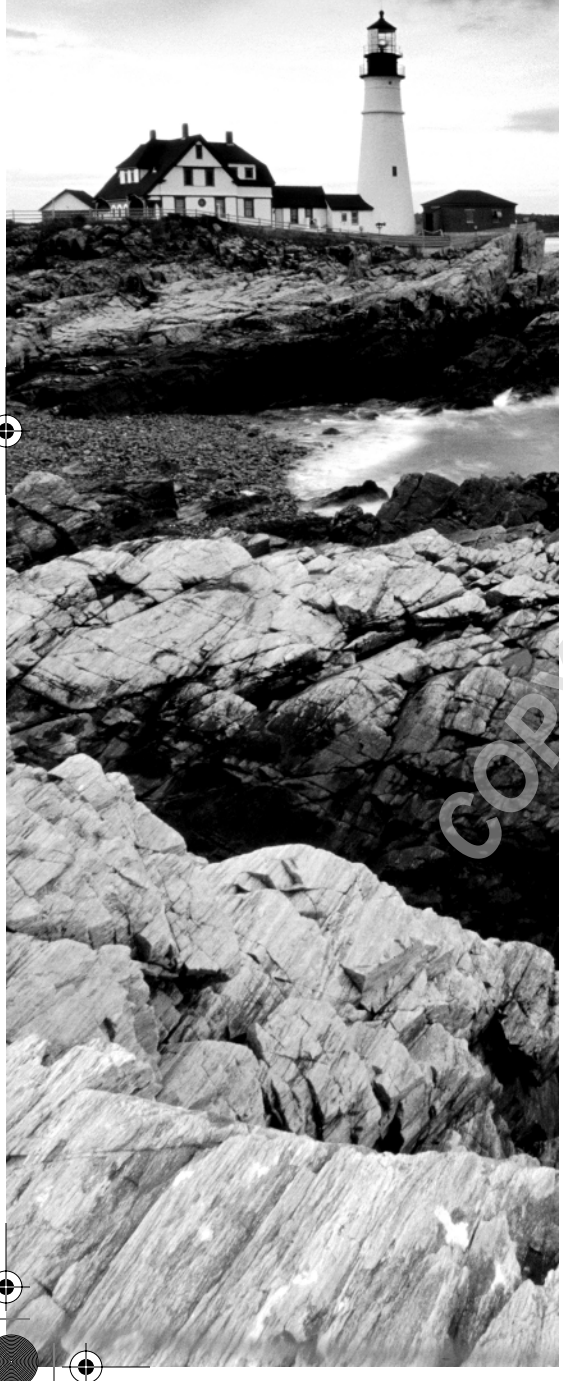
Chapter 1



Designing a Database Solution

MICROSOFT EXAM OBJECTIVES COVERED IN THIS CHAPTER:

- ✓ **Design a logical database.**
  - Design a normalized database.
  - Optimize the database design by denormalizing.
  - Design data flow architecture.
  - Design table width.
- ✓ **Design an application solution that uses appropriate database technologies and techniques.**
  - Design a solution for storage of XML data in the database.
- ✓ **Design objects that define data.**
  - Design user-defined data types.
  - Design tables that use advanced features.
- ✓ **Design attributes.**
  - Decide whether to persist an attribute.
  - Specify domain integrity by creating attribute constraints.
  - Choose appropriate column data types and sizes.
- ✓ **Design entities.**
  - Define entities.
  - Define entity integrity.
  - Normalize tables to reduce data redundancy.
  - Establish the appropriate level of denormalization.
- ✓ **Design entity relationships (ERs).**
  - Specify ERs for referential integrity.
  - Specify foreign keys.
  - Create programmable objects to maintain referential integrity.

Designing databases is becoming somewhat of a lost art form. Unfortunately, not enough energy is spent on the initial database design. Considering the life span of your database solution and the dependency of this solution on the fundamental database design, you should commit the necessary time to designing a framework that fulfills your current requirements and that can evolve as these requirements change.

