3022_12_CMP1 22/12/03 3:21 pm Page 341

12

Programmatic Data Protection

In this chapter, you'll examine the predicates and domains that could not be implemented using simple check or default constraints. To implement these types of checks you have three different mechanisms at your disposal.

- **Triggers**: These differ from constraints in that they are pieces of code that you attach to a table (or tables). This code is automatically run whenever the event(s) specified occur in the table(s). They are extremely flexible and can access multiple columns, multiple rows, multiple tables, and even multiple databases. As a simple case, let's consider a situation in which you want to ensure that an update of a value is performed on both the tables where it occurs. You can write a trigger that will disallow the update unless it occurs in both tables.

  This is the second best place to protect the data. You do have to code every trigger rule in T-SQL, but the user cannot get around this by any error in an external program.

- **Stored procedures**: Stored procedures are pieces of precompiled T-SQL stored in the database, dealing with tables in a very flexible manner, so that different business rules can be applied to the same table under different circumstances. A simple example of a stored procedure is one that returns all the data held in a given table. In a more complex scenario, they can be used to grant different permissions to different users regarding manipulation of tables. This isn't all that great a solution to protect the data because you have to code the rules into any procedures that access the data. However, because it's in the central location, you can use this to implement rules that may be different based on various situations.

- **Client executable code**: Useful in dealing with situations in which business rules are optional or are flexible in nature. A common example is asking the user "Are you sure you wish to delete this row?" SQL Server is a server product, and if you ask it to delete data, it deletes data. Most server products work in this manner, leaving the client programs the responsibility of implementing flexible business rules and warnings. Bear in mind that applications come and go, but the data must always be protected.

3022_12_CMP1 22/12/03 3:21 pm Page 342

In moving down this list the solutions become less desirable, yet each one has specific benefits that are appropriate in certain situations.

Your test tables are the same tables as the ones you used in the previous chapter, and all of the modifications you had previously made are intact.

Triggers

Triggers are a type of stored procedure attached to a table or view and are executed only when the contents of a table are changed. They can be used to enforce almost any business rule, and are especially important for dealing with situations that are too complex for a check constraint to handle.

Triggers need to be used when one of the following needs to be done:

- Perform cross-database referential integrity
- Check inter-row rules, where just looking at the current row isn't enough for the constraints
- Check inter-table constraints, when rules require access to data in a different table
- Introduce desired side effects to your data modification queries

The main advantage that triggers have over constraints is the ability to directly access other tables, and to execute multiple rows at once. In a trigger you can run almost every T-SQL command, except for the following:

`ALTER DATABASE`	`CREATE DATABASE`	`DROP DATABASE`
`RESTORE LOG`	`RECONFIGURE`	`RESTORE DATABASE`

There are two different models of triggers that you'll look at using the following:

- **AFTER**: Meaning the trigger fires after the command has affected the table, though not on views. AFTER triggers are used for building enhanced business rule handlers and you would generally put any kind of logging mechanisms into them. You may have an unlimited number of AFTER triggers that fire on `INSERT`, `UPDATE`, and `DELETE` or any combination of them. Even if you have an INSTEAD-OF trigger, you may still have as many AFTER triggers as wanted, because they can all be combined into a single trigger.
- **INSTEAD OF**: Meaning that the trigger operates instead of the command (`INSERT`, `UPDATE`, or `DELETE`) affecting the table or view. In this way, you can do whatever you want with the data, either modifying it as sent, or putting it into another place. You can have only a single `INSERT`, `UPDATE`, and `DELETE` trigger of this type per table. You can however combine all three into one and have a single trigger that fires for all three operations.

3022_12_CMP1 22/12/03 3:21 pm Page 343

The following section will describe some common techniques for employing triggers in your data protection code. For a deeper explanation on how to code triggers, see Appendix E.

Using AFTER Triggers to Solve Common Problems

In this section you'll look at how you use triggers to solve typical problems, such as the following:

- Cascading inserts
- Range checks
- Cascading deletes setting the values of child tables to NULLs

It's important to consider that whatever you're doing with AFTER triggers depends on preexisting data in your table passing all constraint requirements. For instance, it wouldn't be proper to insert rows in a child table (thereby causing its entire trigger/constraint chain to fire) when the parent's data hasn't been validated. Equally you wouldn't want to check the status of all the rows in your table until you've completed all of your changes to them; the same could be said for cascading delete operations. The three examples that follow are but a small subset of all the possible uses for triggers; they are just a sample to get things rolling. Each of the snippets we'll present in the next three subsections will fit into a trigger (of any type) which will be of the following basic format:

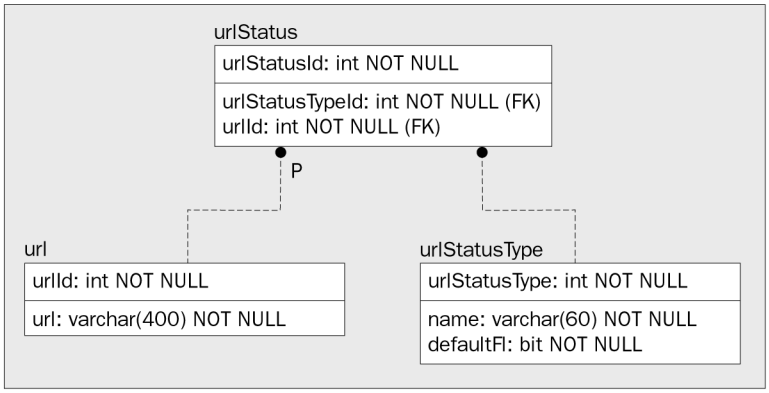
```
CREATE TRIGGER <triggerName>
ON <tableName>
FOR <action> AS
--------------------------------------------------------------------------------
-- Purpose : Trigger on the <action> that fires for any <action> DML
--------------------------------------------------------------------------------
BEGIN
   DECLARE @rowsAffected int,     --stores the number of rows affected
   @errNumber int,                --used to hold the error number after DML
   @msg varchar(255),             --used to hold the error message
   @errSeverity varchar(20),
   @triggerName sysname,

   SET @rowsAffected = @@rowcount
   SET @triggerName = object_name(@@procid)      --used for messages

   --no need to continue on if no rows affected
   IF @rowsAffected = 0 return

<insert snippets here >

END
```

You'll generally write your triggers so that when an error occurs, you'll raise an error, and roll back the transaction to halt any further commands. The <insert snippets here> section will be replaced by chunks of code that do some check or some action as outlined in the following sections covering some uses of AFTER triggers.

