CHAPTER 15

Using Transactions and Locks

About this chapter

You can group SQL statements into units called *transactions*. Transactions are simply groups of SQL statements with the property that either all or none of the group is executed. You should design each transaction to perform a task that changes your database from one consistent state to another. These units play an important role in protecting your database from media and system failures.

When several people use the same database at the same time, problems can occur. Adaptive Server Anywhere uses transactions in conjunction with *locks* to prevent or eliminate inconsistencies. A lock limits the access of other transactions to a particular row of a table.

This chapter describes transactions and how to use them in applications. It also describes the locking mechanisms at your disposal.

Contents

An overview of transactions

About transaction processing

Adaptive Server Anywhere supports transaction processing, which ensures that logically related commands are executed as a unit. Transactions are fundamental to maintaining the accuracy of your data. Transactions are essential to data recovery in the event of system failure and to the successful interweaving of commands from concurrent users.

Who needs to know about transactions

All developers must be concerned with correctness and concurrency. Even single-user databases must be protected against data loss and may have multiple applications connected to them, or may have multiple connections from a single application. Understanding transactions will allow you to make better use of the facilities provided by Sybase Adaptive Server Anywhere database engines.

Why transactions are needed

Transactions and locks are both needed to ensure that the data stored in a database is entered accurately and stays correct. You will no doubt wish to guard even a small personal database against corruption. If instead you are building or administering a database for a large corporation, it may contain the work of a thousand people and contain records, such as customer information, vital to the livelihood of the corporation.

To ensure data integrity it is essential that you can identify states in which the information in your database is stable and consistent. Regular backups are essential, but you must be sure that the backup contains a consistent set of information if it is to be of any use at a later date. You don't want to make a backup when through someone's changes are only half inserted.

Likewise, the identification of consistent states is essential to managing the work of concurrent users of your database. The database engine must be able to execute the commands of multiple users simultaneous to give all prompt service. Identifying distinct pieces of work facilitates the task of interleaving the commands of the various users.

Transactions are logical units of work

A **transaction** is a **logical unit of work**. Each is a sequence of logically related commands which accomplish one task and transform the database from one consistent state into another.

Transactions are **atomic**. Adaptive Server Anywhere executes all the statements within a transaction as a unit. At the end of each transaction, you **commit** your changes to make them permanent. If for any reason all the commands in the transaction do not process properly, then any intermediate changes are undone, or **rolled back**.

Transactions are key to both the management of concurrent users and to the protection of the database from media and system failures. Transactions break the work of each user into small blocks. These blocks may be safely interleaved and the completion of each block marks a point at which the information is self-consistent

For information about database backups and data recovery, see "Backup and Data Recovery" on page 553.

For further information about concurrent database usage, see "Introduction to concurrency" on page 372

Example

Suppose you worked for a bank and wished to transfer $1000 between two people's accounts. You can accomplish the transfer by completing the following two operations:

1 Debit the first person's account.

2 Credit the second person's account.

Under ordinary circumstances, these two operations will work perfectly, but suppose that while you were in the middle of transferring the money, the database system suddenly failed. You hope that either both commands were executed and the money was transferred successfully, or that neither command was executed. If only the first command completed, then the information in the database would be inconsistent because $1000 would be missing. It is unacceptable to leave the debit in the database without recording the credit. Either both the debit and the credit must be processed, or neither. In case of failure, the debit needs to be undone.

Each transaction is processed entirely or not at all

Transaction processing ensures that each transaction is processed in its entirety or not at all. Transaction processing is fundamental to ensuring that a database contains correct information. It addresses two distinct, yet related, problems: data recovery and database consistency.

In the case of the transfer of funds, you can guard against corrupting the bank records by grouping all the necessary commands in one transaction.

Using transactions

Adaptive Server Anywhere expects you to group your commands into transactions. Knowing which commands or actions signify the start or end of a transaction lets you take full advantage of this feature.

Starting transactions

Transactions start with one of the following events:

- The first statement following a connection to a database
- The first statement following the end of a transaction

Completing transactions

Transactions complete with one of the following events:

- A COMMIT statement makes the changes to the database permanent.
- A ROLLBACK statement undoes all the changes made by the transaction.
- A statement with a side effect of an automatic commit is executed: Database definition commands, such as ALTER, CREATE, COMMENT, and DROP all have the side effect of an automatic commit.
- A disconnection from a database performs an implicit rollback.

Options in Interactive SQL

Interactive SQL provides you with two options which let you control when and how transactions end:

- If you set the option AUTO_COMMIT to ON, Interactive SQL automatically commits your results following every successful statement and automatically perform a ROLLBACK after each failed statement.
- The setting of the option COMMIT_ON_EXIT controls what happens to uncommitted changes when you exit Interactive SQL. If this option is set to ON (the default), Interactive SQL does a COMMIT; otherwise it undoes your uncommitted changes with a ROLLBACK statement.

☞ Adaptive Server Anywhere also supports Transact-SQL commands, such as BEGIN TRANSACTION, for compatibility with Sybase Adaptive Server Enterprise. For further information, see "Transact-SQL Compatibility" on page 781

☞ You may identify important states within a transaction and return to them selectively using SAVEPOINTS. These are discussed further in section "Savepoints within transactions" on page 418.

Transactions and data recovery

Suppose that a system failure or power outage suddenly takes your database engine down. Adaptive Server Anywhere is carefully designed to protect the integrity of your database in such circumstances. It provides you with a number of independent means of restoring your database. For example, it provides you with a log file which you may store on a separate drive so that should the system failure damage one drive, you still have a means of restoring your data.

In such circumstances, transaction processing allows the database server to identify states in which your data is in a consistent state. Transaction processing ensures that if, for any reason, a transaction is not successfully completed, then the entire transaction is undone, or rolled back. The database is left entirely unaffected by failed transactions.

Adaptive Server Anywhere's transaction processing ensures that the contents of a transaction are processed securely, even in the event of a system failure in the middle of a transaction.

For a detailed description of data recovery mechanisms, see chapter "Backup and Data Recovery" on page 553.

The remainder of this chapter is devoted to concurrency and consistency of transactions.

Introduction to concurrency

Concurrency means processing more than one transaction at the same time

Adaptive Server Anywhere can execute more than one transaction at the same time. The term **concurrency** refers to this ability. Were it not for special mechanisms within the database server, concurrent transactions could interfere with each other to produce inconsistent and incorrect information.

A transaction is a logical unit of work. When a database engine executes transactions sequentially, the database is advanced step by step from one consistent state to the next.

To respond promptly to requests from various users, the database engine must execute statements from other users without waiting for a first user to complete a transaction. Without a control mechanism, the actions of the various users could interfere with each other. For example, two users might try to update the same price at the same time.

Individual users group their work into transactions which, when executed in isolation, modify the database in a safe manner. When executing more than one transaction at once, the database server must take precautions to maintain this security and restrict interference between transactions. Interference can cause the database to end up in a state which is not possible to obtain by executing the separate transactions one at a time.

Transactions processed at the same time are said to be **concurrent**. While executing the SQL statements which comprise one transaction, the database engine can execute some or all of the statements in other transactions. All Sybase database engines and servers can execute multiple transactions concurrently.

Why concurrency benefits you

Often, databases form common repositories of information and are shared by a large number of people. These people may need frequent access to the information as part of their jobs. To avoid impeding their work, the database engine must be able to process many transactions at the same time.

Adaptive Server Anywhere allows many simultaneous connections to one database. Usually, these connections are formed by separate users. Concurrent transaction processing means not only that a database engine can accept multiple connections, but that it can also process transactions from more than one connected user or application simultaneously.

Example

Consider a database for a department store. The database system must allow many clerks to update customer accounts simultaneously. Each clerk must be able to update the status of the accounts as they assist each customer. Each clerk cannot afford to wait until no one else is using the database.

Who needs to know about concurrency

Concurrency is a concern to all database administrators and developers. Even if you are working with a single-user database, you must be concerned with concurrency. In addition to connections from multiple users, your Adaptive Server Anywhere database can accept separate connections from multiple applications and even multiple connections from a single application. These applications and connections can interfere with each other in exactly the same way as multiple users in a network setting.

Tutorial 1: The dirty read

The following tutorial demonstrates one type of inconsistency which can occur when multiple transactions are executed concurrently. Two employees at a typical small merchandising company both access the corporate database at the same time. The first person is the company's Sales Manager. The second is the Accountant.

The Sales Manager wants to increase the price of one of the tee shirts sold by their firm by 95¢, but is having little trouble with the syntax of the SQL language. At the very same time, unbeknownst to the Sales Manager, the Accountant is trying to calculate the retail value of the current inventory to include in a report he volunteered to bring to the next management meeting.

In this example, you will play the role of two people, both using the demonstration database concurrently.

1 Start Interactive SQL.

2 Connect to the sample database: Select Connect from the Command menu. Enter the user ID **DBA** and the password **SQL**. In the Advanced tab of the connection window, name this connection **Sales Manager**.

User ID:	**DBA**
Password:	**SQL**
Connection Name:	**Sales Manager**
Database File:	**asademo**

3 Start a second copy of Interactive SQL.

4 Again enter **DBA** and **SQL** as the user ID and password, but this time name the connection **Accountant**. Click the OK button.

User ID:	**DBA**
Password:	**SQL**
Connection Name:	**Accountant**

5 Pretend you are the Sales Manager. You have decided to raise the price of all the tee shirts by 95¢. In the window labeled "Sales Manager," execute the following commands:

```
SELECT id, name, unit_price
FROM product;
```

```
UPDATE PRODUCT
SET unit_price = unit_price + 95
WHERE NAME = 'Tee Shirt';
```

id	name	unit_price
300	Tee Shirt	104.00
301	Tee Shirt	109.00
302	Tee Shirt	109.00
400	Baseball Cap	9.00
401	Baseball Cap	10.00
500	Visor	7.00
501	Visor	7.00
600	Sweatshirt	24.00
601	Sweatshirt	24.00
700	Shorts	15.00

You observe immediately that you should have entered 0.95 instead of 95, but before you can fix your error, the accountant accesses the database from another office.

6 The company's accountant is worried that too much money is tied up in inventory. Pretend you are the accountant. In the window named "Accountant," execute the following commands to calculate the total retail value of all the merchandise in stock:

```
SELECT SUM( quantity * unit_price )
  AS inventory
FROM product;
```

inventory
21453.00

Unfortunately, this calculation isn't accurate. The Sales Manager accidentally raised the price of the visor $95, and the result reflects this erroneous price. This mistake demonstrates one typical type of inconsistency known as a **dirty read**. You, as the Accountant, accessed data which the Sales Manager has entered, but has not yet committed.

