Holdable result sets and autocommit

When autocommit is on, a positioned update or delete statement will automatically cause the transaction to commit.

If the result set has holdability `ResultSet.CLOSE_CURSORS_AT_COMMIT`, combined with autocommit on, Derby gives an exception on positioned updates and deletes because the cursor is closed immediately before the positioned statement is commenced, as mandated by JDBC. In contrast, no such implicit commit is done when using result set updates methods.

Non-holdable result set example

The following example uses *Connection.createStatement* to return a `ResultSet` that will close after a commit is performed.

```
Connection conn = ds.getConnection(user, passwd);
Statement stmt =
conn.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
                  ResultSet.CONCUR_READ_ONLY,
                  ResultSet.CLOSE_CURSORS_AT_COMMIT);
```

Locking, concurrency, and isolation

This section discusses topics pertinent to multi-user systems, in which concurrency is important.

Derby is configured by default to work well for multi-user systems. For single-user systems, you might want to tune your system so that it uses fewer resources; see Lock granularity.

Isolation levels and concurrency

Derby provides four transaction isolation levels. Setting the transaction isolation level for a connection allows a user to specify how severely the user's transaction should be isolated from other transactions.

For example, it allows you to specify whether transaction A is allowed to make changes to data that have been viewed by transaction B before transaction B has committed.

A connection determines its own isolation level, so JDBC provides an application with a way to specify a level of transaction isolation. It specifies four levels of transaction isolation. The higher the transaction isolation, the more care is taken to avoid conflicts; avoiding conflicts sometimes means locking out transactions. Lower isolation levels thus allow greater concurrency.

Inserts, updates, and deletes always behave the same no matter what the isolation level is. Only the behavior of select statements varies.

To set isolation levels you can use the JDBC *Connection.setTransactionIsolation* method or the SQL SET ISOLATION statement.

If there is an active transaction, the network client driver always commits the active transaction, whether you use the JDBC *Connection.setTransactionIsolation* method or the SQL SET ISOLATION statement. It does this even if the method call or statement does not actually change the isolation level (that is, if it sets the isolation level to its current value). The embedded driver also always commits the active transaction if you use the SET ISOLATION statement. However, if you use the *Connection.setTransactionIsolation* method, the embedded driver commits the active transaction only if the call to *Connection.setTransactionIsolation* actually changes the isolation level.

The names of the isolation levels are different, depending on whether you use a JDBC method or SQL statement. The following table shows the equivalent names for isolation levels whether they are set through the JDBC method or an SQL statement.

Table 6. Mapping of JDBC transaction isolation levels to Derby isolation levels

Isolation Levels for JDBC	Isolation Levels for SQL
Connection.TRANSACTION_READ_UNCOMMITTED (ANSI level 0)	UR, DIRTY READ, READ UNCOMMITTED
Connection.TRANSACTION_READ_COMMITTED (ANSI level 1)	CS, CURSOR STABILITY, READ COMMITTED
Connection.TRANSACTION_REPEATABLE_READ (ANSI level 2)	RS
Connection.TRANSACTION_SERIALIZABLE (ANSI level 3)	RR, REPEATABLE READ, SERIALIZABLE

These levels allow you to avoid particular kinds of transaction anomalies, which are described in the following table.

Table 7. Transaction anomalies

Anomaly	Example

Dirty Reads
 	
A dirty read happens when a transaction reads data that is being modified by another transaction that has not yet committed.
	Transaction A begins.
	
	`UPDATE employee SET salary = 31650 WHERE empno = '000090'`
	
	Transaction B begins.
	
	`SELECT * FROM employee`
	
	(Transaction B sees data updated by transaction A. Those updates have not yet been committed.)

Nonrepeatable Reads
 	
Nonrepeatable reads happen when a query returns data that would be different if the query were repeated within the same transaction. Nonrepeatable reads can occur when other transactions are modifying data that a transaction is reading.
	Transaction A begins.
	
	`SELECT * FROM employee WHERE empno = '000090'`
	
	Transaction B begins.
	
	`UPDATE employee SET salary = 30100 WHERE empno = '000090'`
	
	(Transaction B updates rows viewed by transaction A before transaction A commits.) If Transaction A issues the same SELECT statement, the results will be different.