3022_12_CMP1 22/12/03 3:21 pm Page 344

Cascading Inserts

By cascading inserts, you refer to the following situation: After a row is inserted into a table, one or more other new rows are automatically inserted into other tables. This is especially important when you're creating mandatory one-to-one or one-to-many relationships. For example, say you have the following set of tables:



In this case the `url` table defines a set of URLs for your system, despite the fact that the relationship between `url` and `urlStatus` is a one-to-many. You begin by building a trigger that inserts a row into the `urlStatus` table on an insert that creates a new row with the `urlId` and the default `urlStatusType` based on `defaultFl` having the value of 1. (We'll assume for now that there is a maximum of one row with a `defaultFl` equal to 1, and will implement a check to make sure this is so.)

```
--add a row to the urlStatus table to tell it that the new row
--should start out as the default status
INSERT INTO urlType (urlId, urlTypeId)
SELECT INSERTED.urlId, urlType.urlTypeId
FROM INSERTED
   CROSS JOIN urlType      --use cross join with a where clause
                             --as this isn't technically a join between
                             --INSERTED and urlType
WHERE urlType.defaultFl = 1

SET @errorNumber = @@error
IF @errorNumber <> 0
BEGIN
   SET @msg = 'Error: ' + CAST(@errorNumber as varchar(10)) +
                           ' occurred during the creation of urlStatus'
   RAISERROR 50000 @msg
   ROLLBACK TRANSACTION
   RETURN
END
```

3022_12_CMP1 22/12/03 3:21 pm Page 345

A trigger might also be needed to disallow the deleting of the default `urlType`, because you would always want to have a default `urlStatus` value. However, creation of this trigger will leave a situation where you want to do the following:

- Delete a `url`, but you cannot because of `urlType`s that exist
- Delete the `urlType`s, but you cannot because it's the last one for a `url`

This is a difficult situation to handle using conventional triggers and constraints, and may well be easier to solve by adding a column to the table informing the triggers (or constraints) that the row is available for deletion. This is a general problem in the case of tables in which you want to implement a two-way relationship, but also want to enforce a one-way relationship under certain circumstances.

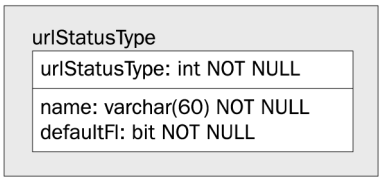
Range Checks

A range check means simply ensuring that a given range of data conforms to the data validation rules. Some examples are as follows:

- Balancing accounts to make sure that there isn't a negative balance left by the transaction
- Ensuring that a proper number of rows exist for a relationship (cardinality), such as a "one-to-between-five-and-seven" relationship
- Making sure that in a table (or group of rows in the table) there exists only a single row that is the primary or default row

For our example, we make sure that there is no more than a single default flag set to `true` or `1` in the table; the others must be `0` or `false`. You'll also throw in, for good measure, code that will take any new value where the `defaultFl` column is `1` and set all of the others to `0`. If the user manually sets more than one `defaultFl` column to `1`, then a check is made afterwards to cause the operation to fail.

We'll use as our example table the `urlStatusType` table you built earlier, as shown here:



3022_12_CMP1 22/12/03 3:21 pm Page 346

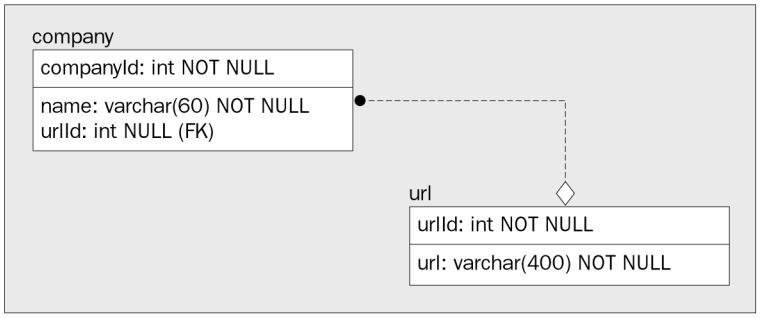
and you'll implement out range checking criteria as follows:

```
--if the defaultFl column was modified
IF UPDATE(defaultFl)
BEGIN
   --update any other rows in the status type table to not default if
   --a row was inserted that was set to default
   UPDATE urlStatusType
   SET defaultFl = 0
   FROM urlStatusType
          --only rows that were already default
   WHERE urlStatusType.defaultFl = 1
          --and not in the inserted rows
     AND urlStatusTypeId NOT IN
     ( SELECT urlStatusTypeId
     FROM inserted
     WHERE defaultFl = 1
     )

   SET @errorNumber = @@error
   IF @errorNumber <> 0
   BEGIN
      SET @msg = 'Error: ' + CAST(@errorNumber as varchar(10)) +
                  ' occurred during the modification of defaultFl'
      RAISERROR 50000 @msg
      ROLLBACK TRANSACTION
      RETURN
   END

   --see if there is more than 1 row set to default
   --like if the user updated more than one in a single operation
   IF ( SELECT count(*)
        FROM urlStatusType
        WHERE urlStatusType.defaultFl = 1 ) > 1
   BEGIN
      SET @msg = 'Too many rows with default flag = 1'
      RAISERROR 50000 @msg
      ROLLBACK TRANSACTION
      RETURN
   END
END
```

Cascading Deletes Setting the Values of Child Tables to NULL

When you have tables with optional alternate keys, instead of cascading on a parent delete, you may sometimes wish to remove the link and set any foreign key references to `NULL`. To do this in a `DELETE` trigger, you simply update every row in the child table that refers to the value in the row(s) you're deleting. Take the following relationship for example:

3022_12_CMP1 22/12/03 3:21 pm Page 347



The `url` table contains all of the Internet addresses for your system, and the company uses one of the values for a company URL. If you want to delete one or more of the `url` table's rows, you don't want to delete all the companies that use the URL. So you might implement this by setting the child rows to `NULL` instead of deleting the rows themselves. Because SQL Server only implements a cascading delete where the child row is deleted, you'll have to use a trigger. The trigger code that does this is as follows:

```
UPDATE company
SET urlId = NULL
FROM DELETED
   JOIN company
   ON DELETED.urlId = company.urlId

SET @errorNumber = @@error
IF @errorNumber <> 0
BEGIN
   SET @msg = 'Error: ' + CAST(@errorNumber as varchar(10)) +
                          ' occurred during delete cascade set NULL'
   RAISERROR 50000 @msg
   ROLLBACK TRANSACTION
   RETURN
END
```

Uses of INSTEAD OF Triggers

`INSTEAD OF` triggers can be used to automatically set or modify values in your statements. `INSTEAD OF` triggers fire prior to both constraints and AFTER triggers, so you should know that they can be used to modify data en route to the table. You'll consider three examples of how they can be used.