☞ You can eliminate dirty reads and other inconsistencies explained in "Isolation levels and consistency" on page 386.

7 Return to the role of Sales Manager. In the first window fix the error by rolling back your first changes and entering the correct UPDATE command. Check that your new values are correct.

```
ROLLBACK;

UPDATE product
SET unit_price = unit_price + 0.95
WHERE NAME = 'Tee Shirt';
```

id	name	unit_price
300	Tee Shirt	9.95
301	Tee Shirt	14.95
302	Tee Shirt	14.95
400	Baseball Cap	9.00
401	Baseball Cap	10.00
500	Visor	7.00
501	Visor	7.00
600	Sweatshirt	24.00
601	Sweatshirt	24.00
700	Shorts	15.00

8 The Accountant does not know that the amount he calculated was in error. You can see the correct value by executing his SELECT statement again in his window.

```
SELECT SUM( quantity * unit_price )
  AS inventory
FROM product;
```

inventory
6687.15

9 Finish the transaction in the Sales Manager's window. She would enter a COMMIT statement to make his changes permanent, but you may wish to enter a ROLLBACK, instead, to avoid changing the copy of the demonstration database on your machine.

```
ROLLBACK;
```

The accountant unknowingly receives erroneous information from the database because the database engine is processing the work of both the Sales Manager and the Accountant simultaneously. The following section introduces other types of inconsistencies and introduces means to eliminate them.

Using locks to ensure consistency

Row-level locking

Adaptive Server Anywhere uses row-level locking to allow transactions to execute concurrently without interference, or with limited interference. Any transaction can acquire a lock to prevent other concurrent transactions from modifying or even accessing a particular row. This row-level locking scheme always stops some types of interference. For example, a transaction which is updating a particular row of a table always acquires a lock on that row to ensure that no other transaction can update or delete the same row at the same time.

Some inconsistency may be tolerable

Inconsistency in the information an application sees is tolerable in some cases. Therefore, you do not need to prohibit all forms of inconsistent behavior in all cases. For this reason, Adaptive Server Anywhere grants you control over the level of consistency required in the information any transaction sees.

In the above example, you were able to observe the Sales Manager's uncommitted results in the Accountant's window. This type of inconsistency is known as a **dirty read**. You were able to observe this problem because some locking features are turned off in the demonstration database allowing this type of inconsistency to occur. Various locking options let you eliminate such interference.

Improving concurrency

Transaction Size Affects Concurrency

The way you group SQL statements into transactions can have significant effects on data integrity and on system performance. If you make a transaction too short and it does not contain an entire logical unit of work, then inconsistencies can be introduced into the database. If you write a transaction which is too long and contains several unrelated actions, then there is greater chance that a ROLLBACK will unnecessarily undo work that could have been committed quite safely into the database.

Locks obtained during a transaction are not released until a COMMIT or ROLLBACK statement is issued. If your transactions are long and lock large amounts of data, they can lower concurrency by preventing other transactions from being processed simultaneously.

There are many factors which determine the appropriate length of a transaction, depending on the type of application and the environment. Some guidelines are given towards the end of this chapter.

Use fewer locks for better concurrency

The number of locks which your transactions place can affect the level of concurrency which your database can support. Each lock limits the access to some specific row of a table. If another transaction should require access to the same row, then it may have to wait until your transaction completes. The more locks you place, the more likely it is that other transactions will be delayed. You should use sufficient locking to ensure the integrity of your data, but it is good practice to use the lowest level of locking which will suffice. Doing so allows your database to process transactions as quickly as possible.

The locking scheme and the options which allow you to limit interference and control locking are discussed in detail later in this chapter.

Typical inconsistencies

Adaptive Server Anywhere affords you control over the amount of locking it uses to isolate transactions. For example, you can eliminate inconsistencies such as the dirty read demonstrated in the previous example.

This section revisits the dirty read and introduces two other typical types of inconsistencies which you may encounter. Knowledge of these inconsistencies will help you select appropriate levels of isolation for the transactions in your own databases.

It then proceeds to introduce a fourth type of inconsistency known as lost updates. These are particularly interested to you if you use something known as cursors in your SQL programs.

Three typical types of inconsistency

There are three typical types of inconsistency that can occur during the execution of concurrent transactions. This list is not exhaustive as other types of inconsistencies can also occur. These three types are mentioned in the ISO SQL/92 standard and are important because behavior at lower isolation levels is defined in terms of them.

You will recognize the first type as the one you demonstrated in the previous tutorial.

- **Dirty read** Transaction A modifies a row, but does not commit or roll back the change. Transaction B reads the modified row. Transaction A then either further changes the row before performing a COMMIT, or rolls back its modification. In either case, transaction B has seen the row in a state which was never committed.
- **Non-repeatable read** Transaction A reads a row. Transaction B then modifies or deletes the row and performs a COMMIT. If transaction A then attempts to read the same row again, the row will have been changed or deleted.
- **Phantom row** Transaction A reads a set of rows that satisfy some condition. Transaction B then executes an INSERT, or an UPDATE on a row which did not previously meet A's condition. Transaction B commits these changes. These newly committed rows now satisfy the condition. Transaction A then repeats the initial read and obtains a different set of rows.

Other types of inconsistencies can also exist. These three were chosen for the ISO SQL/92 standard because they are typical problems and because it was convenient to describe amounts of locking between transactions in terms of them.

☞ Amounts of locking will be described further in "Isolation levels and consistency" on page 386.

Correctness

To process transactions concurrently, the database engine must execute some component statements of one transaction, then some from other transactions, before continuing to process further operations from the first. The order in which the component operations of the various transactions are interwoven is called the **schedule**.

Applying transactions concurrently in this manner can result in many possible outcomes, including the three particular inconsistencies described in the previous section. Sometimes, the final state of the database also could have been achieved had the transactions been executed sequentially, meaning that one transaction was always completed in its entirety before the next was started. A schedule is called **serializable** whenever executing the transactions sequentially, in *some* order, could have left the database in the same state.

Serializability is the commonly accepted criterion for correctness. A serializable schedule is accepted as correct because the database is not influenced by the concurrent execution of the transactions.

Serializable means that concurrency has added no effect

Even when transactions are executed sequentially, the final state of the database can depend upon the order in which these transactions are executed. For example, if one transaction sets a particular cell to the value 5 and another sets it to the number 6, then the final value of the cell is determined by which transaction executes last.

Knowing a schedule is serializable does not settle which order transactions would best be executed, but rather states that concurrency has added no effect. Outcomes which may be achieved by executing the set of transactions sequentially in some order are *all assumed correct*.

Unserializable schedules introduce inconsistencies

The three types of inconsistencies introduced in the previous section are typical of the types of problems which appear when the schedule is not serializable. In each case, the inconsistency appeared because the statements were interleaved in such a way as to produce a result that would not be possible if all transactions were executed sequentially. For example, a dirty read can only occur if one transaction can select rows while another transaction is in the middle of inserting or updating data.

Two-phase locking

Two-phase locking is important in the context of ensuring that schedules are serializable. The **two-phase locking protocol** specifies a procedure each transaction should follow.

This protocol is important because, if observed by all transactions, it will guarantee a serializable, and thus correct, schedule. It may also help you understand why some methods of locking permit some types of inconsistencies.

The two-phase locking protocol

1 Before operating on any row, a transaction must acquire a lock on that row.

2 After releasing a lock, a transaction must never acquire any more locks.

In practice, a transaction normally holds locks until it terminates with either a COMMIT or ROLLBACK statement. Releasing locks before the end of the transaction disallows the operation of rolling back the changes whenever doing so would necessitate operating on rows to return them to an earlier state

The two-phase locking protocol allows the statement of the following important theorem:

The two-phase locking theorem
If all transactions obey the two-phase locking protocol, then all possible interleaved schedules are serializable.

In other words, if all transactions follow the two-phase locking protocol, then none of the inconsistencies mentioned above are possible.

This protocol defines the operations necessary to ensure complete consistency of your data, but you may decide that some types of inconsistencies are permissible during some operations on your database. Eliminating all inconsistency often means reducing the efficiency of your database. Thus, Adaptive Server Anywhere affords you control the correctness of each transaction through the use of various types of locking.

How locking works

When the database engine processes a transaction, it can lock one or more rows of a table. The locks maintain the reliability of information stored in the database by preventing concurrent access by other transactions. They also improve the accuracy of result queries by identifying information which is in the process of being updated.

The Adaptive Server Anywhere engine places these locks automatically and needs no explicit instruction. It holds all the locks acquired by a transaction until the transaction is completed, for example by either a COMMIT or ROLLBACK statement, with a single exception noted below.

The transaction that has access to the row is said to hold the lock. Depending on the type of lock, other transactions may have limited access to the locked row, or none at all.

Adaptive Server Anywhere allows users to determine the *amount* of locking employed in each connection. The amount of locking controls the correctness of the schedules. Unfortunately it also affects concurrency because locks placed by one transaction may delay or obstruct the progress of another.

☞ Amounts of locking will be described further in "Isolation levels and consistency" on page 386.

Objects that can be locked

Adaptive Server Anywhere uses row-level locking, meaning that each lock locks a row, rather than always locking an entire page or range of rows. Some transactions may require locks on many rows, or rows that meet a specific criterion. To meet these requirements, locks can be placed on the flowing objects.

- **rows in tables** A transaction can lock a particular row, for example, to prevent another transaction from changing it.
- **scan positions in either indexed or sequential scans** Transactions typically scan rows using the ordering imposed by an index, or scan rows sequentially. In either case, a lock can be placed on the scan position. For example, placing a lock in an index can prevent another transaction from inserting a row with a specific value or range of values.
- **table schemas** A transaction can lock the schema of a table, preventing other transactions from modifying the table's structure.

Of these objects, the most intuitive are likely rows. It is understandable that a transaction reading, updating, deleting, or inserting a row should limit the simultaneous access to other transactions. Similarly, a transaction changing the structure of a table, perhaps inserting a new column, could greatly impact other transactions. In such a case, it is essential to limit the access of other transactions to prevent errors.

Scan positions

A **scan position** is a location in an ordering of rows of a table. In this case, there are only two possible types of ordering affected. First, rows can be ordered through use of an index, based on a particular criterion established when the index was constructed. Secondly, when performing a sequential scan of a table, Adaptive Server Anywhere must select an order in which to process the rows.