Phantom Reads
 	
Records that appear in a set being read by another transaction. Phantom reads can occur when other transactions insert rows that would satisfy the WHERE clause of another transaction's statement.
	Transaction A begins.
	
	`SELECT * FROM employee WHERE salary > 30000`
	
	Transaction B begins.
	
	`INSERT INTO employee`

Anomaly	Example
	`(empno, firstnme, midinit, lastname, job, salary) VALUES ('000350', 'NICK', 'A','GREEN','LEGAL COUNSEL',35000)` Transaction B inserts a row that would satisfy the query in Transaction A if it were issued again.

The transaction isolation level is a way of specifying whether these transaction anomalies are allowed. The transaction isolation level thus affects the quantity of data locked by a particular transaction. In addition, a DBMS's locking schema might also affect whether these anomalies are allowed. A DBMS can lock either the entire table or only specific rows in order to prevent transaction anomalies.

The following table shows which anomalies are possible under the various locking schemas and isolation levels.

Table 8. When transaction anomalies are possible

Isolation Level	Table-Level Locking	Row-Level Locking
TRANSACTION_READ_UNCOMMITTED	Dirty reads, nonrepeatable reads, and phantom reads possible	Dirty reads, nonrepeatable reads, and phantom reads possible
TRANSACTION_READ_COMMITTED	Nonrepeatable reads and phantom reads possible	Nonrepeatable reads and phantom reads possible
TRANSACTION_REPEATABLE_READ	Phantom reads not possible because entire table is locked	Phantom reads possible
TRANSACTION_SERIALIZABLE	None	None

The following *java.sql.Connection* isolation levels are supported:

- `TRANSACTION_SERIALIZABLE`

  `RR`, `SERIALIZABLE`, or `REPEATABLE READ` from SQL.

  TRANSACTION_SERIALIZABLE means that Derby treats the transactions as if they occurred serially (one after the other) instead of concurrently. Derby issues locks to prevent all the transaction anomalies listed in Transaction anomalies from occurring. The type of lock it issues is sometimes called a *range lock*.
- `TRANSACTION_REPEATABLE_READ`

  `RS` from SQL.

  TRANSACTION_REPEATABLE_READ means that Derby issues locks to prevent only dirty reads and nonrepeatable reads, but not phantoms. It does not issue range locks for selects.
- `TRANSACTION_READ_COMMITTED`

  `CS` or `CURSOR STABILITY` from SQL.

TRANSACTION_READ_COMMITTED means that Derby issues locks to prevent only dirty reads, not all the transaction anomalies listed in Transaction anomalies.

TRANSACTION_READ_COMMITTED is the default isolation level for transactions.

- `TRANSACTION_READ_UNCOMMITTED`

  `UR`, `DIRTY READ`, or `READ UNCOMMITTED` from SQL.

  For a SELECT INTO, FETCH with a read-only cursor, full select used in an INSERT, full select/subquery in an UPDATE/DELETE, or scalar full select (wherever used), READ UNCOMMITTED allows:

  - Any row that is read during the unit of work to be changed by other application processes.
  - Any row that was changed by another application process to be read even if the change has not been committed by the application process.

  For other operations, the rules that apply to READ COMMITTED also apply to READ UNCOMMITTED.

Configuring isolation levels

If a connection does not specify its isolation level, it inherits the default isolation level for the Derby system. The default value is `CS`.

When set to `CS`, the connection inherits the TRANSACTION_READ_COMMITTED isolation level. When set to `RR`, the connection inherits the TRANSACTION_SERIALIZABLE isolation level, when set to `RS`, the connection inherits the TRANSACTION_REPEATABLE_READ isolation level, and when set to `UR`, the connection inherits the TRANSACTION_READ_UNCOMMITTED isolation level.

To override the inherited default, use the methods of *java.sql.Connection*.

In addition, a connection can change the isolation level of the transaction within an SQL statement. For more information, see "SET ISOLATION statement" in the *Derby Reference Manual*. You can use the WITH clause to change the isolation level for the current statement only, not the transaction. For information about the WITH clause, see "SELECT statement" in the *Derby Reference Manual*.