You should invest in the process of a formal logical database design, followed by a review, before you translate that model into your physical database design. In this chapter, I will go through the various issues you should consider in both the logical and physical database designs.

Database developers are invariably concerned about performance. Although performance is important, it should not be your primary concern at this stage in the implementation of your database solution. At this stage in your database solution's life cycle, you should be more concerned about data integrity, so I will go through the different types of data integrity before covering the various ways in which you can enforce your data integrity.

SQL Server 2005 has a great range of features to take advantage of, so make sure you are aware of the product's capabilities. This will enable you to deliver an optimal database solution that will keep everyone happy: the developers, the database administrators (DBAs), management, and those "ever-demanding" users.

Designing a Logical Database

Designing a database solution customarily begins with a logical database design. This logical database design cycle generally involves the following processes:

- Determining the data to be stored
- Determining the data relationships
- Logically structuring the data

When designing the logical database model, it is important to keep the users in mind. (Unfortunately, they are quite important!) So, your logical database model should be organized into logical entities that are easily understood and maintained. Appropriate naming conventions are also important, although beyond the scope of this chapter.

The logical database model should reduce data repetition, and it typically does this through a process known as *normalization*.

Understanding Normalization

Simplified, *normalization* is all about eliminating duplicate data in a relational database design. Although you can adopt a formal process, you will find that most database developers tend to naturally design normalized databases instead.

The Most Difficult Question: What Is Normalization?

How many developers or DBAs do you meet who can remember the database fundamentals, such as "Codd's Twelve Rules," functional dependencies, and transitive dependencies, and are able to define normalization and the five normal forms? I've been training SQL Server worldwide for more than 10 years, and students generally struggle. Out of curiosity, I returned to my lecture notes from the University of New South Wales to see how my lecturer, Geoff Whale, defined *normalisation* (we spell it differently in Australia): "Normalisation—simply 'common sense.'" Thanks, Geoff....

Basically, normalization is about converting an *entity* into tables of progressively smaller degree and cardinality until you reach an optimum level of decomposition. So, no (or little) data redundancy exists.

Removing redundancy achieves a number of benefits. First, you save space by storing a data element only once. Second, it is then easier to maintain data consistency because you have only one instance of the data element to maintain.

However, it is important to understand the disadvantage of normalization: that you need to join tables again to retrieve related data. Join operations are one of the most expensive operations in any relational database management system (RDBMS), including SQL Server. So, it is possible to "overnormalize," but I'll talk more about that in the "Understanding Denormalization" section.

The formal normalization process typically starts with unnormalized data and progressively goes through a number of classifications called *normal forms* (NFs). Several normal forms exist; in fact, there are to date theories for six normal forms, with variations such as the Boyce-Codd and domain/key normal forms. Generally, however, third normal form (3NF) is commonly sufficient for most database solutions and is what is strived for in an initial, good, logical database design.

You'll now go through the first three normal forms using, as an example, a customer order form that you need to design a database model for, as shown in Figure 1.1.

You can also represent the customer order form, using Victor's Notation, as follows:

```
Order(OrderNumber, CustomerNumber, Name, PassportNumber, Address, Region,
PostCode, Country, OrderDate (ProductNumber, Product, Quantity, UnitPrice))
```

FIGURE 1.1 Customer order form

Customer Order

Order Number: 19710622 | Order Date: 20-Sep-2001

Customer Number: 69 | Customer Name: Marion Siekierski | Passport Number: P00PI

Customer Address: 2 Arco Road, Garmisch-Partenkirchen, Dili, East Timor

Product Number	Product	Quantity	Unit Price
DILI001	Database Analysis	1	$ 5,000.00
DILI002	Cherry Cheesecake	1	$ 6.99
DARWIN001	Muffin	1	$ 3.30
SEMINYAK001	Pink Silly Cow	1	

Achieving First Normal Form

A table is considered to be in first normal form (1NF) if it meets the following conditions:

