various other databases (credit checking, blocked accounts, online fraud, etc.) before finalizing the transaction. In this case, you may end up locking the account for several minutesleading to disgruntlement if the customer happens to be trying to withdraw funds at the same time.

8.4.4.2. Optimistic locking strategy

The optimistic locking strategy assumes that it is unlikely that the row will be updated between the initial `SELECT` and the end of the transaction, and therefore does not attempt to lock that row. Instead, the optimistic strategy requires that we perform a check just before the update to ensure that the row has not been altered.

To detect if a row has been changed, we simply refetch the rowlocking the row as we do soand compare the current values with the previous values.

Example 8-12 demonstrates the optimistic locking strategy. If the account row has changed since the time of the initial balance check, the transaction will be aborted (line 33), although alternatively you could retry the transaction.

Example 8-12. Optimistic locking strategy

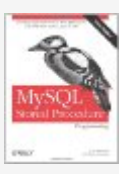
```
1 CREATE PROCEDURE tfer_funds6
2 (from_account INT, to_account INT, tfer_amount NUMERIC(10,2),
3 OUT status INT, OUT message VARCHAR(30) )
4
5 BEGIN
6
7 DECLARE from_account_balance NUMERIC(8,2);
8 DECLARE from_account_balance2 NUMERIC(8,2);
9 DECLARE from_account_timestamp1 TIMESTAMP;
10 DECLARE from_account_timestamp2 TIMESTAMP;
11
12 SELECT account_timestamp,balance
13 INTO from_account_timestamp1,from_account_balance
14 FROM account_balance
15 WHERE account_id=from_account;
16
17 IF (from_account_balance)=tfer_amount) THEN
18
19 -- Here we perform some long running validation that
20 -- might take a few minutes */
21 CALL long_running_validation(from_account);
22
23 START TRANSACTION;
24
25 -- Make sure the account row has not been updated since
26 -- our initial check
27 SELECT account_timestamp, balance
28 INTO from_account_timestamp2,from_account_balance2
29 FROM account_balance
30 WHERE account_id=from_account
31 FOR UPDATE;
32
33 IF (from_account_timestamp1 <> from_account_timestamp2 OR
34 from_account_balance <> from_account_balance2) THEN
35 ROLLBACK;
36 SET status=-1;
37 SET message=CONCAT("Transaction cancelled due to concurrent update",
38 " of account",from_account);
39 ELSE
40 UPDATE account_balance
41 SET balance=balance-tfer_amount
42 WHERE account_id=from_account;
43
44 UPDATE account_balance
45 SET balance=balance+tfer_amount
46 WHERE account_id=to_account;
47
48 COMMIT;
49
50 SET status=0;
51 SET message="OK";
52 END IF;
53
54 ELSE
55 ROLLBACK;
56 SET status=-1;
57 SET message="Insufficient funds";
58 END IF;
59 END$$
```

Optimistic locking strategies are often employed by transactions that involve user interaction, since there is sometimes the chance that a user will "go to lunch," leaving a pessimistic lock in place for an extended period. Since stored programs do not involve direct user interaction, optimistic strategies in stored programs are not required for this reason. However, an optimistic strategy might still be selected as a means of reducing overall lock duration and improving application throughputat the cost of occasionally having to retry the transaction when the optimism is misplaced.

8.4.4.3. Choosing between strategies

Don't choose between optimistic and pessimistic strategies based on your personality or disposition. Just because your analyst assures you that you are a fairly fun-loving, optimistic guy or gal, that does not mean you should affirm this by always choosing the optimistic locking strategy!

The choice between the two strategies is based on a trade-off between concurrency and robustness: pessimistic locking is less likely to require transaction retries or failures, while optimistic locking minimizes the duration of locks, thus improving concurrency and transaction throughput. Usually, we choose optimistic locking only if the duration of the locks or the number of rows locked by the pessimistic solution would be unacceptable.



MySQL Stored Procedure Programming

ISBN: 0596100892
EAN: 2147483647

Year: 2004
Pages: 208

Authors: [Guy Harrison](#), [Steven Feuerstein](#)

BUY ON AMAZON

Variance Responses
What Exactly Is a Project Change and What's the Big Deal Anyway?
Principles of Managing Project Quality

[INDESIGN TYPE: PROFESSIONAL TYPOGRAPHY WITH ADOBE INDESIGN CS2](#)

Keep It Consistent: Except....
Balancing Ragged Lines
Indent to Here
Up Next
Right Indent Tab

Understanding the Domino Web Application Server
Implementing View-Level Security
Introducing Workflow

[JUNOS COOKBOOK / COOKBOOKS \(O'REILLY\)](#)

Gathering Information Before Contacting Support
Using the Second Routing Engine to Upgrade to a New Software Version
Dealing with Nonconfigurable Interfaces
Viewing Routes Learned by IS-IS
Using Fast Reroute to Reduce Packet Loss Following a Link Failure

Writing External Applications to Query and Manipulate Database Data
Monitoring and Enhancing MS-SQL Server Performance
Repairing and Maintaining MS-SQL Server Database Files

[TELECOMMUNICATIONS ESSENTIALS, SECOND EDITION: THE COMPLETE GLOBAL SOURCE \(2ND EDITION\)](#)

The PSTN Infrastructure
LAN Basics
IPTV
VPNs
3G: Moving Toward Broadband Wireless