In all cases except when you change the isolation level using the WITH clause, changing the isolation level commits the current transaction. In most cases, the current transaction is committed even if you set the isolation level in a way that does not change it (that is, if you set it to its current value). See Isolation levels and concurrency for details.

Note: For information about how to choose a particular isolation level, see "Shielding users from Derby class-loading events" in *Tuning Derby* and Multi-thread programming tips.

Lock granularity

Derby can be configured for *table-level* locking. With table-level locking, when a transaction locks data in order to prevent any transaction anomalies, it always locks the entire table, not just those rows being accessed.

By default, Derby is configured for row-level locking. Row-level locking uses more memory but allows greater concurrency, which works better in multi-user systems. Table-level locking works best with single-user applications or read-only applications.

You typically set lock granularity for the entire Derby system, not for a particular application. However, at runtime, Derby may escalate the lock granularity for a particular transaction from row-level locking to table-level locking for performance reasons. You have some control over the threshold at which this occurs. For information on turning

off row-level locking, see "*derby.storage.rowLocking*" in the *Derby Reference Manual*. For more information about automatic lock escalation, see "About the system's selection of lock granularity" and "Transaction-based lock escalation" in *Tuning Derby*. For more information on tuning your Derby system, see "Tuning databases and applications," also in *Tuning Derby*.

Types and scope of locks in Derby systems

There are several types of locks available in Derby systems, including exclusive, shared, and update locks.

Exclusive locks

When a statement modifies data, its transaction holds an *exclusive* lock on data that prevents other transactions from accessing the data.

This lock remains in place until the transaction holding the lock issues a commit or rollback. Table-level locking lowers concurrency in a multi-user system.

Shared locks

When a statement reads data without making any modifications, its transaction obtains a *shared lock* on the data.

Another transaction that tries to read the same data is permitted to read, but a transaction that tries to update the data will be prevented from doing so until the shared lock is released. How long this shared lock is held depends on the isolation level of the transaction holding the lock. Transactions using the TRANSACTION_READ_COMMITTED isolation level release the lock when the transaction steps through to the next row. Transactions using the TRANSACTION_SERIALIZABLE or TRANSACTION_REPEATABLE_READ isolation level hold the lock until the transaction is committed, so even a SELECT can prevent updates if a commit is never issued. Transactions using the TRANSACTION_READ_UNCOMMITTED isolation level do not request any locks.

Update locks

When a user-defined update cursor (created with the FOR UPDATE clause or by using concurrency mode `ResultSet.CONCUR_UPDATABLE`) reads data, its transaction obtains an *update* lock on the data.

If the user-defined update cursor updates the data, the update lock is converted to an exclusive lock. If the cursor does not update the row, when the transaction steps through to the next row, transactions using the TRANSACTION_READ_COMMITTED isolation level release the lock. (For update locks, the TRANSACTION_READ_UNCOMMITTED isolation level acts the same way as TRANSACTION_READ_COMMITTED.)

Update locks help minimize deadlocks.

Lock compatibility

The following table shows the compatibility between lock types. "Yes" means that the lock types are compatible, while "No" means that they are incompatible.

Table 9. Lock Compatibility Matrix

Lock Type	Shared	Update	Exclusive
Shared	Yes	Yes	No
Update	Yes	No	No
Exclusive	No	No	No

Scope of locks

The amount of data locked by a statement can vary.

Table locks

A statement can lock the *entire table*.

Table-level locking systems always lock entire tables.

Row-level locking systems can lock entire tables if the WHERE clause of a statement cannot use an index. For example, UPDATES that cannot use an index lock the entire table.

Row-level locking systems can lock entire tables if a high number of single-row locks would be less efficient than a single table-level lock. Choosing table-level locking instead of row-level locking for performance reasons is called *lock escalation*. For more information about this topic, see "About the system's selection of lock granularity" and "Transaction-based lock escalation" in *Tuning Derby*.

Single-row locks

A statement can lock only *a single row* at a time.