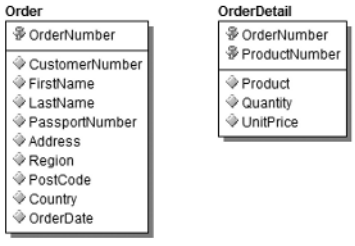
- Every column is atomic; it cannot be further decomposed into more subcolumns.
- It is a valid table, so you separate any repeating groups or multivalued columns.
- A unique key has been identified for each row. This primary key is typically denoted as being underlined.
- All *attributes* are functionally dependent on all or part of the key.

So in this example, you need to decompose the [Name] attribute and separate the repeating group representing the OrderDetail tuple:

```
Order(OrderNumber, CustomerNumber, FirstName, LastName,
 PassportNumber, Address, Region, PostCode, Country,
 OrderDate)
OrderDetail(OrderNumber, ProductNumber, Product, Quantity,
 UnitPrice)
```

Figure 1.2 shows the end result.

FIGURE 1.2 1NF



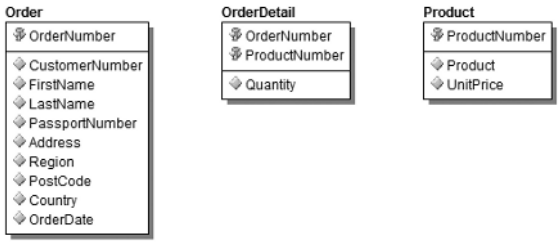
Achieving Second Normal Form

The normal forms are cumulative, so on top of being in 1NF, a table is in second normal form (2NF) when all nonkey attributes are fully functionally dependent on the entire key. So, you are effectively identifying and getting rid of partial dependencies. Look for composite keys and where an attribute might be dependent on only one of the key columns. In this case, the [Product] and [UnitPrice] attributes depend only on [ProductNumber], not on the ([OrderNumber], ProductNumber]) combination. So to meet 2NF, you need to separate these attributes into a distinct entity:

```
Order(OrderNumber, OrderDate, CustomerNumber, FirstName,
 LastName, PassportNumber, Address, Region, PostCode,
 Country)
OrderDetail(OrderNumber, ProductNumber, Quantity)
Product(ProductNumber, Product, UnitPrice)
```

Figure 1.3 represents this graphically.

FIGURE 1.3 2NF



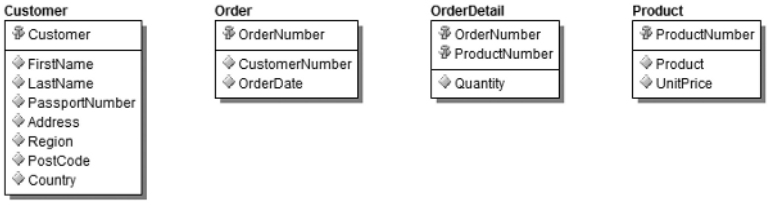
Achieving Third Normal Form

Once in 2NF, you now look for transitive dependencies. *"What, Victor? In plain English, please!"* Basically, you need to ensure that no nonkey attribute depends on another nonkey attribute. In this case, the [FirstName], [LastName], [PassportNumber], [Address], [Region], [PostCode], and [Country] attributes all depend on the [CustomerNumber] attribute, not on the [OrderNumber] attribute. You are basically ensuring that an entity holds only the information related to it. This table will finally be in 3NF:

```
Order(OrderNumber, CustomerNumber, OrderDate)
Customer(CustomerNumber, FirstName, LastName,
 PassportNumber, Address, Region, PostCode, Country)
OrderDetail(OrderNumber, ProductNumber, Quantity)
Product(ProductNumber, Product, UnitPrice)
```

Figure 1.4 shows how you have created a separate entity to store the customer data.