Automatically Maintaining Columns

As an example, you're going to build two triggers, an `INSTEAD OF INSERT` trigger and an `INSTEAD OF UPDATE` trigger, which will automatically set the first letter of the names of your artists to uppercase, instead of the current all lowercase format. You'll make use of a function called `string$properCase`. This function is created in Appendix C during the exercise on creating scalar functions (see the Apress web site). It's also located in the code downloads. You don't need the exact implementation here. It's simply a function that takes a name and cleans up the format as you'll see in the following example:

3022_12_CMP1 22/12/03 3:21 pm Page 348


```
CREATE TRIGGER artist$insteadOfInsert ON artist
INSTEAD OF INSERT
AS
INSERT INTO artist(name, defaultFl, catalogNumberMask)
SELECT dbo.string$properCase(name), defaultfl, catalogNumberMask
FROM INSERTED
GO

CREATE TRIGGER artist$insteadOfUpdate on artist
INSTEAD OF UPDATE
AS
UPDATE artist
SET name = string$properCase(INSERTED.name),
           defaultFl = INSERTED.defaultFl,
           catalogNumberMask = INSERTED.catalogNumberMask
FROM artist
JOIN INSERTED ON artist.artistId = INSERTED.artistId
GO
```

To test your `INSERT` trigger, you execute the following:

```
-- insert fairly obviously improperly formatted name
INSERT INTO artist (name, defaultFl, catalogNumberMask)
VALUES ('eLvIs CoStElLo',0,'77______') -- then retrieve the last inserted
value into this table
SELECT artistId, name
FROM artist
WHERE artistId = ident_current('artist')
```

which returns the required response, as shown here:

```
artistId                    name
------------------------------------------------------------
19                          Elvis Costello
```

Next you'll test the `UPDATE` trigger. First, you check all of the values in the table as they stand right now:

```
SELECT artistId, name
FROM artist
```

This returns a list of untidily formatted rows:

```
artistId                    name
------------------------------------------------------------
19                          Elvis Costello
1                           THE BEATLES
15                          The Monkees
2                           THE WHO
```

3022_12_CMP1 22/12/03 3:21 pm Page 349

Then you run the following query, which looks as if it will set the names to all uppercase:

```
UPDATE artist
SET name = UPPER(name)
```

However, you see that all of the rows are not in uppercase, though they are now formatted in a tidier manner.

```
artistId                    name
------------------------------------------------------------
19                          Elvis Costello
1                           The Beatles
15                          The Monkees
2                           The Who
```

`INSTEAD OF` triggers are the best place to do this sort of data manipulation, because it saves you from inserting bad data, then having to take an additional step in an `AFTER` trigger to update it. Any time you need to extend the way that an `INSERT`, `UPDATE`, or `DELETE` operation is implemented in a generic way, `INSTEAD OF` triggers are great.

Conditional Insert

During our demonstration of error handling, a batch insert was created that added albums to a table based on the `catalogNumber`, though it didn't match the mask we had set up for an artist. Here, you'll build a more streamlined example, where the table (or tables) in such a system will accept any data from the user, while placing any invalid data in an exception-handling table so that someone can later fix any problems. First you'll have to drop the original constraint that you added in the previous chapter:

```
ALTER TABLE album
DROP CONSTRAINT chkAlbum$catalogNumber$function$artist$catalogNumberValidate
```

At this point you're unprotected from any invalid values inserted into the `catalogNumber` column. However, you'll build an `INSTEAD OF` trigger that will take all valid rows and insert (or update) them in the `album` table. If they are invalid, you'll insert them in a new table that you'll create called `albumException`, which will have the same structure as the `album` table, with a different primary key value and a new column called `operation` (for insert or update).

```
CREATE TABLE albumException
(
   albumExceptionId int NOT NULL IDENTITY,
   name varchar(60) NOT NULL,
   artistId int NOT NULL,
   catalogNumber char(12) NOT NULL,
   exceptionAction char(1),
   exceptionDate datetime
)
```

3022_12_CMP1 22/12/03 3:21 pm Page 350

Now you create a simple `INSTEAD OF` trigger to intercept the user's data and attempt to validate the catalog number. If the data is valid, you update the table, otherwise it will be added to the exception table.

```
CREATE TRIGGER album$insteadOfUpdate
ON album
INSTEAD OF UPDATE
AS

DECLARE @errorValue int -- this is the variable for capturing error status

UPDATE album
SET name = INSERTED.name, artistId = INSERTED.artistId,
           catalogNumber = INSERTED.catalogNumber
FROM inserted
JOIN album
   ON INSERTED.albumId = album.albumId
   -- only update rows where the criteria is met
   WHERE dbo.album$catalogNumbervalidate(INSERTED.catalogNumber,
          INSERTED.artistId) = 1

-- check to make certain that an error did not occur in this statement
SET @errorValue = @@error
IF @errorValue <> 0
BEGIN
   RAISERROR 50000 'Error inserting valid album rows'
   ROLLBACK TRANSACTION
   RETURN
END

-- get all of the rows where the criteria isn't met
INSERT INTO albumException (name, artistId, catalogNumber, exceptionAction,
                            exceptionDate)
SELECT name, artistId, catalogNumber, 'U',getdate()
FROM INSERTED
WHERE NOT(      -- generally the easiest way to do this is to copy
                -- the criteria and do a not(where ...)
          dbo.album$catalogNumbervalidate(INSERTED.catalogNumber,
          INSERTED.artistId) = 1 )

SET @errorValue = @@error
IF @errorValue <> 0
BEGIN
   RAISERROR 50000 'Error logging exception album rows'
   ROLLBACK TRANSACTION
   RETURN
END
GO
```

