

# Ensuring Data Integrity

## About this chapter

This chapter describes the facilities in Adaptive Server Anywhere for ensuring that the data in your database is valid and reliable. Building integrity constraints right into the database is the surest way to make sure your data stays in good shape.

Several types of integrity constraints can be enforced. You can ensure individual entries are correct by imposing constraints and CHECK conditions on tables and columns. Setting column properties by choosing an appropriate data type or setting special default values assists this task.

The SQL statements in this chapter use the CREATE TABLE statement and ALTER TABLE statement, basic forms of which were introduced in "Working with Database Objects" on page 65.


## Contents

Topic	Page
Data integrity overview	348
Using column defaults	352
Using table and column constraints	356
Enforcing entity and referential integrity	360
Integrity rules in the system tables	365

## Data integrity overview

For data to have integrity means that the data is valid—correct and accurate—and that the relational structure of the database is intact. The relational structure of the database is enforced through referential integrity constraints. These are rules that maintain the consistency of data between tables.

Adaptive Server Anywhere supports stored procedures and JDBC, which allow you detailed control over how data gets entered into the database. You can also create triggers: custom procedures stored in the database that are invoked automatically when a certain action, such as an update of a particular column, is carried out.

 Procedures and triggers are discussed in "Using Procedures, Triggers, and Batches" on page 221.

### How data can become invalid

Here are a few examples of how the data in a database may become invalid if proper checks are not made. Each of these examples can be prevented by facilities described in this chapter.

#### Incorrect information

- ◆ A sales transaction takes place, but the operator entering the date of the transaction does so incorrectly
- ◆ A zero is missed off a salary entry, making an employee's salary ten times too small

#### Duplicated data

- ◆ A new department has been created, with **dept\_id** 200, and needs to be added to the department table of the organization's database—but two people enter this information into the table

#### Foreign key relations invalidated

- ◆ In a reorganization, the department identified by **dept\_id** 300 is closed down.

Each employee record for employees in this department is given a new **dept\_id** entry, and then the department 300 row is deleted from the department table. But one employee was missed, and still has **dept\_id** 300 in their record.

## Integrity constraints belong in the database

In order to ensure that the data in a database are valid, you need to formulate checks that define valid and invalid data and design rules to which data must adhere. The rules to which data must conform are often called **business rules**. The collective name for checks and rules is **constraints**.

### Build integrity constraints into database

Constraints built into the database itself are inherently more reliable than those built into client applications, or spelled out as instructions to database users. Constraints built into the database are part of the definition of the database itself and enforced consistently across all applications.

Setting a constraint once, in the database, imposes it for all subsequent interactions with the database, no matter from what source. In contrast, constraints built into client applications are vulnerable every time the software is altered, and may need to be imposed in several applications, or several places in a single client application.

## How database contents get changed

Information in database tables is changed by submitting SQL statements from client applications. Only a few SQL statements actually modify the information in a database.

- ◆ Information in a row of a table may be **updated**, using the UPDATE statement.
- ◆ An existing row of a table may be **deleted**, using the DELETE statement.
- ◆ A new row may be **inserted** into a table, using the INSERT statement.

## Data integrity tools

To assist in maintaining data integrity, you can use defaults, data constraints, and constraints that maintain the referential structure of the database.

### Defaults

You can assign default values to columns, to make certain kinds of data entry more reliable. For example:

- ◆ A column can have a current date default for recording the date of transactions with any user or client application action.
- ◆ A particular kind of default allows column values to be incremented automatically whenever a new row is entered. Items such as purchase orders, for example, can be guaranteed unique sequential numbers in this way without any user action.