In the case of a sequential scan, the specific ordering is defined by the internal workings of the database engine. You should not rely on the order of rows in a sequential scan. From the point of view of scanning the rows, however, Adaptive Server Anywhere treats the request similarly to an indexed scan, albeit using an ordering of its own choosing. It can place locks on positions in the scan as it would were it using an index.

Through locking a scan position, a transaction prevents some actions by other transactions relating to a particular range of values in that ordering of the rows. Phantom and anti-phantom locks are always placed on scan positions.

For example, a transaction might delete a row, hence deleting a particular primary key value. Until this transaction either commits the change or rolls it back, it must protect its right to do either. In the case of a deleted row, it must ensure that no other transaction can insert a row using the same primary key value, hence making a rollback operation impossible. A lock on the scan position this row occupied reserves this right while having the least impact on other transactions.

The four types of locks

Adaptive Server Anywhere uses four distinct types of locks to implement its locking scheme and ensure appropriate levels of isolation between transactions:

- **read lock** (shared)
- **phantom lock** (shared)
- **write lock** (exclusive)
- **anti-phantom lock** (shared)

Each of these locks has a separate purpose. They work together and all are needed. Each addresses a particular set of inconsistencies which would occur in their absence. Depending on the isolation level you select, the database server will use some or all of them to maintain the degree of consistency you require.

Exclusive versus shared locks

These four types of locks each fall into one of two categories:

- **Exclusive locks** Only one transaction can hold an exclusive lock on the row of a table at one time. No transaction can obtain an exclusive lock while any other transaction holds a lock of any type on the same row. Once a transaction acquires an exclusive lock, requests to lock the row by other transactions will be denied.

  Write locks are exclusive.

- **Shared locks** Any number of transactions may acquire shared locks on any one row at the same time. Shared locks are sometimes referred to as non-exclusive locks.

  Read locks, phantom locks, and anti-phantom locks are shared.

Only one transaction should change any one row at one time. Otherwise, two simultaneous transaction might try to change one value to two different new ones. Hence, a write lock should be, and is, exclusive.

By contrast, no difficulty arises if more than one transaction wants to read a row. Since neither is changing it, there is no conflict of interest. Hence, read locks may be shared.

You may apply similar reasoning to phantom and anti-phantom locks. Many transactions can prevent the insertion of a row in a particular scan position by each acquiring an phantom lock. Similar logic applies for anti-phantom locks. When a particular transaction requires exclusive access, it can easily achieve exclusive access by obtaining both a phantom and an anti-phantom lock on the same row. These locks to not conflict when they are held by the same transaction.

Which specific locks conflict?

The following table identifies the combination of locks that conflict.

	read	**write**	**phantom**	**anti-phantom**
read		conflict		
write	conflict	conflict		
phantom				conflict
anti-phantom			conflict	

These conflicts arise only when the locks are held by different transactions. For example, one transaction can obtain both phantom and anti-phantom locks on a single scan position to obtain exclusive access to a location.

Uses for locks

The above types of locks have the following uses:

- A transaction acquires a **write lock** whenever it inserts, updates, or deletes a row. No other transaction can obtain either a read or a write lock on the same row when a write lock is set. A write lock is an exclusive lock.
- A transaction can acquire a **read lock** when it reads a row. Several transactions can acquire read locks on the same row (a read lock is a shared or nonexclusive lock). Once a row has been read locked, no other transaction can obtain a write lock on it. Thus, a transaction can ensure that no other transaction modifies or deletes a row by acquiring a read lock.
- A **phantom lock** is a shared lock placed on an indexed scan position to prevent phantom rows. It prevents other transactions from inserting a row into a table immediately before the row which is phantom locked. Phantom locks for lookups using indexes require a read lock on each row that is read, and one extra read lock to prevent insertions into the index at the end of the result set. Phantom rows for lookups that do not use indexes require a read lock on all rows in a table to prevent insertions from altering the result set, and so can have a bad effect on concurrency.
- An **anti-phantom lock** is a shared lock placed on an indexed scan position to reserve the right to insert a row. Once one transaction acquires an anti-phantom lock on a row, no other transaction can acquire a phantom lock on the same row. A read lock on the corresponding row is always acquired at the same time as an anti-phantom lock to ensure that no other process can update or destroy the row, thereby bypassing the anti-phantom lock.

Adaptive Server Anywhere uses these four types of locks as necessary to ensure the level of consistency that you require. You need not explicitly request the use of a particular lock. You control the level of consistency, as is explained in the next section. Knowledge of the types of locks will guide you in choosing isolation levels and understanding the impact of each level on performance.

Isolation levels and consistency

There are four isolation levels

Adaptive Server Anywhere allows you to control the degree to which the operations in one transaction are visible to the operations in other concurrent transactions. You do so by setting a database option called the **isolation level**. Adaptive Server Anywhere has four different isolation levels that prevent some or all inconsistent behavior. These four isolation levels are numbered from 0 through 3. Level 3 provides the highest level of isolation. At level 3, all schedules are serializable.

For example, you can set the isolation for the current connection to level 3 by executing the following statement:

```
SET TEMPORARY OPTION ISOLATION_LEVEL = 3
```

For information about serializable schedules see "Correctness" on page 380.

Lower levels allow more inconsistencies, but you will find them useful when you must give your database a high level of concurrency. Because transactions at lower isolation levels use fewer locks, they tend to reduce blocking. It is less likely that one transaction will need access to rows for which another transaction has acquired a lock.

All isolation levels guarantee that each transaction will execute completely or not at all, and that no updates will be lost. Adaptive Server Anywhere therefore ensures recoverability at all times, regardless of the isolation level.

Isolation levels and dirty reads, non-repeatable reads, and phantom rows

The isolation levels are different with respect to dirty reads, non-repeatable reads, and phantom rows. The four isolation levels have different names under ODBC, as shown in the bottom row of the table. An **x** means that the behavior is prevented, and a ✓ means that the behavior may occur.

Isolation level	0	1	2	3
Dirty reads	✓	**x**	**x**	**x**
Non-repeatable reads	✓	✓	**x**	**x**
Phantom rows	✓	✓	✓	**x**
SQLCA.lock	**RU**	**RC**	**RR**	**TS**

This table demonstrates two points:

- Each isolation level eliminates of the three typical types of inconsistencies.
- Each level eliminates the types of inconsistencies eliminated at all lower levels.

Isolation levels that prevent lost updates

Read locks prevent lost updates. When a transaction first reads a value, it acquires a read lock on that row. Now, should another transaction also wish to update the row, it must acquire a write lock on the row. Since write locks are exclusive locks, the database server will not grant a write lock to a transaction while any other transaction holds a read lock on the same row. No updates can be lost.

Setting the isolation level

The isolation level is a database option. You change database options using the SET OPTION statement. For example, the following command sets the isolation level to 3, the highest level.

```
SET OPTION ISOLATION_LEVEL = 3;
```

Each connection to the database has its own isolation level. In addition, the database can store a default isolation level for each user or group. You can change the isolation of your connection and the default level associated with your user ID using the SET command. If you have permission, you can also change the isolation level for other users or groups.

In fact, the above command changes both the isolation level for your present connection, and also changes the default level associated with your user ID. Thus, if you form a second connection after executing this command, the isolation will initially be set to level 3.

When you connect to a database, the database server determines your initial isolation level as follows:

1 A default isolation level may be set for each user and group. If a level is stored in the database for your user ID, then the database server uses it.

2 If not, the database server checks the groups to which you belong until it finds a level. All users are members of the special group PUBLIC. If it finds no other setting first, then Anywhere will use the level assigned to that group.

Once connected to the database, you can change the isolation level for your connection using the TEMPORARY option of the SET command. Temporary options stay in effect only as long as you remain connected. They do not change defaults stored for your user ID. They affect only the connection in which the command is executed. To set your isolation level to level 2 for the duration of your present session, you use the following command.

```
SET TEMPORARY OPTION ISOLATION_LEVEL = 2;
```

To set the option for a group, prepend the name of the group and a period to ISOLATION_LEVEL. For example, the following command sets the default isolation for the special group PUBLIC.

```
SET OPTION PUBLIC.ISOLATION_LEVEL = 3;
```

To change the default isolation level of a group you must have permission to do so. Because the group PUBLIC is special, you must have dba privilege to change settings associated with it.

Setting temporary options for the PUBLIC group has a special effect. The setting stays in effect only as long as the present database engine remains in operation. Should the database engine shut down, the original default will appear when the database engine is restarted. Again, because PUBLIC is a special group, you must have dba privilege to set its options.

For further information about users and groups, please refer to "Managing User IDs and Permissions" on page 575.

For a description of the SET OPTION statement syntax, see "SET OPTION statement" on page 553 of the book *Adaptive Server Anywhere Reference Manual*

You may wish to change the isolation level in mid-transaction if, for example, just one table or group of tables requires serialized access. For information about changing the isolation level with in a transaction, see "Changing the isolation level within a transaction" on page 394.

Setting the isolation level from an ODBC-enabled application

ODBC uses the isolation feature to support assorted database lock options. For example, in PowerBuilder you can use the Lock attribute of the transaction object to set the isolation level when you connect to the database. The Lock attribute is a string, and is set as follows:

```
// Set the lock attribute to read uncommitted
// in the default transaction object SQLCA.
SQLCA.lock = "RU"
```

When is Lock honored?
This option is honored only at the moment the CONNECT occurs. Changes to the Lock attribute after the CONNECT have no effect on the connection.

Understanding and choosing isolation levels

The choice of isolation level depends on the kind of task an application is carrying out. This section gives some guidelines for choosing isolation levels.

When you choose an appropriate isolation level you must balance the need for consistency and accuracy in the information your transaction is using, with the need for concurrent transactions to proceed unimpeded. If a transaction involves only one or two specific values in one table, it is unlikely to interfere as much with other processes as one which searches many large tables and may need to lock many rows or entire tables and may take a very long time to complete.

If your transactions is transferring money between bank accounts or even checking account balances, you will likely want to do your utmost to ensure that the information you return is correct. On the other hand, if just want a rough estimate of the proportion of inactive accounts, then you may not care whether your transaction waits for others or not and indeed may be willing to sacrifice some accuracy to avoid interfering with other users of the database.

Furthermore, a transfer may affect only the two rows which contain the two account balances, whereas all the accounts must be read in order to calculate the estimate. For this reason, the transfer is less likely to delay other transactions.

Adaptive Server Anywhere provides four levels of isolation: levels 0, 1, 2, and 3. The third level provides complete isolation and ensures that transactions are interleaved in such a manner that the schedule is serializable.