3022_12_CMP1 22/12/03 3:21 pm Page 351

Now, updating a row to a proper value can simply happen with the following statements:

```
-- update the album table with a known good match to the catalog number
UPDATE album
SET catalogNumber = '222-22222-22'
WHERE name = 'the white album'

-- then list the artistId and catalogNumber of the album in the "real" table
SELECT artistId, catalogNumber
FROM album
WHERE name = 'the white album'

-- as well as the exception table
SELECT artistId, catalogNumber, exceptionAction, exceptionDate
FROM albumException
WHERE name = 'the white album'
```

The catalog number matches what you've updated it to and there are no exceptions in the `albumException` table:

```
ArtistId        catalogNumber
-------------------------------------------------
1               222-22222-22

artistId        catalogNumber exceptionAction exceptionDate
-----------------------------------------------------------------------------------------
```

Then you do an obviously invalid update:

```
UPDATE album
SET catalogNumber = '1'
WHERE name = 'the white album'

-- then list the artistId and catalogNumber of the album in the "real" table
SELECT artistId, catalogNumber
FROM album
WHERE name = 'the white album'

-- as well as the exception table
SELECT artistId, catalogNumber, exceptionAction, exceptionDate
FROM albumException
WHERE name = 'the white album'
```

3022_12_CMP1 22/12/03 3:21 pm Page 352

You see that you haven't updated the row–no error was returned, but you've now added a row to your exception table, thereby alerting you to the error:

```
artistId            catalogNumber
------------------------------------------------------
1                   222-22222-22

artistId            catalogNumber       exceptionAction     exceptionDate
-------------------------------------------------------------------------------------------------------------
1                   1                   U                   2001-01-07 01:02:59.363
```

It's left to the reader to create the `INSTEAD OF INSERT` trigger to accompany the `UPDATE` case. You would also probably want to extend the exception table to include some sort of reason for the failure if you had more than one possibility.

Modifying the Data Represented in a View

In general, doing inserts, updates, and deletes on views has always had its drawbacks, because the following criteria must be met to execute a modification statement against a view:

- `UPDATE` and `INSERT` statements may only modify a view if they only reference the columns of one table at a time.
- `DELETE` statements can only be used if the view only references a single table.

However, by using `INSTEAD OF` triggers, you can implement the `INSERT`, `UPDATE`, and `DELETE` mechanisms on views. In this example, you'll create an extremely simple view and an insert trigger to allow inserts:

```
-- create view, excluding the defaultFl column, which you don't want to let
-- users see in this view, and you want to view the names in upper case.
CREATE VIEW vArtistExcludeDefault
AS
SELECT artistId, UPPER(name) AS name, catalogNumberMask
FROM artist
GO

-- then you create a very simple INSTEAD OF insert trigger

CREATE TRIGGER vArtistExcludeDefault$insteadOfInsert
ON vArtistExcludeDefault
INSTEAD OF INSERT
AS
BEGIN
   --note that you don't use the artistId from the INSERTED table
   INSERT INTO artist (name, catalogNumberMask, defaultFl)
   SELECT NAME, catalogNumberMask, 0 --only can set defaultFl to 0
                                     --using the view
   FROM INSERTED
END
GO
```

3022_12_CMP1 22/12/03 3:21 pm Page 353

Then you simply insert using the view just as you would the table (excluding the identity column, which you cannot set a value for), as shown here:

```
INSERT INTO vArtistExcludeDefault (name, catalogNumberMask)
VALUES ('The Monkees','44______')
```

However, the view has other ideas, as follows:

```
Server: Msg 233, Level 16, State 2, Line 1
The column 'artistId' in table 'vArtistExcludeDefault' cannot be NULL.
```

This isn't what you expected or wanted. So you have to reformulate your insert to include the `artistId` and an invalid value. Now the insert works just fine, as shown:

```
INSERT INTO vArtistExcludeDefault (artistId, name, catalogNumberMask)
VALUES (-1, 'The Monkees','44______')

SELECT * FROM vArtistExcludeDefault
WHERE artistId = ident_current('vArtistExcludeDefault')
```

which gives you back the value that you hoped it would:

```
artistId    name            defaultFl    catalogNumberMask
----------------------------------------------------------------------------------------------------
15          THE MONKEES     0            44_______
```

It should be noted that if you had two or more tables in the view, and you execute an insert on the view, you could insert data into *all* of the tables that make up the view. This neatly sidesteps the requirement that `INSERT` and `UPDATE` statements on views can touch only single tables at a time, which is very cumbersome. (Further details on views can be found in the next chapter.)

Client Code and Stored Procedures

For quite a while, there has been a programmatic drive to move much of the business rule implementation and data protection code out of SQL Server and into a middle-tier set of interface objects. In this way the database, client and business rules exist in three units that can be implemented independently. Thus business rules that you may well have thought about implementing via constraints and triggers get moved out of this "data" layer and into client-based code, such as a COM object and stored procedures.

Such a multitier design also attempts to make the life of the user easier, because users edit data using custom front-end tools, although the middle-tier services maintain and protect data that passes through them, thereby insulating the users from all the required SQL code. Not only that, but these services can also directly handle any errors that occur and present the user with meaningful error messages. Because application users primarily edit rows one at a time, rather than a large number of rows, this actually works really great.

3022_12_CMP1 22/12/03 3:21 pm Page 354

The other point is that, in most enterprise applications (for instance situations with hundreds of thousands of users on a website), the database is usually considered as the "system bottleneck." Though it's possible to distribute the load on a single server, in many cases it can be much easier to spread the load across many application servers as you learned in Chapter 9.

However, almost any data protection mechanism that is enforced without the use of constraints or triggers may prove problematic. Let's consider your list of possible users that you introduced at the very beginning of the chapter, namely the following:

- Users using custom front-end tools: When users all use the custom front-end tools that are developed for the interface, there is no problem with employing the middle tier. In fact, it can have some great benefits, because as discussed, the object methods used to enforce the rules can be tuned for maximum performance.
- Users using generic data tools such as Microsoft Access**:** Let's consider a case in which a user needs to modify a set of "live" data, but only needs it for a week, or a weekend, and there is no time to write a full-blown application. You won't be able to let them directly access the data because it's in a raw unprotected state. Hence, you'll either have to code a relevant business rule into the Access database, or deny the user and make them wait until an application is created. This type of thing is relatively rare, and you can usually stop this kind of activity with strict policies against such access.
- Data import routines that acquire data from external sources: Almost every system of any magnitude will include some import facilities to take data in a raw format from external systems, maybe from another part of your company or another company altogether, and place this data in a table on the database server. This can be in as simple a form as a user application to import a spreadsheet, or as complex as an interface between all of the schools in a state and the respective state department of education. The tools will range from user applications, DTS, or even BCP (bulk copy program that comes with SQL Server). When the middle tier owns all of the business rules and data-integrity checks, you'll either have to go in through the middle tier (frequently one at a time) or extract all of the business rules and code them into your import routines.
- Raw queries executed by data administrators to fix problems caused by user error: Almost anybody with administration experience has had to remove a few rows from a database that users have erroneously created but cannot remove, and in so doing may have mistakenly deleted the wrong rows, for example, active account rows rather than inactive ones. In this situation, if you had business rules built into a trigger that allowed the deletion of inactive accounts only, an error message would have been returned to the user warning that active accounts could not be deleted. Obviously you cannot protect against a really bad action, such as systematically deleting every row in a table, but when a fully featured database is implemented and the data protected using constraints and triggers, it's next to impossible to make even small mistakes in data integrity.