	<p>☞ These and other column defaults are discussed in "Using column defaults" on page 352.</p>
Constraints	<p>You can use several types of constraints on the data in individual columns or tables. For example:</p> <ul style="list-style-type: none"><li>◆ A NOT NULL constraint prevents a column from containing a null entry.</li><li>◆ Columns can have CHECK conditions assigned to them, to ensure that a particular condition is met by every item in the column. You could ensure, for example, that salary column entries are within a specified range, protecting against user error when typing in new values.</li><li>◆ CHECK conditions can be made on the relative values in different columns, to ensure, for example, that in a library database a date_returned entry is later than a date_borrowed entry.</li><li>◆ More sophisticated CHECK conditions can be enforced using a trigger. Triggers are discussed in "Using Procedures, Triggers, and Batches" on page 221.</li></ul> <p>These and other table and column constraints are discussed in "Using table and column constraints" on page 356. Column constraints can be inherited from user-defined data types.</p>
Entity and referential integrity	<p>The information in relational database tables is tied together by the relations between tables. These relations are defined by the primary keys and foreign keys built in to the database design. The following integrity rules maintain the structure of the database:</p> <ul style="list-style-type: none"><li>◆ <b>Entity integrity</b> Keeps track of the primary keys. It guarantees that every row of a given table can be uniquely identified by a primary key that guarantees IS NOT NULL.</li><li>◆ <b>Referential integrity</b> Keeps track of the foreign keys that define the relationships between tables. It guarantees that all foreign key values either match a value in the corresponding primary key or contain the NULL value if they are defined to allow NULL.</li></ul> <p>☞ For more information about enforcing referential integrity, see "Enforcing entity and referential integrity" on page 360. For more information about designing appropriate primary and foreign key relations, see "Designing Your Database" on page 323.</p>
Triggers for advanced integrity rules	<p>You can also use triggers to maintain data integrity. A <b>trigger</b> is a procedure stored in the database that is executed automatically whenever the information in a specified table is altered. Triggers are a powerful mechanism for database administrators and developers to ensure that data is kept reliable.</p>

☞ For a full description of triggers, see "Using Procedures, Triggers, and Batches" on page 221.

## SQL statements for implementing integrity constraints

The following SQL statements are used to implement integrity constraints:

- ◆ **CREATE TABLE statement** This statement is used to implement integrity constraints as the database is being created.
- ◆ **ALTER TABLE statement** This statement is used to add integrity constraints to an existing database, or to modify constraints for an existing database.
- ◆ **CREATE TRIGGER statement** This statement is used to create triggers to enforce more complex business rules.

☞ For full descriptions of the syntax of these statements, see "SQL Statements" on page 339 of the book *Adaptive Server Anywhere Reference Manual*.

## Using column defaults

Column defaults automatically assign a specified value to particular columns whenever a new row is entered into a database table, without any action on the part of the client application, as long as no value is specified by the client application. If the client application does specify a value for the column, it overrides the column default value.

Column defaults are useful for automatically filling columns with information, such as the date or time a row is inserted, or the user ID of the person entering the information.

Using column defaults encourages data integrity, but does not enforce it. Defaults can always be overridden by client applications.

### Supported default values

The following default values are supported:

- ◆ A string specified in the CREATE TABLE statement or ALTER TABLE statement
- ◆ A number specified in the CREATE TABLE statement or ALTER TABLE statement
- ◆ An automatically incremented number: one more than the previous highest value in the column
- ◆ The current date, time, or timestamp
- ◆ The current user ID of the database user
- ◆ A NULL value
- ◆ A constant expression, as long as it does not reference database objects.

## Creating column defaults

Column defaults can be created at the time a table is created, using the CREATE TABLE statement, or added at a later time using the ALTER TABLE statement.

### Example

The following statement adds a condition to an existing column named **id** in the **sales\_order** table, so that it is automatically incremented (unless a value is specified by a client application):

```
ALTER TABLE sales_order
MODIFY id DEFAULT AUTOINCREMENT
```

Each of the other default values is specified in a similar manner. For a detailed description of the syntax, see "CREATE TABLE statement" on page 415 of the book *Adaptive Server Anywhere Reference Manual*.