For a discussion of serializable schedules and correctness please refer to section "Correctness" on page 380.

Tutorial 2 – The non-repeatable read

The example in section "Introduction to concurrency" on page 372 demonstrated the first type of inconsistency, namely the dirty read. In that example, an accountant made a calculation while the Sales Manager was in the process of updating a price. The accountant's calculation used erroneous information which the Sales Manager had entered and was in the process of fixing.

The following example demonstrates one type of inconsistency, namely non-repeatable reads. In this example, you will play the role of the same two people, both using the demonstration database concurrently. The Sales Manager wishes to offer a new sales price on plastic visors. The Accountant wishes to verify the prices of some items that appear on a recent order.

This example begins with both connections at isolation level 1, rather than at isolation level 0, which is the default for the demonstration database supplied with Adaptive Server Anywhere. By setting the isolation level to 1, you eliminate the type of inconsistency which the previous tutorial demonstrated, namely the dirty read.

1 Start Interactive SQL.

2 Connect to the **asademo** database: Select Connect from the Command menu. Enter the user ID **DBA** and the password **SQL**. In the Advanced tab, name this connection **Sales Manager**.

User ID:	**DBA**
Password:	**SQL**
Connection Name:	**Sales Manager**
Database File:	**asademo**

3 Start a second copy of Interactive SQL. Again, enter **DBA** and **SQL** as the user ID and password, but this time name the connection **Accountant**. Click the OK button.

User ID:	**DBA**
Password:	**SQL**
Connection Name:	**Accountant**

4 Set the isolation level to 1 for the Accountant's connection by executing the following command.

```
SET TEMPORARY OPTION ISOLATION_LEVEL = 1;
```

5 Set the isolation level to 1 in the Sales Manager's window by executing the following command:

```
SET TEMPORARY OPTION ISOLATION_LEVEL = 1;
```

6 The Accountant decides to list the prices of the visors. In the Accountant's window, execute the following command:

```
SELECT id, name, unit_price FROM product
```

id	name	unit_price
300	Tee Shirt	9.00
301	Tee Shirt	14.00
302	Tee Shirt	14.00
400	Baseball Cap	9.00
401	Baseball Cap	10.00
500	Visor	7.00
501	Visor	7.00

7 The Sales Manager decides to introduce a new sale price for the plastic visor. In the Sales Manager's window, execute the following command:

```
SELECT id, name, unit_price FROM product
WHERE name = 'Visor';

UPDATE product
SET unit_price = 5.95 WHERE id = 501;

COMMIT;
```

id	name	unit_price
500	Visor	7.00
501	Visor	5.95

8 Compare the price of the visor in the Sales Manager's window with the price for the same visor in the Accountant's window. The Accountant's window still shows the old price, even though the Sales Manager has entered the new price and committed the change.

This inconsistency is called a *non-repeatable read*, because if the accountant did the same select a second time in the *same transaction*, he wouldn't get the same results. Try it for yourself. In the Accountant's window execute the select command again. Observe that the Sales Manager's sale price now displays.

```
SELECT id, name, price
FROM products;
```

id	name	unit_price
300	Tee Shirt	9.00
301	Tee Shirt	14.00
302	Tee Shirt	14.00
400	Baseball Cap	9.00
401	Baseball Cap	10.00
500	Visor	7.00
501	Visor	5.95

Of course if the accountant had finished his transaction, for example by issuing a COMMIT or ROLLBACK command before using the SELECT again, it would be a different matter. The database is available for simultaneous use by multiple users and it is completely permissible for someone to change values either before or after the accountant's transaction. The change in results is only inconsistent because it happens in the middle of his transaction. Such an event makes the schedule unserializable.

9 The accountant notices this behavior and decides that from now on he doesn't want the prices changing while he looks at them. Repeatable reads are eliminated at isolation level 2. Play the role of the accountant:

```
SET TEMPORARY OPTION ISOLATION_LEVEL = 2;

SELECT id, name, unit_price
FROM product;
```

10 The Sales Manager decides that it would be better to delay the sale on the plastic visor until next week so that she won't have to give the lower price on a big order that she's expecting will arrive tomorrow. In his window, try to execute the following statements. The command will start to execute, then his window will appear to freeze.

```
UPDATE product
SET unit_price = 7.00 WHERE id = 501;
```

The database server must guarantee repeatable reads at isolation level 2. To do so, it places a read lock on each row of the product table that the accountant reads. When the Sales Manager tries to change the price back, his transaction must acquire a write lock on the plastic visor row of the product table. Since write locks are exclusive, his transaction must wait until the Accountant's transaction releases its read lock.

11 The Accountant is finished looking at the prices. He doesn't want to risk accidentally changing the database, so he completes his transaction with a ROLLBACK statement.

```
ROLLBACK;
```

Observe that as soon as the database server executes this statement, the Sales Manager's transaction completes.

id	name	unit_price
500	Visor	7.00
501	Visor	7.00

12 The Sales Manager can finish now. She wishes to commit his change to restore the original price.

```
COMMIT;
```

Types of Locks and different isolation levels

When you upgraded the accountant's isolation from level 1 to level 2, the database server used read locks where none were previously acquired. In general, each isolation level is characterized by the types of locks needed and by how locks held by other transactions are treated.

At isolation level 0, the database server needs only write locks. It makes use of these locks to ensure that no two transactions make modifications which conflict. For example, a level 0 transaction acquires a write lock on a row before it updates or deletes it, and inserts any new rows with a write lock already in place.

Level 0 transactions perform no checks on the rows they are reading. For example, when a level 0 transaction reads a row, it doesn't bother to check what locks may or may not have been acquired on that row by other transactions. Since no checks are needed, level 0 transactions are particularly fast. This speed comes at the expense of consistency. Whenever they read a row which is write locked by another transaction, they risk returning dirty data.

At level 1, transactions check for a write locks before they read a row. Although one more operation is required, these transactions are assured that all the data they read is committed. Try repeating the first tutorial with the isolation level set to 1 instead of 0. You will find that the Accountant's computation cannot proceed while the Sales Manager's transaction, which updates the price of the tee shirts, remains incomplete.

When the Accountant raised his isolation to level 2, the database server began using read locks. From then on, it acquired a read lock for his transaction on each row that matched his selection.

Changing the isolation level within a transaction

You may not have thought about it, but in doing the above tutorial, you demonstrated another important feature of Adaptive Server Anywhere, namely that the isolation level may be changed in the middle of a transaction. The accountant performed one SELECT, upgraded his isolation to level 2, and then performed a second SELECT.

When you change the ISOLATION_LEVEL option in the middle of a transaction, the new setting affects only the following:

- Any cursors opened after the change
- Any statements executed after the change

You may wish to change the isolation level during a transaction, as doing so affords you control over the number of locks your transaction places. You may find a transaction needs to read a large table, but perform detailed work with only a few of the rows. If an inconsistency would not seriously affect your transaction, set the isolation to a low level while you scan the large table to avoid delaying the work of others.

You may also wish to change the isolation level in mid transaction if, for example, just one table or group of tables requires serialized access.

Transaction blocking

In step 10 of the above tutorial, the Sales Manager's screen froze during the execution of his UPDATE command. The database server began to execute his command, then found that the Accountant's transaction had acquired a read lock on the row that the Sales Manager needed to change. At this point, the database server simply paused the execution of the UPDATE. Once the Accountant finished his transaction with the ROLLBACK, the database server automatically released his locks. Finding no further obstructions, it then proceeded to complete execution of the Sales Manager's UPDATE.

In general, a locking conflict occurs when one transaction attempts to acquire an exclusive lock on a row on which another transaction holds a lock, or attempts to acquire a shared lock on a row on which another transaction holds an exclusive lock. One transaction must wait for another transaction to complete. The transaction which must wait is said to be **blocked** by another transaction.

When the database server identifies a locking conflict which prohibits a transaction from proceeding immediately, it can either pause execution of the transaction, or it can terminate the transaction, roll back any changes, and return an error. You control the route by setting the BLOCKING option. When BLOCKING is set to ON, then the second transaction waits as in the above tutorial

For further information regarding the blocking option, see "The BLOCKING option" on page 416.

Tutorial 3 – A phantom row

The following continues the same scenario. In this case, the Accountant views the department table while the Sales Manager creates a new department. You will observe the appearance of a phantom row.

If you have not done so, do steps 1 through 4 of the previous tutorial. These steps describe how to open two copies of Interactive SQL.

1 Start two copies of Interactive SQL as in steps 1 through 3 of the previous tutorial. Name one connection **Sales Manager**. Name the other connection **Accountant**.

User ID:	**DBA**
Password:	**SQL**
Connection Name:	**Sales Manager**/ **Accountant**
Database File:	**asademo**

2 Set the isolation level to 2 in the Sales Manager's window by executing the following command.

```
SET TEMPORARY OPTION ISOLATION_LEVEL = 2;
```

3 Set the isolation level to 2 for the Accountant's connection by executing the following command.

```
SET TEMPORARY OPTION ISOLATION_LEVEL = 2;
```

4 In the Accountant's window, enter the following command to list all the department.

```
SELECT * FROM department
ORDER BY dept_id;
```

dept_id	dept_name	dept_head_id
100	R & D	501
200	Sales	902
300	Finance	1293
400	Marketing	1576
500	Shipping	703

5 The Sales Manager decides to set up a new department to focus on the foreign market. Philip Chin, who has number 129, will head the new department.

```
INSERT INTO department
   (dept_id, dept_name, dept_head_id)
   VALUES(600, 'Foreign Sales', 129);
```

The final command creates the new entry for the new department. It appears as a new row at the bottom of the table in the Sales Manager's window.

6 The Accountant, however, is not aware of the new the new department. At isolation level 2, the database server places locks to ensure that no row changes, but places no locks that stop other transactions from inserting new rows

The Accountant will only discover the new row if he should execute his select command again. In the Accountant's window, execute the SELECT statement again. You will see the new row appended to the table.

```
SELECT * FROM department
ORDER BY dept_id;
```

dept_id	dept_name	dept_head_id
100	R & D	501
200	Sales	902
300	Finance	1293
400	Marketing	1576
500	Shipping	703
600	Foreign Sales	129

The new row that appears is called a **phantom row** because, from the Accountant's point of view, it appears like an apparition, seemingly from nowhere. The Accountant is connected at isolation level 2. At that level, the database server acquires locks only on the rows that he is using. Other rows are left untouched and hence their is nothing to prevent the Sales Manager from inserting a new row.