For row-level locking systems:

- For TRANSACTION_REPEATABLE_READ isolation, the locks are released at the end of the transaction.
- For TRANSACTION_READ_COMMITTED isolation, Derby locks rows only as the application steps through the rows in the result. The current row is locked. The row lock is released when the application goes to the next row.
- For TRANSACTION_SERIALIZABLE isolation, however, Derby locks the whole set before the application begins stepping through.
- For TRANSACTION_READ_UNCOMMITTED, no row locks are requested.

Derby locks single rows for INSERT statements, holding each row until the transaction is committed. If there is an index associated with the table, the previous key is also locked.

Range locks

A statement can lock *a range of rows* (range lock).

For row-level locking systems:

- For *any* isolation level, Derby locks *all the rows in the result* plus an entire range of rows for updates or deletes.
- For the TRANSACTION_SERIALIZABLE isolation level, Derby locks all the rows in the result plus an entire range of rows in the table for SELECTs to prevent nonrepeatable reads and phantoms.

For example, if a SELECT statement specifies rows in the *Employee* table where the *salary* is BETWEEN two values, the system can lock more than just the actual rows it returns in the result. It also must lock the entire *range* of rows between those two values to prevent another transaction from inserting, deleting, or updating a row within that range.

An index must be available for a range lock. If one is not available, Derby locks the entire table.

The following table summarizes the types and scopes of locking.

Table 10. Types and scopes of locking

Transaction Isolation Level	Table-Level Locking	Row-Level Locking
Connection.TRANSACTION_READ_UNCOMMITTE (SQL: UR)	For SELECT statements, table-level locking is never requested using this isolation level. For other statements, same as for TRA	SELECT statements get no locks. For other statements, same as for TRANSACTION_RE D_COMMITTED
Connection.TRANSACTION_READ_COMMITTED (SQL: CS)	SELECT statements get a shared lock on the entire table. The locks are released when the user closes the *ResultSet*. Other statements get exclusive locks on the entire table, which are released when the transaction commits.	SELECTs lock and release single rows as the user steps through the *ResultSet*. UPDATEs and DELETEs get exclusive locks on a range of rows. INSERT statements get exclusive locks on single rows (and sometimes on the preceding rows).
Connection.TRANSACTION_REPEATABLE_READ (SQL: RS)	Same as for TRA	SELECT statements get shared locks on the rows that satisfy the WHERE clause (but do not prevent inserts into this range). UPDATEs and DELETEs get exclusive locks on a range of rows. INSERT statements get exclusive locks on single rows (and sometimes on the preceding rows). IALIZABLE
Connection.TRANSACTION_SERIALIZABLE (SQL: RR)	SELECT statements get a shared lock	SELECT statements get shared locks

Transaction Isolation Level	Table-Level Locking	Row-Level Locking
	on the entire table. Other statements get exclusive locks on the entire table, which are released when the transaction commits.	on a range of rows. UPDATE and DELETE statements get exclusive locks on a range of rows. INSERT statements get exclusive locks on single rows (and sometimes on the preceding rows).

Notes on locking

In addition to the locks already described, foreign key lookups require briefly held shared locks on the referenced table (row or table, depending on the configuration).

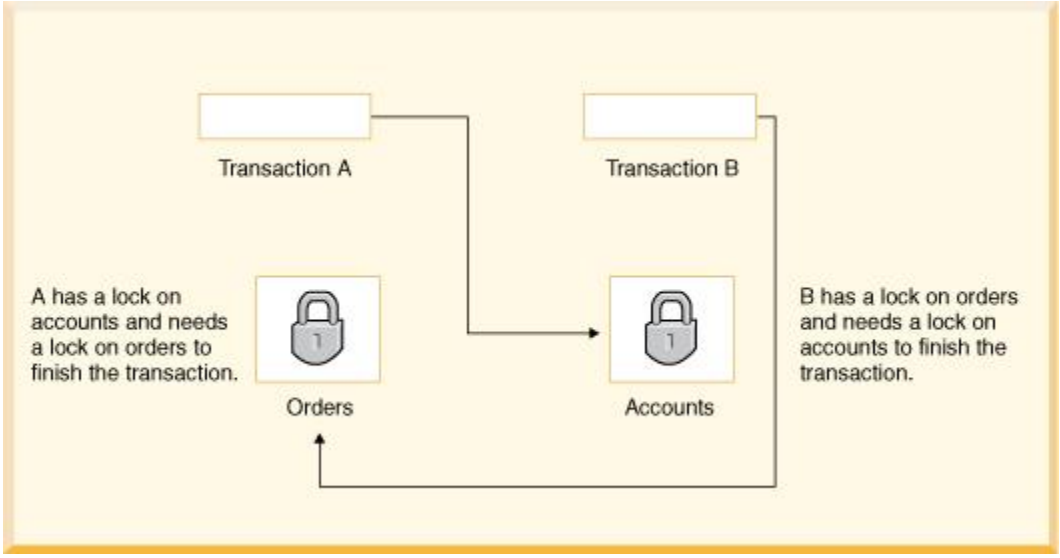
The table and examples in this section do not take performance-based lock escalation into account. Remember that the system can choose table-level locking for performance reasons.