## Modifying and deleting column defaults

Column defaults can be changed or removed by using the same form of the ALTER TABLE statement as used to create defaults. The following statement changes the default value of a column named **order\_date** from its current setting to CURRENT DATE:

```
ALTER TABLE sales_order
MODIFY order_date DEFAULT CURRENT DATE
```

Column defaults are removed by modifying them to be NULL. The following statement removes the default from the **order\_date** column:


```
ALTER TABLE sales_order
MODIFY order_date DEFAULT NULL
```

## Working with column defaults in Sybase Central

All adding, altering, and deleting of column defaults in Sybase Central is carried out in the Type tab of the column properties sheet.

### ❖ To display the property sheet for a column:

- 1 Connect to the database.
- 2 Click the Tables folder for that database, and click the table holding the column you want to change.
- 3 Double-click the Columns folder to open it, and double-click the column to display its property sheet.

 For more information, see the Sybase Central online Help.

## Current date and time defaults

For columns with the DATE, TIME, or TIMESTAMP data type, the current date, current time, or current timestamp may be used as a default. The default specified must be compatible with the column's data type.

### Useful examples of current date default

The following are just a few examples of when a current date default would be useful:

- ◆ To record dates of phone calls in a contact database
- ◆ To record the dates of orders in a sales entry database
- ◆ To record the date a book is borrowed in a library database

### Current timestamp

The current timestamp is used for similar purposes to the current date default, but when greater accuracy is required. For example, a user of a contact management application may have several contacts with a single customer in one day: the current timestamp default would be useful to distinguish these contacts.

The current timestamp is also useful when the sequence of events is important in a database, as it records a date and the time down to a precision of millionths of a second.

☞ For more information about timestamps, times, and dates, see "SQL Data Types" on page 219 of the book *Adaptive Server Anywhere Reference Manual*.

## The user ID default

Assigning a DEFAULT USER to a column is an easy and reliable way of identifying the person making an entry in a database. This information may be required, for example, when salespeople are working on commission.

Building a user ID default into the primary key of a table is a useful technique for occasionally connected users. These users can make a copy of tables relevant to their work on a portable computer, make changes while not connected to a multiuser database, and then apply the transaction log to the server when they return. Incorporating their user ID into the primary key of the table helps to prevent conflicts during the update.

## The autoincrement default

The autoincrement default is useful for numeric data fields. It assigns each new row a value one greater than that of the previous highest value in the column. Autoincrement columns can be used to record purchase order numbers, to identify customer service calls, or other entries where an identifying number is required, but the value of the number itself has no meaning.

Autoincrement columns are typically primary key columns or columns constrained to hold unique values (see "Enforcing entity integrity" on page 360). It is highly recommended that the autoincrement default not be used in cases other than these, as doing so can adversely affect the database performance.

One case when an autoincrement default does not adversely affect performance is when the column is the first column of an index. This is because the server uses an index or key definition to find the highest value.



The next value to be used for each column is stored as a long integer (4 bytes). Using values greater than  $(2^{31} - 1)$ , that is, large double or numeric values, may cause wraparound to negative values, and AUTOINCREMENT should not be used in such cases.

☞ A column with the AUTOINCREMENT default is referred to in Transact-SQL applications as an IDENTITY column. For information on IDENTITY columns, see "The special IDENTITY column" on page 799.

## The NULL default

For columns that allow NULL values, specifying a NULL default is exactly the same as not specifying a default at all. A NULL value is assigned to the column if no value is explicitly assigned by the client when inserting the row.

NULL defaults are typically used when information for some columns is optional or not always available and is not required for the data in the database to be correct.

☞ For more information on the NULL value, see "NULL value" on page 213 of the book *Adaptive Server Anywhere Reference Manual*.

## String and number defaults

A specific string or number can be specified as a default value, as long as the column holds a string or number data type. You must ensure that the default specified can be converted to the column's data type.