7 The Accountant would prefer to avoid such surprises in future, so he raises the isolation level of his current transaction to level 3. Enter the following commands for the Accountant.

```
SET TEMPORARY OPTION ISOLATION_LEVEL = 3;

SELECT * FROM department
ORDER BY dept_id;
```

8 The Sales Manager would like to add a second department to handle sales initiative aimed at large corporate partners. Execute the following command in the Sales Manager's window.

```
INSERT INTO department
   (dept_id, dept_name, dept_head_id)
VALUES(700, 'Major Account Sales', 902);
```

The Sales Manager's window will pause during execution. The command is blocked by the Accountant's locks. Click the **Stop** button to interrupt this entry.

9 To avoid changing the demonstration database that comes with Adaptive Server Anywhere, you should roll back the insertion of the new departments. Execute the following command in the Sales Manager's window:

```
ROLLBACK;
```

When the Accountant raised his isolation to level 3 and again selected all rows in the department table, the database server placed phantom locks on each row in the table, and one extra phantom lock to avoid insertion at the end of the table. When the Sales Manager attempted to insert a new row at the end of the table, it was this final lock that blocked his command.

Notice that the Sales Manager's command was blocked even though the Sales Manager is still connected at isolation level 2. the database server places phantom locks, like read locks, as demanded by the isolation level and statements of each transactions. Once placed, these locks must be respected by all other concurrent transactions.

☞ Further information on the details of the locking methods employed by Adaptive Server Anywhere is located in "How Adaptive Server Anywhere implements locking" on page 408.

Tutorial 4 – Practical locking implications

The following continues the same scenario. In this tutorial, the Accountant and the Sales Manager both have tasks that involve the sales order and sales order items tables. The Accountant needs to verify the amounts of the commission checks paid to the sales employees for the sales they made during the month of April 1994. The Sales Manager, notices that a few orders have not been added to the database and wants to add them.

Their work demonstrates phantom locking. When a transaction at isolation level 3 selects rows which match a given criterion, the database server places phantom locks to stop other transactions from inserting rows which would also match. The number of locks placed on your behalf depends both on the search criterion and on the design of your database.

If you have not done so, do steps 1 through 3 of the previous tutorial which describe how to start two copies of Interactive SQL.

1 Start two copies of Interactive SQL as in steps 1 through 3 of the previous tutorial. Name one connection **Sales Manager**. Name the other connection **Accountant**.

User ID:	**DBA**
Password:	**SQL**
Connection Name:	**Sales Manager**/ **Accountant**
Database File:	**asademo**

2 Set the isolation level to 2 in the Sales Manager's window by executing the following command.

```
SET TEMPORARY OPTION ISOLATION_LEVEL = 2
```

3 Set the isolation level to 2 for the Accountant's connection by executing the following command.

```
SET TEMPORARY OPTION ISOLATION_LEVEL = 2
```

4 Each month, the sales representatives are paid a commission which is calculated as a percentage of their sales for that month. The Accountant is preparing the commission checks for the month of April 1994. His first task is to calculate the total sales of each representative during this month.

Enter the following command in the Accountant's window. Prices, sales order information, and employee data are stored in separate tables. Join these tables using the foreign key relationships which link them to combine these necessary pieces of information.

```
SELECT emp_id, emp_fname, emp_lname,
   SUM(sales_order_items.quantity * unit_price)
      AS "April sales"
FROM employee
   KEY JOIN sales_order
   KEY JOIN sales_order_items
   KEY JOIN product
WHERE '1994-04-01' <= order_date
   AND order_date < '1994-05-01'
GROUP BY  emp_id, emp_fname, emp_lname
```

emp_id	emp_fname	emp_lname	April sales
129	Philip	Chin	2160.00
195	Marc	Dill	2568.00
299	Rollin	Overbey	5760.00
467	James	Klobucher	3228.00
667	Mary	Garcia	2712.00
690	Kathleen	Poitras	2124.00
856	Samuel	Singer	5076.00
902	Moira	Kelly	5460.00
949	Pamela	Savarino	2592.00
1142	Alison	Clark	2184.00
1596	Catherine	Pickett	1788.00

5 The Sales Manager notices a big order sold by Philip Chin was not entered into the database. Philip likes to be paid his commission promptly, so the Sales manager enters the missing order, which was placed on April 25.

In the Sales Manager's window, enter the following commands. The Sales order and the items are entered in separate tables because an one order can contain many items. You should create the entry for the sales order before you add items to it. To maintain referential integrity, the database server won't allow a transaction which adds items to an order which does not exist.

```
INSERT into sales_order
VALUES ( 2653, 174, '1994-04-22', 'r1',
      'Central', 129);

INSERT into sales_order_items
VALUES ( 2653, 1,  601, 100, '1994-04-25' );

COMMIT;
```

6 The Accountant has no way of knowing that the Sales Manager has just added a new order. Had the new order been entered earlier, it would have been included in the calculation of Philip Chin's April sales.

In the Accountant's window, calculate the April sales totals again. Use the same command, but observe that Philip Chin's April sales changes to $4560.00.

emp_id	emp_fname	emp_lname	April sales
129	Philip	Chin	4560.00
195	Marc	Dill	2568.00
299	Rollin	Overbey	5760.00
467	James	Klobucher	3228.00
667	Mary	Garcia	2712.00
690	Kathleen	Poitras	2124.00
856	Samuel	Singer	5076.00
902	Moira	Kelly	5460.00
949	Pamela	Savarino	2592.00
1142	Alison	Clark	2184.00
1596	Catherine	Pickett	1788.00

Imagine that the Accountant now marks all orders placed in April to indicate that commission has been paid. The order that the Sales Manager just entered might be found in the second search and marked as paid, even though it was not included in Philip's total April sales!

7 At isolation level 3, the database server places phantom locks to ensure that no other transactions can add a row which matches the criterion of a search or select.

First, roll back the insertion of Philip's missing order: Execute the following statement in the Sales Manager's window.

8 In the Accountant's window, execute the following two statements.

```
ROLLBACK;

SET TEMPORARY OPTION ISOLATION_LEVEL = 3;
```

9 In the Sales Manager's window, execute the following statements to remove the new order.

```
DELETE FROM sales_order_items
WHERE id = 2653;
```

```
DELETE FROM sales_order
WHERE id = 2653;

COMMIT;
```

10 In the Accountant's window, execute same query as before.

```
SELECT emp_id, emp_fname, emp_lname,
   SUM(sales_order_items.quantity * unit_price)
      AS "April sales"
FROM employee
   KEY JOIN sales_order
   KEY JOIN sales_order_items
   KEY JOIN product
WHERE '1994-04-01' <= order_date
   AND order_date < '1994-05-01'
GROUP BY  emp_id, emp_fname, emp_lname;
```

Because you set the isolation to level 3, the database server will automatically place phantom locks to ensure that the Sales Manager can't insert April order items until the Accountant finishes his transaction.

11 Return to the Sales Manager's window. Again attempt to enter Philip Chin's missing order.

```
INSERT INTO sales_order
VALUES ( 2653, 174, '1994-04-22', 'r1',
      'Central', 129);
```

The Sales Manager's window will hang; the operation will not complete. Click **Stop** to interrupt this entry.

12 The Sales Manager can't enter the order in April, but you would think that she could still enter it in May.

Change the date of the command to April 05 and try again.

```
INSERT INTO sales_order
VALUES ( 2653, 174, '1994-05-05', 'r1',
      'Central', 129);
```

The Sales Manager's window will hang again. Click **Stop** to interrupt this entry. Although the database server places no more locks than necessary to prevent insertions, these locks have the potential to interfere with a large number of other transactions.

The database server places locks in table indices. For example, it places a phantom lock in an index so a new row cannot be inserted immediately before it. However, when no suitable index is present, it must lock every row in the table.

In some situations, phantom locks may block some insertions into a table, yet allow others.

13 The Sales Manager wishes to add a second item to order 2651. Use the following command.

```
INSERT INTO sales_order_items
VALUES ( 2651, 2,  302, 4, '1994-05-22' );
```

All goes well, so the Sales Manager decides to add the following item to order 2652 as well.

```
INSERT INTO sales_order_items
VALUES ( 2652, 2,  600, 12, '1994-05-25' );
```

The Sales Manager's window will hang. Click **Stop** to interrupt this entry.

14 Conclude this tutorial by undoing any changes to avoid changing the demonstration database. Enter the following command in the Sales Manager's window.

```
ROLLBACK;
```

Enter the same command in the Accountant's window.

```
ROLLBACK;
```

You may now close both windows.

Reducing the impact of locking

You should avoid running transactions at isolation level 3 whenever practical. They tend to place large number of locks and hence impact the efficient execution of other concurrent transactions.

When the nature of an operation demands that it run at isolation level 3, you can lower its impact on concurrency by designing the query to read as few rows and index entries as possible. These steps will help the level 3 transaction run more quickly and, of possibly greater importance, will reduce the number of locks it places.

In particular, you may find that adding an index may greatly help speed up transactions, particularly when at least one of them must execute at isolation level 3. An index can have two benefits:

- Orders can be located in an efficient manner
- Fewer locks may be needed for searches which make use of the index.

You should design your database with the operations that you wish it to perform in mind. For example, if selections such as that used by the Accountant are to be run at level 3 frequently, you might find it worthwhile to index the order dates.

To execute each of your commands, the database server must decide which information in the database to access and an order in which to retrieve it. These plans are based both on general strategic principles as well as past experience with your database. the database server provides you with information about its plans. With this knowledge, you can more readily anticipate which rows and indexes the database server will need to read and hence which rows a particular statement is likely to lock.

☞ Further information on the details of the locking methods employed by Adaptive Server Anywhere is located in "How Adaptive Server Anywhere implements locking" on page 408.

☞ For information on performance and how Adaptive Server Anywhere plans its access of information to execute your commands, refer to "Monitoring and Improving Performance" on page 623.

Transactions for which no updates are lost

Depending upon what level of locking you demand from the database server, you can encounter other inconsistencies in addition to the three types of inconsistencies introduced above. This section introduces one additional type of inconsistency. It is of particular relevance to you if you make use of cursors within your SQL programs.

Some applications require that no updates be lost. The following example typifies the lost update problem.

Example

Consider a sequence of instructions which could occur when two people put money into the same account at about the same time.

The initial account balance is $1000, and two people (Alex and Ben, say) are about to deposit money into it. Alex will deposit $2000, while Ben will deposit $100.