Deadlocks

In a database, a deadlock is a situation in which two or more transactions are waiting for one another to give up locks.

For example, Transaction A might hold a lock on some rows in the *Accounts* table and needs to update some rows in the *Orders* table to finish. Transaction B holds locks on those very rows in the *Orders* table but needs to update the rows in the *Accounts* table held by Transaction A. Transaction A cannot complete its transaction because of the lock on *Orders*. Transaction B cannot complete its transaction because of the lock on *Accounts*. All activity comes to a halt and remains at a standstill forever unless the DBMS detects the deadlock and aborts one of the transactions. The following figure shows this situation.

Figure 6. A deadlock where two transactions are waiting for one another to give up locks



Avoiding deadlocks

Using both row-level locking and the TRANSACTION_READ_COMMITTED isolation level makes it likely that you will avoid deadlocks (both settings are Derby defaults). However, deadlocks are still possible.

Derby application developers can avoid deadlocks by using consistent application logic; for example, transactions that access *Accounts* and *Orders* should always access the tables in the same order. That way, in the scenario described above, Transaction B simply waits for transaction A to release the lock on *Orders* before it begins. When transaction A releases the lock on *Orders*, Transaction B can proceed freely.

The appropriate use of indexes can also help you to avoid deadlocks, since indexes make table scans less likely and reduce the number of locks obtained. For more information, see "CREATE INDEX statement" in the *Derby Reference Manual* and the topics under "Avoiding table scans of large tables" in *Tuning Derby*.

Another tool available to you is the LOCK TABLE statement. A transaction can attempt to lock a table in exclusive mode when it starts to prevent other transactions from getting shared locks on a table. For more information, see "LOCK TABLE statement" in the *Derby Reference Manual*.

Deadlock detection

When a transaction waits more than a specific amount of time to obtain a lock (called the deadlock timeout), Derby can detect whether the transaction is involved in a deadlock.

When Derby analyzes such a situation for deadlocks it tries to determine how many transactions are involved in the deadlock (two or more). Usually aborting one transaction breaks the deadlock. Derby must pick one transaction as the victim and abort that transaction; it picks the transaction that holds the fewest number of locks as the victim, on the assumption that transaction has performed the least amount of work. (This may not be the case, however; the transaction might have recently been escalated from row-level locking to table locking and thus hold a small number of locks even though it has done the most work.)

When Derby aborts the victim transaction, it receives a deadlock error (an *SQLException* with an *SQLState* of 40001). The error message gives you the transaction IDs, the statements, and the status of locks involved in a deadlock situation.

```
ERROR 40001: A lock could not be obtained due to a deadlock,
cycle of locks & waiters is:
```

```
Lock : ROW, DEPARTMENT, (1,14)
Waiting XID : {752, X} , APP, update department set location='Boise'
 where deptno='E21'
Granted XID : {758, X} Lock : ROW, EMPLOYEE, (2,8)
Waiting XID : {758, U} , APP, update employee set bonus=150 where
 salary=23840
Granted XID : {752, X} The selected victim is XID : 752
```

For information on configuring when deadlock checking occurs, see Configuring deadlock detection and lock wait timeouts.