Default strings and numbers are useful when there is a typical entry for a given column. For example, if an organization has two offices: the headquarters in **city\_1** and a small office in **city\_2**, you may want to set a default entry for a location column to **city\_1**, to make data entry easier.

## Constant expression defaults

A constant expression can be used as a default value, as long as it does not reference database objects. This allows column defaults to contain entries such as *the date fifteen days from today*, which would be entered as

```
... DEFAULT ( dateadd( day, 15, getdate() ) )
```

## Using table and column constraints

The CREATE TABLE statement and ALTER TABLE statement can specify many different attributes for a table. Along with the basic table structure (number, name and data type of columns, name and location of the table), you can specify other features that allow control over data integrity.

### Caution

*Altering tables can interfere with other users of the database. Although the ALTER TABLE statement can be executed while other connections are active, it is prevented if any other connection is using the table to be altered. For large tables, ALTER TABLE is a time-consuming operation, and no other requests referencing the table being altered are allowed while the statement is being processed.*

This section describes how to use constraints to help ensure that the data entered in the table is correct.

## Using CHECK conditions on columns

A CHECK condition can be applied to values in a single column, to ensure that they satisfy rules. These rules may be rules that data must satisfy in order to be reasonable, or they may be more rigid rules that reflect organization policies and procedures.

You use a CHECK condition to ensure that the values in a column satisfy some definite criterion.

CHECK conditions on individual column values are useful when only a restricted range of values are valid for that column. Here are some examples:

### Example 1

- ◆ You can enforce a particular formatting requirement. If a table has a column for phone numbers you may wish to ensure that they are all entered in the same manner. For North American phone numbers, you could use a constraint such as the following:

```
ALTER TABLE customer
MODIFY phone
CHECK ( phone LIKE '(____) ____-____' )
```

### Example 2

- ◆ You can ensure that the entry matches one of a limited number of values. For example, to ensure that a **city** column only contains one of a certain number of allowed cities (say, those cities where the organization has offices), you could use a constraint like the following:

```
ALTER TABLE office
MODIFY city
```

```
CHECK ( city IN ( 'city_1', 'city_2', 'city_3' ) )
```

- ◆ By default, string comparisons are case insensitive unless the database is explicitly created as a case-sensitive database.

### Example 3

- ◆ You can ensure that a date or number falls in a particular range. For example, you may want to require that the **start\_date** column of an employee table must be between the date the organization was formed and the current date. This could be achieved as follows:

```
ALTER TABLE employee
MODIFY start_date
CHECK ( start_date BETWEEN '1983/06/27'
      AND CURRENT DATE )
```

- ◆ You can use several date formats: the YYYY/MM/DD format used in this example has the virtue of always being recognized regardless of the current option settings.

Column CHECK tests only fail if the condition returns a value of FALSE. If a value of UNKNOWN is returned, the change is allowed.

#### Column CHECK conditions in previous releases

There is a change in the way that column CHECK conditions are held in this release. In previous releases, column CHECK conditions were merged together with all other CHECK conditions on a table into a single CHECK condition. Consequently, they could not be individually replaced or deleted. In this release, column CHECK conditions are held individually in the system tables, and can be replaced or deleted individually. Column CHECK conditions added before this release are still held in a single table constraint, even if the database is upgraded.

## Column CHECK conditions from user-defined data types

You can attach CHECK conditions to user-defined data types. Columns defined on those data types inherit the CHECK conditions. A CHECK condition explicitly specified for the column overrides that from the user-defined data type.

When defining a CHECK condition on a user-defined data type, any variable prefixed with the @ sign is replaced by the name of the column when the CHECK condition is evaluated. For example, the following user-defined data type accepts only positive integers:

```
CREATE DATATYPE posint INT
CHECK ( @col > 0 )
```

Any variable name prefixed with @ could be used instead of @col. Any column defined using the **posint** data type accepts only positive integers unless the column itself has a CHECK condition explicitly specified.