3022_12_CMP1 22/12/03 3:21 pm Page 355

As the data architect, I very much desire the possibilities offered by multitier development. However, the load of business rule and data-integrity rule implementation should be "shared" between the middle tier and database server as appropriate. Two very specific types of such rules that are far better implemented in the database are as follows:

- Any rule that can be placed in a NULL, foreign key, or check constraint: This is due to the fact that, when building additional external interfaces, this kind of check will generally make up quite a bit of the coding effort. Furthermore, base data integrity should be guaranteed at the lowest level possible, which will allow as many programs as possible to code to the database. As a final point, these constraints will be used by the optimizer to make queries run faster.
- Rules that require inter-table validations: Whenever you save a value, you must check to see if a value exists in a different table. The additional table will have to be accessed automatically to make certain that the state that was expected still exists. In some cases the middle tier will try to cache the data on the database to use in validation, but there is no way to spread this cached data to multiple servers in a manner that ensures that the value you're entering has proper data integrity.

Having considered some of the possible drawbacks to middle-tier coding, let's now look at cases where a stored procedure or user code is the optimal place to locate the business rules.

Mutable Rules

For a given set of criteria, on one occasion when they are met, the rule evaluates to true and an action is carried out, whereas on another occasion, the rule evaluates to false and a different action is carried out. The best litmus test for such a rule is to see whether the user is able to bypass it. For instance, in the chapter example, you could attempt to implement the following rule: "The user should enter a valid artist for the album."

As a reminder, you have the album table that contains a `NOT NULL` foreign-key relationship to the artist table. Hence, you're forced to put in a valid `artistId` when you modify the table. From here you have two courses of action.

- Make the `artistId` in the album table nullable and hence optional, thereby allowing the user to select an album without supplying an `artistId`. To follow the rule "should," the front end would then likely open a dialog box asking the user, "Are you sure you don't want to assign an artist to this album?"
- Alternatively, you could rephrase the business rule as "the user could enter an invalid artist." This would mean that the database could allow any value from the `artistId` and indeed let the client application handle the validation by checking the value to see if it's correct and then send a dialog box stating the following: "Are you sure you don't want to assign an artist to this album?" or worse still: "You have entered an invalid `artistId`, you should enter a valid one." You would then have to drop the database validations in case the user says, "Nah, let me enter the invalid value."

3022_12_CMP1 22/12/03 3:21 pm Page 356

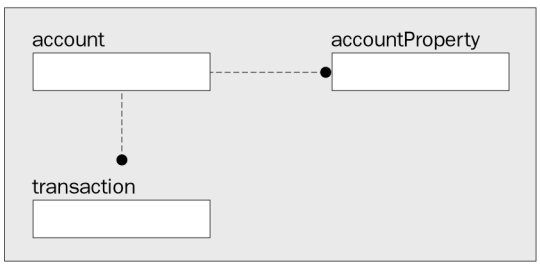
The point I'm trying to make here is that SQL Server cannot converse with the user in an interactive manner. The hard and fast trigger and constraint validations still depend largely on the process of submitting a request and waiting to see if it completes successfully, and you need a more interactive method of communication in which you can influence events after the request has been submitted, and before the result comes back.

In the following examples, you'll look at situations concerning rules that cannot be realistically implemented via triggers and constraints.

> *Admittedly where there's a will there's a way. It's possible, using temporary tables as messaging mechanisms, to "pass" values to a trigger or constraint. In this manner, you can optionally override functionality. However, triggers and constraints aren't generally considered suitable places to implement mutable business rules.*

Optionally Cascading Deletes

Cascading deletes are a great resource, but you should use them with care. As discussed, they automatically remove child rows that are dependent on the content of the deleted parent row, and you would rarely want to warn the user of the existence of these child rows before they are deleted. However this wouldn't be ideal in the case of a bank account. Let's say that you have the following tables:



In this case, it will make sense to automatically cascade deletes of an `accountProperty` row if you want to delete a row from the `account` table. However, if an account has entries in the `transaction` table, you need to ensure that the user is aware of these, and thus warn them if a cascading delete is requested. This will of course increase complexity, because you won't be able to delete an account as well as its properties and transactions in a single statement.

Instead of a single statement, you'll have to execute the following steps:

- Run a `SELECT` statement for each child table that you'll want to optionally cascade delete to, in order to show the user what exists.
- If the user accepts, execute a `DELETE` statement for each child table that had rows related to your primary table, and in the same atomic operation, delete the row you're interested in.

3022_12_CMP1 22/12/03 3:21 pm Page 357

This code could be built into a single stored procedure that checks for the children, and if they exist, returns a resultset of rows for the user to see what needs to be deleted. It will also include a parameter to allow the user to ignore the existence of rows and go ahead and delete them. The following shows you an example of how you might handle this situation (note that for clarity I've removed transactions and error handling, which will be covered in greater depth in the next chapter):

```
CREATE PROCEDURE account$delete
(
    @accountId int,
    @removeChildTransactionsFl bit = 0
) as

-- if they asked to delete them, just delete them
IF @removeChildTransactionsFl = 1
   DELETE [transaction] --table named with keyword
   WHERE  accountId = @accountId
ELSE --check for existence
  BEGIN
      IF EXISTS (SELECT *
                 FROM  [transaction]
                 WHERE  accountId = @accountId)
        BEGIN
             RAISERROR 50000 'Child transactions exist'
             RETURN -100
        END
  END

DELETE account
WHERE  accountID = @accountId
```

Now the user could try to execute the stored procedure with the flag equal to `0`, and if any children existed, the user would get notification. If not, the account would be deleted, and presumably the properties would be removed as well via a cascading relationship.