1 Alex reads the account balance and finds it to be $1000.

2 Ben reads the account balance and finds it to be $1000.

3 Alex adds $2000 to the present balance of $1000 to calculate the new account balance. He then updates the account balance to reflect his deposit. He writes a new balance of $3000 into the database and completes his transaction.

4 Ben adds $100 to the present account balance, which he has read as $1000, to calculate the new account balance. He then updates the account balance to reflect his deposit. He writes a balance of $1100 into the database and completes his transaction.

5 The final balance recorded after the two deposits is $1100. The $2000 deposit entered by Alex has been lost.

While both transactions are perfectly correct in themselves, the interaction between the two creates an invalid result in the database, and Alex's update is lost.

Adaptive Server Anywhere provides you with several means of eliminating lost updates. You can execute such transactions at either isolation level 2 or 3, which guarantee repeatable reads. The next section describes another option.

Cursor stability

A cursor can hold the result of a SELECT

The database server allows you to return the results of a select in the form of a data type called a **cursor**. A cursor is similar to a table, but has the additional property that one row is identified as the present, or current row. Various commands allow you to navigate through the rows of a cursor. For example the FETCH command retrieves a row from the cursor and identifies it as the current row. You can step through all the rows in a cursor by repeatedly calling this command.

Cursors are of most use when you program procedures, or when you write applications which access a database using embedded SQL. They are not as useful when using Interactive SQL interactively.

The rows in a cursor, like those in a table, have no order associated with them. The FETCH command steps through the rows, but the order may appear random and can even be inconsistent. For this reason, you will frequently wish to add an index and impose an order by appending a ORDERED BY phrase to your SELECT statement.

For further information on programming SQL procedures and cursors, see "Using Procedures, Triggers, and Batches" on page 221.

The Adaptive Server Anywhere locking mechanism can achieve **cursor stability**. Cursor stability ensures that no other transactions can modify information which is contained in the present row of your cursor. The information in a row of a cursor may be the copy of information contained in a particular table or may be a combination of data from different rows of multiple tables. More than one table will likely be involved whenever you use a join or sub-selection within a SELECT statement.

The Adaptive Server Anywhere locking scheme assures cursor stability at isolation levels 1 through 3. Cursor stability also eliminates lost updates.

Cursor stability at isolation levels 1, 2, and 3

If you are writing SQL procedures or writing an application which makes use of embedded SQL, you may wish to take advantage of cursor stability. No row fetched through a cursor yields uncommitted data. No updates made through a cursor will be lost. Adaptive Server Anywhere automatically provides cursor stability at isolation levels 1, 2, and 3.

Early release of read locks—an exception

A transaction acquires a read lock on each row of a table which is read when you have set the isolation to level 3. Ordinarily, a transaction never releases a lock, once it has acquired it, before the end of the transaction. Indeed, it is essential that a transaction does not release locks early if the schedule is to be serializable.

Adaptive Server Anywhere always retains write locks until a transaction completes. If it were to release a lock sooner, another transaction could modify that row making it impossible to roll back the first transaction.

Read locks are never released either, except under one, special circumstance. Under isolation level 1, transactions acquire a read lock on a row only when it becomes the current row of a cursor. This lock eliminates lost updates by assuring that no other transaction can modify that row. When, however, that row is no longer current, the lock is released. This behavior is acceptable because the database engine does not need to guarantee repeatable reads at isolation level 1.

Typical transactions at various isolation levels

Various isolation levels lend themselves to particular types of tasks. Use the information below to help you decide which level is best suited to each particular operation.

Typical level 0 transactions

Transactions that involve browsing or performing data entry may last several minutes, and read a large number of rows. If isolation level 2 or 3 is used, concurrency can suffer. Isolation level of 0 or 1 is typically used for this kind of transaction.

For example, a decision support application that reads large amounts of information from the database to produce statistical summaries may not be significantly affected if it reads a few rows that are later modified. If high isolation is required for such an application, it may acquire read locks on large amounts of data, not allowing other applications write access to it.

Typical level 1 transactions

Isolation level 1 is particularly useful in conjunction with cursors, because this combination eliminates lost updates without greatly increasing locking requirements. Adaptive Server Anywhere achieves this benefit through the early release of read locks acquired for the present row of a cursor. These locks must persist until the end of the transaction at either levels two or three in order to guarantee repeatable reads.

For example, a transaction which updates inventory levels through a cursor is particularly suited to this level, because each of the adjustments to inventory levels as items are received and sold would not be lost, yet these frequent adjustments would have minimal impact on other transactions.

Typical level 2 transactions

At isolation level 2, rows which match your criterion cannot be changed by other transactions. You can thus employ this level when you must read rows more than once and rely that rows contained in your first result set won't change.

Because of the relatively large number of read locks required, you should use this isolation level with care. As with level 3 transactions, careful design of your database and indices reduce the number of locks acquired and hence can improve the performance of your database significantly.

Typical level 3 transactions

Isolation level 3 is appropriate for transactions which demand the most in security. The elimination of phantom rows lets you perform multi-step operations on a set of rows without fear that new rows will appear partway through your operations and corrupt the result.

However much integrity it provides, isolation level 3 demands respect when used on large systems which must support a large number of concurrent transactions. Adaptive Server Anywhere places more locks at this level than at any other, raising the likelihood that one transaction will impede the process of many others. You will thus likely wish to use lower isolation levels wherever possible.

Improving concurrency at isolation levels 2 and 3

When you must make use of serializable transactions, design your database, in particular the indices, with the business rules of your project in mind. You may also improve performance by breaking large transactions into several smaller ones, thus shortening the length of time that rows are locked.

Although serializable transactions have the most potential to block other transactions, they are not necessarily less efficient. When processing these transactions, Adaptive Server Anywhere can perform certain optimizations which may improve performance, despite the increased number of locks. For example, since all rows read must be locked whether or not they match the a search criteria, the database server is free to combine the operation of reading rows and placing locks.

How Adaptive Server Anywhere implements locking

Often, the general information about locking provided in the earlier sections will suffice to meet your needs. There are times, however, when you may benefit from more knowledge of what goes on inside Adaptive Server Anywhere when you perform basic types of operations. This knowledge will provide you with a better basis from which to understand and predict potential problems that users of your database may encounter.

The details of locking are best broken into two sections: what happens during a INSERT, UPDATE, DELETE or SELECT and how the various isolation levels affect the placement of read, phantom, and anti-phantom locks.

Although you can control the amount of locking that takes place within the database server by setting the isolation level, there is a good deal of locking that occurs at all levels, even at level 0. These locking operations are fundamental. For example, once one transaction updates a row, no other transaction must can modify the same row before the first transaction completes. Without this precaution, you could not role back the first transaction.

The locking operations which the engine performs at isolation level 0 are the best to learn first exactly because they represent the foundation. The other levels add locking features, but do not remove any present in the lower levels. Thus, moving to higher isolation level adds operations not present in lower levels and never removes any.

Locking during inserts

INSERT operations create new rows. Adaptive Server Anywhere employs the following procedure to ensure data integrity.

1. Make a location in memory to store the new row. The location is initially hidden from the rest of the database, so there is as yet no concern that another transaction could access it.
2. Fill the new row with any supplied values.
3. Write lock the new row.
4. Place an anti-phantom lock in the table to which the row is being added. Recall that anti-phantom locks are exclusive, so once the-anti-phantom lock is acquired, no other transaction can block the insertion by acquiring a phantom lock

5 Insert the row into the table. Other transactions can now, for the first time, see that the new row exists. They can't modify or delete it, though, because of the write lock acquired earlier.

6 Update all affected indexes and verify both referential integrity and uniqueness, where appropriate. Verifying referential integrity means ensuring that no foreign key points to a primary key which does not exist. Primary key values must be unique. Other columns may also be defined to contain only unique values, and if any such columns exist, uniqueness is verified.

7 The transaction can be committed provided referential integrity will not be violated by doing so: record the operation in the transaction log file and release all locks.

8 Insert other rows as required, if you have selected the cascade option, and fire triggers.

Uniqueness

You can ensure that all values in a particular column, or combination of columns, are unique. database server always performs this task by building an index for the unique column, even if you don't ask for an index explicitly.

In particular, all primary key values must be unique. database server automatically builds an index for the primary key of every table. Thus, you don't need to, and indeed shouldn't, explicitly ask database server to index a primary key as you risk asking it to create a redundant index.

Orphans and referential integrity

A foreign key is a reference to a primary key, usually in another table. When that primary key doesn't exist, the offending foreign key is called an **orphan**. Adaptive Server Anywhere automatically ensures that your database contains no orphans. This process is referred to as **verifying referential integrity**. The database server verifies referential integrity by counting orphans.

WAIT FOR COMMIT

You can ask the database server to delay verifying referential integrity to the end of your transaction. In this mode, you can insert one row which contains a foreign key, then insert a second row which contains the missing primary key. You must perform both operations in the same transaction. Otherwise, the database server will not allow your operations.

To request that the database server delay referential integrity checks until commit time, set the value of the option WAIT_FOR_COMMIT to ON. By default, this option is OFF. To turn it on, issue the following command:

```
SET OPTION WAIT_FOR_COMMIT = ON;
```

Before committing a transaction, the database server verifies that referential integrity is maintained by checking the number of orphans your transaction has created. At the end of every transaction, that number must be zero.

Even if the necessary primary key exists at the time you insert the row, the database server must ensure that it still exists when you commit your results. It does so by placing a read lock on the target row. With the read lock in place, any other transaction is still free to read that row, but none can delete or alter it.

Locking during updates

The database server modifies the information contained in a particular record using the following procedure.

1 Write lock the affected row.

2 If any entries changed are included in an index, delete each index entry corresponding to the old values. Make a record of any orphans created by doing so.

3 Update each of the affected values.

4 If indexed values were changed, add new index entries. Verify uniqueness where appropriate and verify referential integrity if a primary of foreign key was changed.

5 The transaction can be committed provided referential integrity will not be violated by doing so: record the operation in the transaction log file, including the previous values of all entries in the row, and release all locks.

6 Cascade the insert or delete operations, if you have selected this option and primary or secondary keys are affected.

You may be surprised to see that the deceptively simple operation of changing a value in a table can necessitate a rather large number of operations. The amount of work that the database server needs to do is much less if the value you are changing is not part of a primary or foreign key. It is lower still if it is not contained in an index, either explicitly or implicitly because you have declared that attribute unique.

The operation of verifying referential integrity during an UPDATE operation is no less simple than when the verification is performed during an INSERT. In fact, when you change the value of a primary key, you may create orphans. When you insert the replacement value, the database server must check for orphans once more.