An ALTER TABLE statement with the DELETE CHECK clause deletes all CHECK conditions from the table definition, including those inherited from user-defined data types.

🔗 For information on user-defined data types, see "User-defined data types" on page 241 of the book *Adaptive Server Anywhere Reference Manual*.

## Working with column constraints in Sybase Central

All adding, altering, and deleting of column constraints in Sybase Central is carried out in the Constraints tab of the column properties sheet.

### ❖ To display the property sheet for a column:

- 1 Connect to the database.
- 2 Click the Tables folder for that database, and click the table holding the column you wish to change.
- 3 Double-click the Columns folder to open it, and double-click the column to display its property sheet.

🔗 For more information, see the Sybase Central online Help.

## Using CHECK conditions on tables

A CHECK condition can be applied as a constraint on the table, instead of on a single column. Such CHECK conditions typically ensure that two values in a row being entered or modified have a proper relation to each other. Column CHECK conditions are held individually in the system tables, and can be replaced or deleted individually. This is more flexible behavior, and CHECK conditions on individual columns are recommended where possible.

For example, in a library database, the **date\_returned** column for a particular entry must be later than (or the same as) the **date\_borrowed** entry:

```
ALTER TABLE loan
ADD CHECK(date_returned >= date_borrowed)
```

## Modifying and deleting CHECK conditions

There are several ways to alter the existing set of CHECK conditions on a table.

- ◆ You can add a new CHECK condition to the table or to an individual column, as described above.
- ◆ You can delete a CHECK condition on a column by setting it to NULL. The following statement removes the CHECK condition on the **phone** column in the **customer** table:

```
ALTER TABLE customer
MODIFY phone CHECK NULL
```

- ◆ You can replace a CHECK condition on a column in the same way as adding a CHECK condition. The following statement adds or replaces a CHECK condition on the **phone** column of the **customer** table:

```
ALTER TABLE customer
MODIFY phone
CHECK ( phone LIKE '___-___-___' )
```

- ◆ There are two ways of modifying a CHECK condition defined on the table, as opposed to a CHECK condition defined on a column:
  - ◆ You can add a new CHECK condition using ALTER TABLE with an ADD table-constraint clause.
  - ◆ You can delete all existing CHECK conditions, including column CHECK conditions, using ALTER TABLE DELETE CHECK, and then add in new CHECK conditions.

All CHECK conditions on a table, including CHECK conditions on all its columns and CHECK conditions inherited from user-defined data types, are removed using the ALTER TABLE statement with the DELETE CHECK clause, as follows:

```
ALTER TABLE table_name
DELETE CHECK
```

Deleting a column from a table does not delete CHECK conditions associated with the column that are held in the table constraint. If the constraints are not removed, any attempt to query data in the table will produce a column not found error message.

Table CHECK conditions fail only if a value of FALSE is returned. If a value of UNKNOWN is returned, the change is allowed.

## Enforcing entity and referential integrity

The relational structure of the database enables information within the database to be identified by the personal server, and ensures that relationships between tables, described in the database structure, are properly upheld by all the rows in each table.

### Enforcing entity integrity

When a row is inserted or is updated, the database server ensures that the primary key for the table is still valid: that each row in the table is uniquely identified by the primary key.

#### Example 1

The **employee** table in the sample database uses an employee ID as the primary key. When a new employee is added to the table, the database server checks that the new employee ID value is unique and is not NULL.

#### Example 2

The **sales\_order\_items** table in the sample database uses two columns to define a primary key.

This table holds information about items ordered. One column contains an **id** specifying an order, but there may be several items on each order, so this column by itself cannot be a primary key. An additional **line\_id** column identifies which line corresponds to the item. The columns **id** and **line\_id**, taken together, specify an item uniquely, and form the primary key.

### If a client application breaches entity integrity

Entity integrity requires that each value of a primary key be unique within the table, and that there are no NULL values. If a client application attempts to insert or update a primary key value, and provides values that are not unique, entity integrity would be breached.