Rules That Are Based Upon How a Person Feels

Sadly for the system architect, this is a reality that must be faced. In a typical system, you'll have many hard and fast business rules that cannot be broken. However a large number of mutable rules appear as hard and fast rules. As an example, consider the following statement:

> *It's the company's policy that you don't ship the product out to a customer until you have payment.*

This doesn't seem at all problematic until you get a request for a shipment to take place as a goodwill gesture.

3022_12_CMP1 22/12/03 3:21 pm Page 358

As a further example consider the following rule:

Products will be shipped out in the same sequence as the orders were received.

Consider the case of a quality-control system that deals with product Y. Let's say that a constraint is applied such that a given customer specification must be met, otherwise the product cannot be shipped to the customer. So the user calls up and requests a large shipment, and the clerk who handles these orders calls up the user interface to choose some of product Y to ship. However there aren't enough products to fulfill the order. When the clerk contacts the customer to inform him of this problem, the customer requests a closely related product rather than waiting for more product Y to become available. What the clerk now discovers is that because the business rules stand, she is unable to process this request, and so has to send it to the ordering manager to fulfill.

A stored procedure approach to this problem might allow users of a different security group to override the given settings where appropriate. In this way the clerk can get a list of product Y to ship, choose materials, add them to an order, and press a button that starts shipping the product, thereby calling a stored procedure `productOrder$ship`. This procedure will check that the product is within the customer specifications, and then set the order to be shipped. If the values are out of specification, it would deny the user. However, the ordering manager will be able to execute a different stored procedure, say `productOrder$shipOutOfSpec`, that would allow the user to ship an out-of-specification product.

The following are two things of note in this scenario:

- The front-end would be responsible for knowing that the product was out of specification and wouldn't allow the user to execute the `productOrder$shipOutOfSpec` procedure, even though knowing this might be based on querying the rights on each stored procedure (using the `PERMISSIONS` function).
- `productOrder$ship` and `productOrder$shipOutOfSpec` would probably end up calling a different stored procedure that would be used to encapsulate the actual time to start shipping the order. However, by presenting actual object and method names that are different, you can limit any situational coding in your procedures (if this type of user, do this, else do this) and present a secure interface based just on the stored procedures that are used for the enforcement.

The reality for a data architect is that there will always be those who work "outside the rules" for both proper and improper reasons, and you must take this into account whenever you build a truly user-friendly system.

Case Study

Continuing with the data protection code, you'll now build a few of the necessary triggers, and you'll finally learn a few optional rules that you'll need to implement your system.

3022_12_CMP1 22/12/03 3:21 pm Page 359

Remove the Time from Transaction Dates Trigger

The first trigger you'll create is an `INSTEAD OF` trigger to format the transaction date. There is no reason to store the time for the transaction date, and so you start out with your template trigger code, and add code to remove the time element from the dates using the `date$removeTime` function used previously for a check constraint, as shown here:

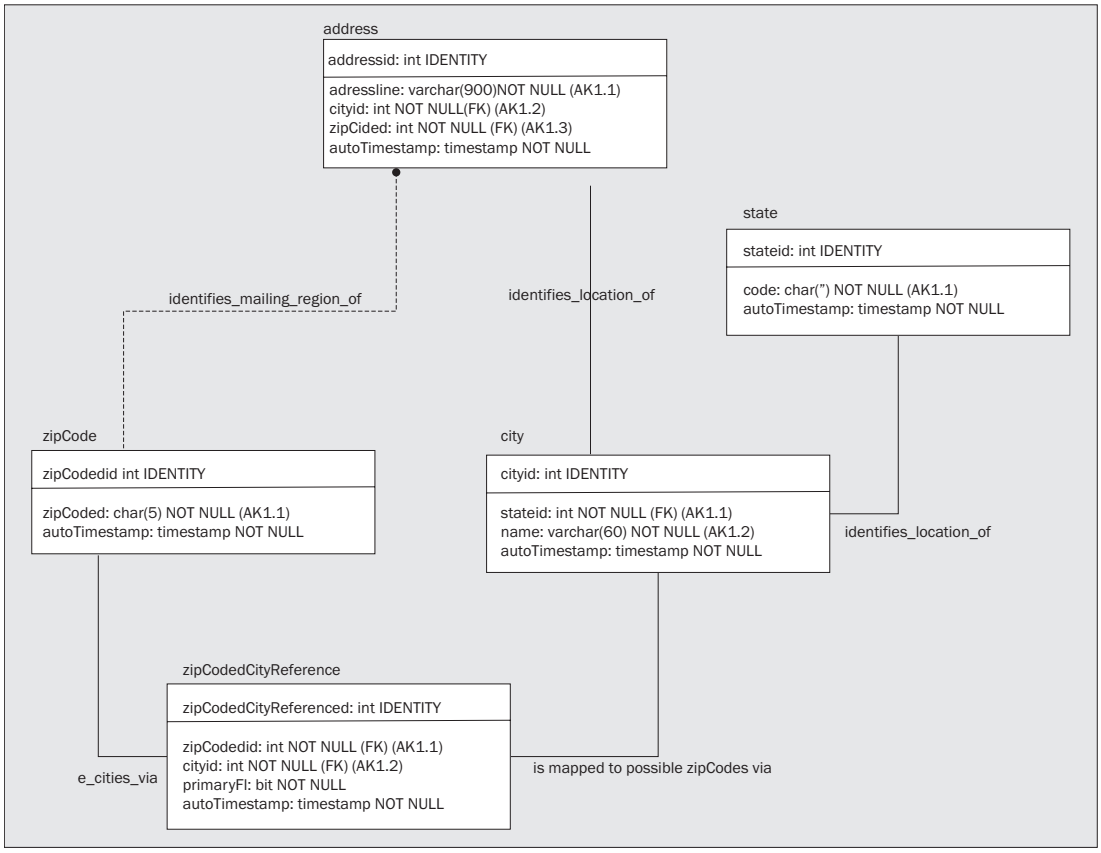
```
CREATE TRIGGER transaction$insteadOfInsert
ON [transaction]
INSTEAD OF INSERT
AS
--------------------------------------------------------------------------------
-- Purpose : Trigger on the insert that fires for any insert DML
-- : * formats the date column without time
--------------------------------------------------------------------------------
BEGIN
   DECLARE @rowsAffected int,     -- stores the number of rows affected
   @errorNumber int,                 -- used to hold the error number after
DML
   @msg varchar(255),              -- used to hold the error message
   @errSeverity varchar(20),
   @triggerName sysname

   SET @rowsAffected = @@rowcount
   SET @triggerName = object_name(@@procid) --used for messages

   --no need to continue on if no rows affected
   IF @rowsAffected = 0 RETURN

   --perform the insert that you're building the trigger instead of
   INSERT INTO [transaction] (accountId, number, date, description,
                              amount, signature, payeeId,
                              transactionTypeId)
   SELECT accountId, number, dbo.date$removeTime(date) AS date,
          description, amount, signature, payeeId, transactionTypeId
   FROM INSERTED

   SET @errorNumber = @@error
   IF @errorNumber <> 0
   begin
      SET @msg = 'Error: ' + CAST(@errorNumber AS varchar(10)) +
                          ' occurred during the insert of the rows into ' +
                          ' the transaction table.'
      RAISERROR 50000 @msg
      ROLLBACK TRANSACTION
      RETURN
    END
END
```