Note: Deadlocks are detected only within a single database. Deadlocks across multiple databases are not detected. Non-database deadlocks caused by Java synchronization primitives are not detected by Derby.

Lock wait timeouts

Even if a transaction is not involved in a deadlock, it might have to wait a considerable amount of time to obtain a lock because of a long-running transaction or transactions holding locks on the tables it needs.

In such a situation, you might not want a transaction to wait indefinitely. Instead, you might want the waiting transaction to abort, or *time out*, after a reasonable amount of time, called a *lock wait timeout*.

Configuring deadlock detection and lock wait timeouts

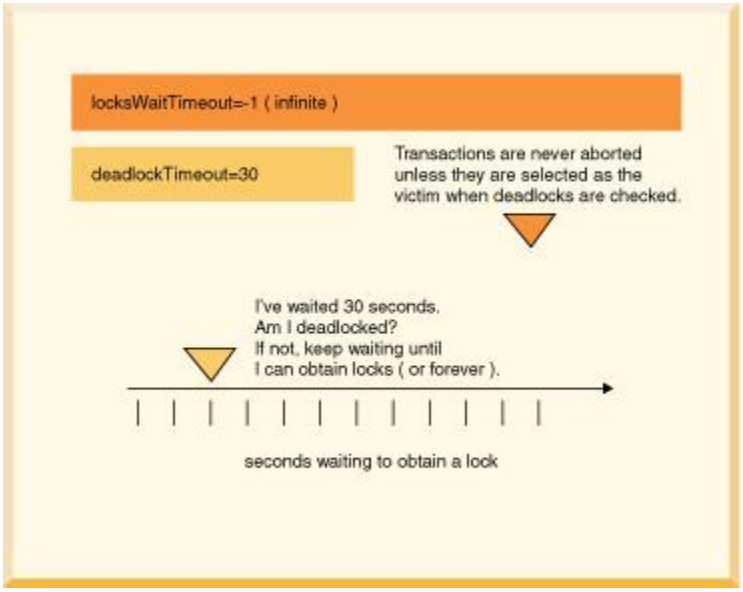
You configure the amount of time a transaction waits before Derby does any deadlock checking with the *derby.locks.deadlockTimeout* property.

You configure the amount of time a transaction waits before timing out with the *derby.locks.waitTimeout* property. When configuring your database or system, you should consider these properties together. For example, in order for any deadlock checking to occur, the *derby.locks.deadlockTimeout* property must be set to a value lower than the *derby.locks.waitTimeout* property. If it is set to a value equal to or higher than the *derby.locks.waitTimeout*, the transaction times out before Derby does any deadlock checking.

By default, *derby.locks.waitTimeout* is set to 60 seconds. -1 is the equivalent of no wait timeout. This means that transactions never time out, although Derby can choose a transaction as a deadlock victim.

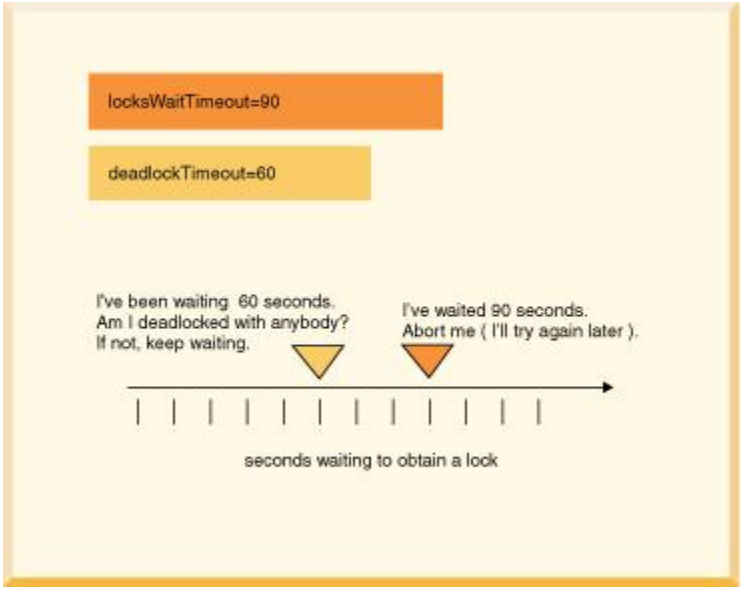
In the following figure, *derby.locks.deadlockTimeout* is set to 30 seconds, while *derby.locks.waitTimeout* has no limit.