Locking during deletes

The DELETE operation follows almost the same steps as the INSERT operation, except in the opposite order.

1. Write lock the affected row.
2. Delete each index entry present for the any values in the row. Immediately prior to deleting each index entry, acquire one or more phantom locks as necessary to prevent another transaction inserting a similar entry before the delete is committed. In order to verify referential integrity, the database server also keeps track of any orphans created as a side effect of the deletion.
3. Remove the row from the table so that it is no longer visible to other transactions. The row cannot be destroyed until the transaction is committed because doing so would remove the option of rolling back the transaction.
4. The transaction can be committed provided referential integrity will not be violated by doing so: record the operation in the transaction log file including the values of all entries in the row, release all locks, and destroy the row.
5. Cascade the delete operation, if you have selected this option and have modified a primary or foreign key.

Phantom locks

The database server must ensure that the DELETE operation can be rolled back. It does so in part by acquiring phantom locks. These locks are not exclusive, but deny other transactions the right to insert entries which make it impossible to roll back the DELETE operation. For example, the row deleted may have contained a primary key value for the table, or another unique value. Were another transaction allowed to insert a row with the same value, the DELETE could not be undone without violating the unique property of the primary key or attribute.

Adaptive Server Anywhere enforces uniqueness constraints through indexes. In the case of a simple table with only a one-attribute primary key, a single phantom lock may suffice. Other arrangements can quickly escalate the number of locks required. For example, the table may have no primary key or other index associated with any of the attributes. Since the rows in a table have no fundamental ordering, the only way of preventing inserts may be to phantom lock the entire table.

Deleting a row can mean acquiring a great many locks. You can minimize the effect on concurrency in your database in a number of ways. As described earlier, indexes and primary keys reduce the number of locks required because they impose an ordering on the rows in the table. the database server takes advantage of these orderings. Instead of acquiring locks on every row in the table, it can simply lock the *next* row. Without the index, the rows have no order and thus the concept of a next row is meaningless.

The database server acquires phantom locks on the row following the row deleted. Should you delete the last row of a table, then it simply places the phantom lock on an invisible end row. In fact, if the table contains no index, the number of phantom locks required is one more than the number of rows in the table.

Phantom locks and read locks

While one or more phantom lock excludes an anti-phantom lock, and one or more read lock excludes a write lock, no interaction exists between phantom/anti-phantom locks and read/write locks. For example, although a write lock is exclusive and so can not be acquired on a row which contains read locks, it can be acquired on a row which has only a phantom lock. More options are open to the database server because of this flexible arrangement, but it means that the engine must generally take the extra precaution of acquiring a read lock when acquiring a phantom lock. Otherwise, there another transaction could in effect remove the phantom lock by deleting the row on which it was acquired.

Selecting at isolation level 0

No locking operations are required when executing a SELECT statement at isolation level 0. Each transaction is not protected from changes introduced by other transactions. It is the responsibility of the programmer or database user to interpret the result of these queries with this limitation in mind.

Selecting at isolation level 1

You may be surprised to learn that Adaptive Server Anywhere uses almost no more locks when running a transaction at isolation level 1 than it does at isolation level 0. Indeed, the database server modifies its operation in only two ways.

The first difference in operation has nothing to do with acquiring locks, but rather with respecting them. At isolation level 0, a transaction is free to read any row, whether or not another transaction has acquired a write lock on it. By contrast, before reading each row an isolation level 1 transaction must check whether a write lock is in place. It cannot read past any write-locked rows because doing so might entail reading dirty data.

The second difference in operation creates cursor stability. Cursor stability is achieved by acquiring a read lock on the current row of a cursor. This read lock is released when the cursor is moved. More than one row may be affected if the contents of the cursor is the result of a join. In this case, the database server acquires read locks on all rows which have contributed information to the cursor's current row and removes all these locks as soon as another row of the cursor is selected as current. A read lock placed to ensure cursor stability is the only type of lock that does not persist until the end of a transaction.

Selecting at isolation level 2

At isolation level 2, Adaptive Server Anywhere modifies its procedures to ensure that your reads are repeatable. If your SELECT command returns values from every row in a table, then the database server acquires a read lock on each row of a table as it reads it. If, instead, your SELECT contains a WHERE clause, or other condition which restricts the rows to selected, then the database server instead reads each row, tests the values in the row against your criterion, and then acquires a read lock on the row if it meets your criterion.

As at all isolation levels, the locks acquired at level 2 include all those set at levels 1 and 0. Thus, cursor stability is again insured and dirty reads are not permitted.

Selecting at isolation level 3

When operating at isolation level 3, Adaptive Server Anywhere is obligated to ensure that all schedules are serializable. In particular, in addition to the requirements imposed at each of the lower levels, it must eliminate phantom rows.

To accommodate this requirement, the database server uses read locks and phantom locks. When you make a selection, the database server acquires a read lock on each row which contributes information to your result set. Doing so ensures that no other transactions can modify that material before you have finished using it.

This requirement is similar to the procedures that the database server engine uses at isolation level 2, but differs in that a lock must be acquired for each row read, *whether or not it meets any attached criterion*. For example, if you select the names of all employees in the sales department, then the engine must lock all the rows which contain information about a sales person, whether the transaction is executing at isolation level 2 or 3. At isolation level 3, however, it must also acquire read locks on each of the rows of employees which are *not* in the sales department. Otherwise, someone else accessing the database could potentially transfer another employee to the sales department while you were still using your results.

The fact that a read lock must be acquired on each row whether or not it meets your criterion has two important implications.

- The database server may need to place many more locks than would be necessary at isolation level 2.
- The database server can operate a little more efficiently: It can immediately acquire a read lock on each row at as it reads it, since the locks must be placed whether or not the information in the row is accepted.

The number of phantom locks the engine places can very greatly and depends upon your criterion and on the indexes available in the table. Suppose you select information about the employee with Employee ID 123. If the employee id is the primary key of the employee table, then the database server can economize its operations. It can use the index, which is automatically built for a primary key, to locate the row efficiently. In addition, there is no danger that another transaction could change another Employee's ID to 123 because primary key values must be unique. The engine can guarantee that no second employee is assigned that ID number simply by acquiring a read lock on only the one row containing information about the employee with that number.

By contrast, the database server would likely have to acquire many more locks were you to, instead, select all the employees in the sales department. Since any number of employees could be added to the department, the engine will likely have to read every row in the employee table and test whether each person is in sales. If this is the case, both read and phantom locks must be acquired for each row.

Special optimizations

The previous sections describe the locks acquired when all transactions are operating at a given isolation level. For example, when all transactions are running at isolation level 2, locking is performed as described in the appropriate section, above.

In practice, your database is likely to need to process multiple transactions which are at different levels. A few transactions, such as the transfer of money between accounts, must be serializable and so run at isolation level 3. For other operations, such as updating an address or calculating average daily sales, a lower isolation level will often suffice.

While the database server is not processing any transactions at level 3, it optimizes some operations so as to improve performance. In particular, many extra phantom and anti-phantom locks are often necessary to support a level 3 transaction. Under some circumstances, the database server can avoid either placing or checking for some types of locks when no level 3 transactions are present.

For example, the engine uses phantom locks to guard against two distinct types of circumstances:

1 Ensure that deletes in tables with unique attributes can be rolled back.

2 Eliminate phantom rows in level 3 transactions.

If no level 3 transactions are using a particular table, then the database server need not place phantom locks in the index of a table which contains no unique attributes. If, however, even one level 3 transaction is present, all transactions, even those at level 0, must place phantom locks so that the level 3 transactions can identify their operations.

Naturally, the database server always attaches notes to a table when it attempts the types of optimizations described above. Should a level 3 transaction suddenly start, you can be confident that the necessary locks will be put in place for it.

You may have little control over the mix of isolation levels in use at one time as so much will depend on the particular operations that the various users of your database wish to perform. Where possible, however, you may wish to select the time that level 3 operations execute because they have the potential to cause significant slowing of database operations. The impact is magnified because the database server is forced to perform extra operations for lower-level operations.

Locking conflicts

When a transaction attempts to acquire a lock on a row, but is forbidden by a lock held by another transaction, a locking conflict arises and the progress of the transaction attempting to acquire the lock is impeded or **blocked**.

☞ The next section, "The BLOCKING option" on page 416," describes transaction blocking.

☞ "Transaction blocking and deadlock" on page 416 describes deadlock, which occurs when two or more transactions are blocked by each other in such a way that none can proceed.

The BLOCKING option

If two simultaneously transactions have each acquired a read lock on a single row, the behavior when one of them attempts to modify that row depends on the database setting BLOCKING. To modify the row, that transaction must acquire a write lock, yet it cannot do so while the other transaction holds a lock on the row.

- If BLOCKING is ON (the default setting), then the transaction that attempts to write waits until the other transaction releases its read lock. At that time, the write goes through.
- If BLOCKING has been set to OFF, then the transaction that attempts to write receives an error.

When BLOCKING is set to OFF, the transaction terminates instead of waiting and any changes it has made are rolled back. In this event, try executing the transaction again, later.

Blocking is more *likely* to occur at higher isolation levels because more locking and more checking is done. Higher isolation levels usually provides less concurrency. How much less depends on the individual natures of the concurrent transactions.

☞ For information about the BLOCKING option, see "BLOCKING option" on page 145 of the book *Adaptive Server Anywhere Reference Manual*.

Transaction blocking and deadlock

Transaction blocking can lead to **deadlock**, where a set of transactions arrive at a state where none of them can proceed.

Reasons for deadlocks

A deadlock can arise for two reasons:

- **A cyclical blocking conflict** Transaction A is blocked on transaction B, and transaction B is blocked on transaction A. Clearly, more time will not solve the problem, and one of the transactions must be canceled, allowing the other to proceed. The same situation can arise with more than two transactions blocked in a cycle.
- **All active database threads are blocked** When a transaction becomes blocked, its database thread is not relinquished. If the database is configured with three threads and transactions A, B, and C are blocked on transaction D which is not currently executing a request, then a deadlock situation has arisen since there are no available threads.

Adaptive Server Anywhere automatically cancels the last transaction that became blocked (eliminating the deadlock situation), and returns an error to that transaction indicating which form of deadlock occurred.

Determining who is blocked

You can use the **sa_conn_info** system procedure to determine which connections are blocked on which other connections. This procedure returns a result set consisting of a row for each connection. One column of the result set lists whether the connection is blocked, and if so which other connection it is blocked on.

For more information, see "sa_conn_info system procedure" on page 753 of the book *Adaptive Server Anywhere Reference Manual*

Savepoints within transactions

Adaptive Server Anywhere supports **savepoints** within a transaction.