Ninety-five percent of the update trigger is the same code, so this is left to the reader.

3022_12_CMP1 22/12/03 3:21 pm Page 360

Validate City and Zip Codes Match Trigger

In your `address` table, you have the `cityId` and `zipCodeId` columns, which are foreign keys to the `zipCode` and `city` tables, respectively. You have also included a `zipCodeCityReference` table that is used to help the user select city and state from the zip code. You would fill this table from a commercially available zip code database (like from the US Postal Service). A final purpose of this table is to validate the `city` and `zipCodeId` values. As a reminder, here was the physical data model for these tables:



You can do this by making sure that the `cityId` and `zipCodeId` pair that has been entered is found in the reference table, as follows:

3022_12_CMP1 22/12/03 3:21 pm Page 361

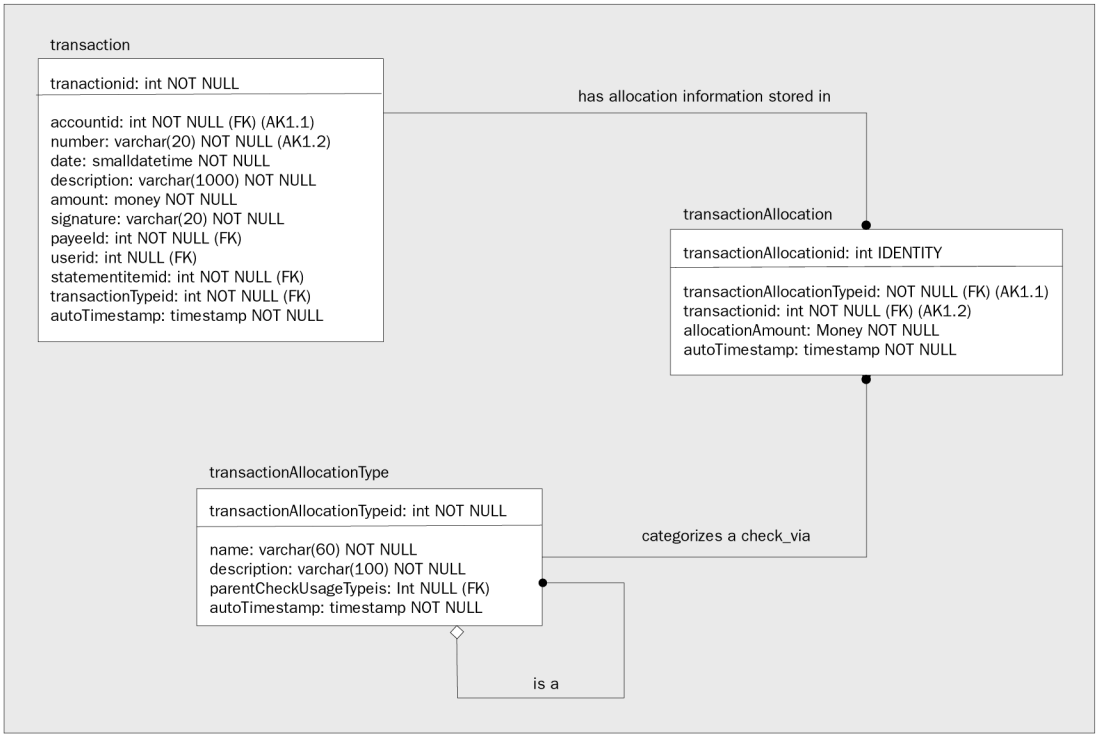

```
CREATE TRIGGER address$afterInsert
ON address
AFTER UPDATE
AS
--------------------------------------------------------------------------------
-- Purpose : Trigger on the <action> that fires for any <action> DML
--------------------------------------------------------------------------------
BEGIN
   DECLARE @rowsAffected int, -- stores the number of rows affected
   @errNumber int, -- used to hold the error number after DML
   @msg varchar(255), -- used to hold the error message
   @errSeverity varchar(20),
   @triggerName sysname

   SET @rowsAffected = @@rowcount
   SET @triggerName = object_name(@@procid) --used for messages

   --no need to continue on if no rows affected
   IF @rowsAffected = 0 RETURN

   DECLARE @numberOfRows int
   SELECT @numberOfRows = (SELECT count(*)
                           FROM INSERTED
                                   JOIN zipCodeCityReference AS zcr
                                        ON zcr.cityId = INSERTED.cityId
and
                                           zcr.zipCodeId =
INSERTED.zipCodeId)
   IF @numberOfRows <> @rowsAffected
   BEGIN
      SET @msg = CASE WHEN @rowsAffected = 1
                      THEN 'The row you inserted has ' +
                           'an invalid cityId and zipCodeId pair.'
                      ELSE 'One of the rows you were trying to insert ' +
                           'has an invalid cityId and zipCodeId pair.'
                 END
   END
   RAISERROR 50000 @msg
   ROLLBACK TRANSACTION
   RETURN
END
```

Ninety-nine percent of the insert trigger is the same code, so this is left to the reader.