If an attempt to breach entity integrity is detected, the new information is not added to the database. Instead, the client application receives an error.


It is up to the application programmer to decide how to present this information to the user and enable the user to take appropriate action. The appropriate action is usually just to provide a different, unique, value for the primary key.

## Primary keys enforce entity integrity

Once the primary key for each table is specified, no further action is needed by client application developers or by the database administrator to maintain entity integrity.

The primary key for a table is defined by the table owner when the table is created. If the structure of a table is modified at a later date, the primary key may also be redefined.

Some application development systems and database design tools allow you to create and alter database tables. If you are using such a system, you may not have to enter the CREATE TABLE or ALTER TABLE command explicitly; the application generates the statement itself from the information you provide.

 For information on creating primary keys, see "Creating primary and foreign keys" on page 74. For the detailed syntax of the CREATE TABLE statement, see "CREATE TABLE statement" on page 415 of the book *Adaptive Server Anywhere Reference Manual*. For information about changing table structure, see the "ALTER TABLE statement" on page 351 of the book *Adaptive Server Anywhere Reference Manual*.

## Enforcing referential integrity

A foreign key relates the information in one table (the **foreign** table) to information in another (**referenced** or **primary**) table. A particular column, or combination of columns, in a foreign table is designated as a foreign key to the primary table.

For the foreign key relationship to be valid, the entries in the foreign key must correspond to the primary key values of a row in the referenced table. Occasionally, some other unique column combination may be referenced, instead of a primary key.

### Example 1

The sample database contains an employee table and a department table. The primary key for the employee table is the employee ID, and the primary key for the department table is the department ID.

One of the items of information about each employee is the department ID of the department to which they belong. In the employee table, the department ID is called a **foreign key** for the department table; each department ID in the employee table corresponds exactly to a department ID in the department table.

The foreign key relationship is a many-to-one relationship. Several entries in the employee table have the same department ID entry, but the department ID is the primary key for the department table, and so is unique. If a foreign key were able to reference a column in the department table containing duplicate entries, there would be no way of knowing which of the rows in the department table is the appropriate reference.


#### Example 2

Suppose the database also contained an office table, listing office locations. The employee table might have a foreign key for the office table that indicates where the employee's office is located. The database designer may wish to allow for an office location not being assigned at the time the employee is hired. In this case, the foreign key is **optional** and should allow the NULL value to indicate that it is optional when the office location is unknown or when the employee does not work out of an office. A foreign key that is not optional is called **mandatory**.

## Foreign keys enforce referential integrity

Like primary keys, foreign keys are created using the CREATE TABLE statement or ALTER TABLE statement.

Once a foreign key has been created, the column or columns in the key can contain only values that are present as primary key values in the table associated with the foreign key.

 For information on creating foreign keys, see "Creating primary and foreign keys" on page 74.

## Losing referential integrity

Referential integrity can be lost in the following ways:

- ◆ If a primary key value is updated or deleted, all those foreign keys referencing it would be left in an invalid state.
- ◆ If a new row is added to the foreign table, and a value is entered for the foreign key that has no corresponding primary key value, the database would be left in an invalid state.

Adaptive Server Anywhere provides protection against both types of integrity loss.



## If a client application breaches referential integrity

If a client application updates or deletes a primary key value in a table, and if that primary key value is referenced by a foreign key elsewhere in the database, there is a danger of a breach of referential integrity.

### Example

If the server allowed the primary key to be updated or deleted, and made no alteration to the foreign keys that referenced it, the foreign key reference would be invalid. Any attempt to use the foreign key reference, for example in a SELECT statement using a KEY JOIN clause, would fail, as no corresponding value in the referenced table would exist.

While breaches of entity integrity are generally straightforward for Adaptive Server Anywhere to handle, simply by refusing to enter the data and returning an error message, potential breaches of referential integrity are more complicated.