Figure 7. Configuration with deadlock checking after 30 seconds and no lock wait timeouts



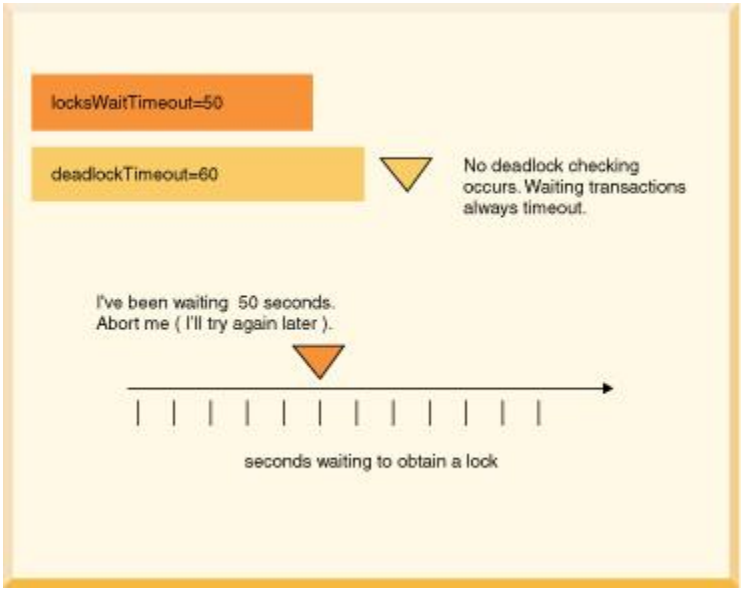
In the following figure, *derby.locks.deadlockTimeout* is set to 60 seconds, while *derby.locks.waitTimeout* is set to 90 seconds.

Figure 8. Configuration with deadlock checking after 60 seconds and lock wait timeout at 90 seconds



In the following figure, *derby.locks.deadlockTimeout* is set to 60 seconds, while *derby.locks.waitTimeout* is set to 50 seconds, lower than the deadlock timeout limit.

Figure 9. Configuration with no deadlock checking and a 50-second lock wait timeout



Debugging deadlocks

If deadlocks occur frequently in your multi-user system with a particular application, you might need to do some debugging.

Derby provides the SYSCS_DIAG.LOCK_TABLE diagnostic table to help you debug deadlocks. This diagnostic table shows all of the locks that are currently held in the Derby database. You can reference the SYSCS_DIAG.LOCK_TABLE diagnostic table directly in a statement.

For example:

```
SELECT * FROM SYSCS_DIAG.LOCK_TABLE
```

When the SYSCS_DIAG.LOCK_TABLE diagnostic table is referenced in a statement, a snapshot of the lock table is taken.

For more information about how to use this table, see "SYSCS_DIAG.LOCK_TABLE diagnostic table" in the *Derby Reference Manual*.

You can also set the property *derby.locks.deadlockTrace* to dump additional information to the *derby.log* file about any deadlocks that occur on your system. See *Derby Reference Manual* for more information on this property. Also see "Monitoring deadlocks" in the *Derby Server and Administration Guide*.

Additional general information about diagnosing locking problems can be found in the Derby Wiki at http://wiki.apache.org/db-derby/LockDebugging.

Programming applications to handle deadlocks

When you configure your system for deadlock and lockwait timeouts and an application could be chosen as a victim when the transaction times out, you should program your application to handle them.

To do this, test for *SQLExceptions* with *SQLStates* of 40001 (deadlock timeout) or 40XL1 (lockwait timeout).

In the case of a deadlock you might want to re-try the transaction that was chosen as a victim. In the case of a lock wait timeout, you probably do not want to do this right away.

The following code is one example of how to handle a deadlock timeout.

```
/// if this code might encounter a deadlock,
```