3022_12_CMP1 22/12/03 3:21 pm Page 362

Transaction Allocations Cannot Exceed the Amount Stored Trigger

In this example, you consider the transaction allocation amounts for transactions. The column you're concerned with is the allocationAmount on the transactionAllocation table:



Whenever a user modifies an existing allocation, you check to make sure that he hasn't allocated more money to the transaction than what actually exists. So you write the following trigger:

```
CREATE TRIGGER transactionAllocation$afterInsert
ON transactionAllocation
AFTER INSERT AS
-------------------------------------------------------------------------------
-- Purpose : Trigger on the insert that fires for any insert DML
-- : * protects against allocations that are greater than 100%
-------------------------------------------------------------------------------
BEGIN
   DECLARE @rowsAffected int,     -- stores the number of rows affected
   @errNumber int,                -- used to hold the error number after DML
   @msg varchar(255),             -- used to hold the error message
   @errSeverity varchar(20),
   @triggerName sysname

   SET @rowsAffected = @@rowcount
   SET @triggerName = object_name(@@procid) --used for messages
```

3022_12_CMP1 22/12/03 3:21 pm Page 363


```
        -- no need to continue on if no rows affected
        IF @rowsAffected = 0 RETURN

        -- get the total of all transactionAllocations that are affected by your
        -- insert and get all transactions affected

        IF EXISTS (
           SELECT * FROM (
              SELECT transactionId, sum(amount) AS amountTotal
              FROM transactionAllocation
                                    --note this transactionId expands your
                                    --query to look at all allocations that
                                    --are for any transaction who's
                                    --allocation you have touched
              WHERE transactionId IN (SELECT transactionId FROM INSERTED)
              GROUP BY transactionId ) AS allocAmounts

           -- join to the transaction to get the amount of
           -- the transaction
           JOIN [transaction]
              ON allocAmounts.transactionId = [transaction].transactionId

           -- check to make sure that the transaction amount isn't greater
           -- than the allocation amount
           WHERE [transaction].amount > allocAmounts.amountTotal )
           BEGIN
              SET @msg = CASE WHEN @rowsAffected = 1
                              THEN 'The row you inserted has' +
                                   'made the transaction exceed its transaction.
    '
                              ELSE 'One of the rows you were trying to insert '
    +
                                   'has made the transaction exceed its ' +
                                   'transaction.'
                         END
           END
        RAISERROR 50000 @msg
        ROLLBACK TRANSACTION
        RETURN
    END
```

Note that you solved a fairly complex problem using a complex query, and you've used neither cursors nor temporary tables. It's important to formulate your queries in triggers using as few operators as possible, so that you can minimize the possibility of introducing logic errors into very hard-to-find places when building complex queries. There will always be trade-offs in usability vs. readability or understandability. In almost any system, slow triggers will be a problem, especially if you have to insert a large number of rows at a time.

Again, 95 percent of the update trigger for this and the `transaction` table is the same code, so it isn't included here.

3022_12_CMP1 22/12/03 3:21 pm Page 364

Optional Rules

In your system, you have a few optional business rules that need implementing. The following were identified in your original analysis:

- **User should get a warning if they attempt to withdraw too much money**: You would likely create a stored procedure that checks the balance of the user account prior to saving a transaction amount. The client-side code would compare the balance with the amount of the transaction that was attempting to complete.
- **Account must be balanced weekly**: When starting up, the application would need to call a stored procedure that finds out when the account was last balanced, compares it to the current date, and issues a reminder or warning: a reminder might be sent to a manager to advise if the balancing was severely overdue.

In Chapter 14 you'll look at the best practices for building stored procedures.

Best Practices

- **Use triggers to perform data validations that check constraints cannot handle**: The work of some trigger functionality may be moved off into middle-tier objects, though triggers do have performance benefits. Use triggers when the following types of validations need to be made:
  - ***Cross-database referential integrity:*** Just basic RI, but SQL Server doesn't manage declarative constraints across database boundaries.
  - ***Intra-table inter-row constraints:*** For example, when you need to see that the sum of a column value over multiple rows is less than some value (possibly in another table).
  - ***Inter-table constraints:*** For example, if a value in one table relies on the value in another. This might also be written as a functions-based check constraint.
  - ***Introducing desired side effects to your queries:*** For example, cascading inserts, maintaining denormalized data, and so on.
- **Make sure that triggers are able to handle multi-row operations**: Although most modifications are in terms of a single row, if a user enters more than one row there is a possibility of invalid data being entered.
- **Use client code as a last resort**: When constraints and triggers will not solve the problem, then you cannot implicitly trust that the data in the table will meet the requirements. This is because triggers and constraints cannot be gotten around unless a conscious effort is made by dropping or disabling them.
  - *Note: This best practice pertains only if you use SQL Server as more than a data storage device, which is a role that SQL Server plays very well. Many systems will use it with only base table structures and unique constraints (if that). However, the best practice for using SQL Server is to apply the protection code as close to the table as possible so no mistakes can be made by having code to protect the data in multiple locations.*

3022_12_CMP1 22/12/03 3:21 pm Page 365

Summary

You've now finished the task of developing the data storage for your databases. In the last two chapters you've built the physical storage for the data by creating the tables. In this chapter you took the next step and completed your scheme to safeguard it. During this process, you looked at the following resources that SQL Server gives you to protect your data from having invalid values:

- **Defaults**: Though you might not think defaults can be considered data-protection resources, you should know that they can be used to automatically set columns where the purpose of the column might not be apparent to the user (and the database adds a suitable value for the column).
- **Check constraints**: These are very important in ensuring that your data is within specifications. You can use almost any scalar functions (user-defined or system) as long as you end up with a single logical expression.
- **Triggers**: Triggers are very important resources that allow you to build pieces of code that fire automatically on any `INSERT`, `UPDATE`, and `DELETE` operation that is executed against a single table.
- **Stored procedures/front-end code**: I've simply made mention of these mechanisms of data protection because you'll want to ensure that they are used prudently in handling business rules.

Throughout this process you've made use of user-defined functions, which are a new resource that SQL Server 2000 offers you. With them you can encapsulate many of the algorithms that you employ to enhance and extend your defaults, check constraints, triggers, and as you'll find in the next chapter, stored procedures.

Once you've built and implemented a set of appropriate data-safeguarding resources, you can then trust that the data in your database has been validated. You should never need to revalidate your data once it's stored in your database, but it's a good idea to do random sampling so you know that there are no integrity gaps that have slipped by you. Every relationship, every check constraint, every trigger that you devise will help to make sure this is true.

In the next chapter, you'll look at some more advanced data access and modification methods, in particular the use of views and stored procedures.

3022_12_CMP1 22/12/03 3:21 pm Page 366