There are several options available to ensure that referential integrity is maintained. These options are called **referential integrity actions**.

## Referential integrity actions

The simplest way to maintain referential integrity when a referenced primary key is updated or deleted is to disallow the update or delete.

Often it is also possible to take an action on each foreign key to maintain referential integrity. The CREATE TABLE and ALTER TABLE statements allow database administrators and table owners to specify what action should be taken on foreign keys that reference a modified primary key.

Each of the available referential integrity actions may be specified separately for updates and deletes of the primary key:

- ◆ **RESTRICT** Generate an error if an attempt is made to modify a referenced primary key value, and do not carry out the modification. This is the default referential integrity action.
- ◆ **SET NULL** Set all foreign keys that reference the modified primary key to NULL.
- ◆ **SET DEFAULT** Set all foreign keys that reference the modified primary key to the default value for that column (as specified in the table definition).
- ◆ **CASCADE** When used with ON UPDATE, update all foreign keys that reference the updated primary key to the new value. When used with ON DELETE, delete all rows containing foreign keys that reference the deleted primary key.

Referential integrity actions are implemented using system triggers. The trigger is defined on the primary table, and is executed using the permissions of the owner of the primary table.

## Referential integrity checking

Using a database option to control check time

For foreign keys defined to RESTRICT operations that would violate referential integrity, checks are carried out by default at the time a statement is executed. If you specify a CHECK ON COMMIT clause, then the checks are carried out only when the transaction is committed.

The setting of the WAIT\_FOR\_COMMIT database option controls the behavior when a foreign key is defined to restrict operations that would violate referential integrity. This option is overridden by the CHECK ON COMMIT clause.

With the default, **wait\_for\_commit** set to OFF, an operation that would leave the database inconsistent is not allowed to execute. For example, a DELETE operation of a department that has employees in it is not allowed. The statement:

```
DELETE FROM department
WHERE dept_id = 200
```

gives the error primary key for row in table 'department' is referenced in another table.

If **wait\_for\_commit** is set to ON, referential integrity is not checked until a commit is executed. If the database is in an inconsistent state, the commit is not allowed and an error is reported. In this mode, a department with employees could be deleted. However, the change could not be committed to the database until one of the following actions is taken:

- ◆ The employees belonging to that department are also deleted or reassigned.
- ◆ This search condition can also be used on a SELECT statement to select the rows that violate referential integrity.
- ◆ The **dept\_id** row is inserted back into the **department** table.
- ◆ The transaction is rolled back to undo the DELETE operation.

## Integrity rules in the system tables

All the information about integrity checks and rules in a database is held in the following system tables:

System table	Description
<b>SYS.SYSTABLE</b>	CHECK constraints are held in the <b>view_def</b> column of SYS.SYSTABLE. For views, the <b>view_def</b> holds the CREATE VIEW command that created the view. You can check whether a particular table is a base table or a view by looking at the <b>table_type</b> column, which is BASE or VIEW.
<b>SYS.SYSTRIGGER</b>	Referential integrity actions are held in SYS.SYSTRIGGER. The <b>referential_action</b> column holds a single character indicating whether the action is cascade (C), delete (D), set null (N), or restrict (R). The <b>event</b> column holds a single character specifying the event that causes the action to occur: a delete (D), insert (I), update (U), or update of column-list (C). The <b>trigger_time</b> column shows whether the action occurs after (A) or before (B) the triggering event.
<b>SYS.SYSFOREIGNKEYS</b>	This view presents the foreign key information from the two tables SYS.SYSFOREIGNKEY and SYS.SYSFKCOL in a more readable format.
<b>SYS.SYSCOLUMNS</b>	This view presents the information from the SYS.SYSCOLUMN table in a more readable format. It includes default settings and primary key information for columns.

🔗 For a description of the contents of each system table, see "System Tables" on page 771 of the book *Adaptive Server Anywhere Reference Manual*. You can use Sybase Central or Interactive SQL to browse these tables and views.