A SAVEPOINT statement defines an intermediate point during a transaction. You can undo all changes after that point using a ROLLBACK TO SAVEPOINT statement. Once a RELEASE SAVEPOINT statement has been executed or the transaction has ended, you can no longer use the savepoint.

No locks are released by the RELEASE SAVEPOINT or ROLLBACK TO SAVEPOINT commands: locks are released only at the end of a transaction.

Naming and nesting savepoints

Savepoints can be named and they can be nested. By using named, nested savepoints, you can have many active savepoints within a transaction. Changes between a SAVEPOINT and a RELEASE SAVEPOINT can still be canceled by rolling back to a previous savepoint or rolling back the transaction itself. Changes within a transaction are not a permanent part of the database until the transaction is committed. All savepoints are released when a transaction ends.

Savepoints make use of the rollback log. They cannot be used in bulk operations mode. There is very little additional overhead in using savepoints.

Particular concurrency issues

This section discusses the following particular concurrency issues:

Primary key generation

You will encounter situations where the database should automatically generate a unique number. For example, if you are building a table to store sales invoices you might prefer that the database assign unique invoice numbers automatically, rather than require sales staff pick them.

There are many methods for generating such numbers.

Example

For example, invoice numbers could be obtained by adding 1 to the previous invoice number. This method will not work when there is more than one person adding invoices to the database. Two people may decide to use the same invoice number.

There is more than one solution to the problem:

- Assign a range of invoice numbers to each person who adds new invoices.

  You could implement this scheme by creating a table with two columns *user name* and *invoice number*. The table would have one row for each user that adds invoices. Each time a user adds an invoice, the number in the table would be incremented and used for the new invoice. In order to handle all tables in the database, the table should have three columns: table name, user name, and last key value. You should periodically check that each person still has a sufficient supply of numbers.

- Create a table with two columns: *table name* and *last key value*.

  One row in this table would contain the last invoice number used. Each time someone adds an invoice, establish a new connection, increment the number in the table, and commit the change immediately. The incremented number can be used for the new invoice. Other users will be able to grab invoice numbers because you updated the row with a separate transaction that only lasted an instant.

- Probably the best solution is to use a column with a default value of AUTOINCREMENT.

  For example,

```
CREATE TABLE orders (
   order_id INTEGER NOT NULL DEFAULT AUTOINCREMENT,
   order_date DATE,
   primary key( order_id )
)
```

On INSERTs into the table, if a value is not specified for the autoincrement column, a unique value is generated. If a value is specified, it will be used. If the value is larger than the current maximum value for the column, that value will be used as a starting point for subsequent INSERTs. The value of the most recently inserted row in an autoincrement column is available as the global variable **@@identity**.

Unique values in replicated databases
Different techniques are required if you replicate your database and more than one person can add entries which must later be merged.
☞. See "Replication and concurrency" on page 422.

Autoincrement using PowerBuilder
Adaptive Server Anywhere supports an AUTOINCREMENT default value on fields. However, this type of field cannot be used by a PowerBuilder data window for primary key generation, because PowerBuilder is unable to find the record after insertion. If a table in your database is only added to from a PowerBuilder script, you may want to use the AUTOINCREMENT default value mechanism.

Data definition statements and concurrency

The CREATE INDEX statement, ALTER TABLE statement, and DROP statement are prevented whenever the statement affects a table that is currently is used by another connection. These statements can be time consuming and the database server will not process requests referencing the same table while the command is being processed.

The CREATE TABLE statement does not cause any concurrency conflicts.

The GRANT statement, REVOKE statement, and SET OPTION statement also do not cause concurrency conflicts. These commands affect any new SQL statements sent to the database engine, but do not affect existing outstanding statements.

GRANT and REVOKE for a user are not allowed if that user is connected to the database.

Data definition statements and replicated databases
Using data definition statements in replicated databases requires special care.
See the separate manual entitled *Data Replication with SQL Remote*.

Replication and concurrency

Some computers on your network might be portable computers that people take away from the office or which are occasionally connected to the network. There may be several database applications that they would like to use while not connected to the network.

Database replication is the ideal solution to this problem. Using SQL Remote, you can publish information in a consolidated, or master, database to any number of other computers. You can control precisely the information replicated on any particular computer. Any person can receive particular tables, or even portions of the rows or columns of a table. By customizing the information each receives, you can ensure that their copy of the database is no larger than necessary to contain the information they require.

☞ Extensive information on replication is provided in the separate manual entitled *Data Replication with SQL Remote*. The information in this section is, thus, not intended to be complete. Rather, it introduces concepts related directly to locking and concurrency considerations. For further, detail please refer to the supplementary manual.

SQL Remote allows replicated databases to be updated from a central, consolidated database, as well as updating this same central data as the results of transactions processed on the remote machine. Since updates can occur in either direction, this ability is referred to as **bi-directional replication**.

Since the results of transactions can affect the consolidated database, whether they are processed on the central machine or on a remote one, the effect is that of allowing concurrent transactions.

Transactions may happen at the same time on different machines. They may even involve the same data. In this case, though, the machines may not be physically connected. No means may exist by which the remote machine can contact the consolidated database to set any form of lock or identify which rows have changed. Thus, locks can not prevent inconsistencies as they do when all transactions are processed by a single engine.

An added complication is introduced by the fact that any given remote machine may not hold a full copy of the database. Consider a transaction executed directly on the main, consolidated database. It may affect rows in two or more tables. The same transaction might not execute on a remote database, as there is no guarantee that one or both of the affected tables is replicated on that machine. Even if the same tables exist, they may not contain exactly the same information, depending upon how recently the information in the two databases has been synchronized.

To accommodate the above constraints, replication is not based on transactions, but rather on operations. An **operation** is a change to one row in a table. This change could be the result of an UPDATE, INSERT, or DELETE statement. An operation resulting from an UPDATE or DELETE identifies the initial values of each column and a transaction resulting from an INSERT or UPDATE records the final values.

A transaction may result in none, one, or more than one operation. One operation will never result from two or more transactions. If two transaction modify a table, then two or more corresponding operations will result.

If an operation results from a transaction processed on a remote computer, then it must be passed to the consolidated database so that the information can be merged. If, on the other hand, an operation results from a transaction on the consolidated computer, then the operation may need to be sent to *some* remote sites, but not others. Since each remote site may contain a replica of a portion of the complete database, SQL Remote knows to pass the operation to a remote site only when it affects that portion of the database.

Transaction log based replication

SQL Remote uses a **transaction log based** replication mechanism. When you activate SQL Remote on a machine, it scans the transaction log to identify the operations it must transfer and prepares one or more messages.

SQL Remote can pass these messages between computers using a number of methods. It can create files containing the messages and store them in a designated directory. Alternatively, SQL Remote can pass messages using any of the most common messaging protocols. You likely can use your present e-mail system.

Conflicts may arise when merging operations from remote sites into the consolidated database. For example, two people, each at a different remote site, may have changed the same value in the same table. Whereas the locking facility built into Adaptive Server Anywhere can eliminate conflict between concurrent transactions handled by the same engine, it is impossible to automatically eliminate all conflicts between two remote users who both have permission to change the same value.

As the database administrator, you can avoid this potential problem through suitable database design or by writing conflict resolution algorithms. For example, you can decide that only one person will be responsible for updating a particular range of values in a particular table. If such a restriction is impractical, then you can instead use the conflict resolution facilities of SQL Remote to implement triggers and procedures which resolve conflicts in a manner appropriate to the data involved.

☞ SQL Remote provides the tools and programming facilities you need to take full advantage of database replication. For further information, see the manual *Data Replication with SQL Remote*.

Summary

Transactions and locking are perhaps second only in importance to relations between tables. While the integrity and performance of any database can benefit from the judicious use of locking and careful construction of transactions. Both are essential to creating databases which must execute commands from a large number of people or processes simultaneously.

Transactions group SQL statements into logical units of work. You may end each by either rolling back any changes you have made or by committing these changes and so making them permanent.

Transactions are essential to data recovery in the event of system failure. They also play a pivotal role in interweaving statements from concurrent transactions.

To improve performance, multiple transactions must be executed concurrently. Each transaction is composed of component SQL statements. When two or more transactions are to be executed concurrently, the database server must *schedule* the execution of the individual statements. Simultaneously transactions have the potential to introduce new, inconsistent results that could not arise were these same transactions executed sequentially.

Many types of inconsistencies are possible, but four typical types are particularly important because they are mentioned in the ISO SQL/92 standard and the isolation levels are defined in terms of them.

- **Dirty read** One transaction reads data modified, but not yet committed, by another.
- **Non-repeatable read** A transaction reads the same row twice and gets different values.
- **Phantom row** A transaction selects rows, using a certain criterion, twice and finds new rows in the second result set.
- **Lost Update** One transaction's changes to a row are completely lost because another transaction is allowed to save an update based on earlier data.

A schedule is called *serializable* whenever the effect of executing the statements according to the schedule is the same as could be achieved by executing each of the transactions sequentially. Schedules are said to be *correct* if they are serializable. A serializable schedule will cause none of the above inconsistencies.

Locking controls the amount and types of interference permitted. Adaptive Server Anywhere provides you with four levels of locking: isolation levels 0, 1, 2, and 3. At the highest isolation, level 3, Adaptive Server Anywhere guarantees that the schedule is serializable, meaning that the effect of executing all the transactions is equivalent to running them sequentially.

Unfortunately, locks acquired by one transaction may impede the progress of other transactions. Because of this problem, lower isolation levels are desirable whenever the inconsistencies they may allow are tolerable. Increased isolation to improve data consistency frequently means lowering the concurrency, the efficiency of the database at processing concurrent transactions. You must frequently balance the requirements for consistency against the need for performance to determine the best isolation level for each operation.

Conflicting locking requirements between different transactions may lead to blocking or deadlock. Adaptive Server Anywhere contains mechanisms for dealing with both these situations, and provides you with options to control them.

Transactions at higher isolation levels do not, however, *always* impact concurrency. Other transactions will be impeded only if they require access to locked rows. You can improve concurrency through careful design of your database and transactions. For example, you can shorten the time that locks are held by dividing one transaction into two shorter ones, or you might find that adding an index allows your transaction to operate at higher isolation levels with fewer locks.

The increased popularity of portable computers will frequently mean that your database may need to be replicated. Replication is an extremely convenient feature of Adaptive Server Anywhere, but it introduces new considerations related to concurrency. These topics are covered in a separate manual.