FIGURE 1.4 3NF



Once your logical database design is in 3NF, you have the foundation for a good database solution. The next phase is typically to start the physical database design and design your entities. However, at this stage in the book, it is appropriate for me to discuss denormalization techniques.

Understanding Denormalization

Denormalization is the process of taking a normalized database design and bringing back levels of data redundancy to improve database performance. As discussed, a fully normalized database design will increase the number of join operations that SQL Server will have to perform, but an additional consideration of locking goes on when you are accessing multiple tables to retrieve the required data. By deliberately recombining tables or duplicating data, you can reduce the amount of work SQL Server has to perform as there are fewer joins to perform and locks to maintain. Additionally some techniques of denormalization can dramatically reduce the amount of calculations that SQL Server will have to perform, thereby reducing processor resources.

The obvious disadvantage of denormalization is that you have multiple instances of the same data, which will lead to update anomalies unless you design some mechanism to maintain the denormalized data. This critical design issue depends on two factors. First, the denormalized data has to be up-to-date. Second, you must consider the volatility of the denormalized data in your database solution. Ideally, denormalized data should be nonvolatile.

If your denormalized data needs to be maintained in real time, then you will probably have to take advantage of Data Manipulation Language (DML) triggers. On the other hand, if the data does not have to be 100 percent accurate, you can perhaps get away with running a scheduled Transact-SQL (T-SQL) script after-hours via a SQL Server Agent job.

You can denormalize using different techniques. Figure 1.5 illustrates a *foreign key* being replicated to reduce the number of joins that SQL Server will have to perform. In the denormalized design, you do not need to join the [OrderDetail] table to the [Product] table just to retrieve [DistributorNumber], so SQL Server has one less join to perform.

This is the normalized version:

```
OrderDetail(OrderNumber, ProductNumber, Quantity)
Product(ProductNumber, Product, UnitPrice)
Distributor(DistributorNumber, Distributor, Address, State, PostCode, Country)
Denormalized:
```

```
OrderDetail(OrderNumber, ProductNumber, DistributorNumber Quantity)
Product(ProductNumber, Product, UnitPrice)
Distributor(DistributorNumber, Distributor, Address, State, PostCode, Country)
```

[DistributerNumber] is a good candidate for denormalization since it is typically non-volatile. You could maintain the denormalized data via a *DML trigger* without too much performance overhead.

Figure 1.6 shows an example of a calculated year-to-date sales column being added to the [Product] table because it is a calculation that users frequently need in their reports. So instead of having to run a T-SQL query that would have to use the `SUM()` aggregate function and `GROUP BY` clause, users can simply look up the [YTDSales] column. Not only will this avoid SQL Server having to perform a join and consequently locking the table, this could substantially improve performance if you had millions of records across which the year-to-date sales calculation would have to be made.

FIGURE 1.5 Duplicating the foreign key to reduce the number of joins

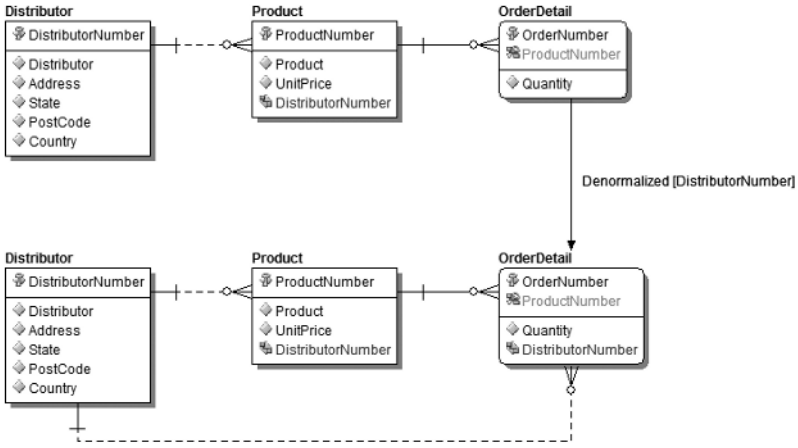
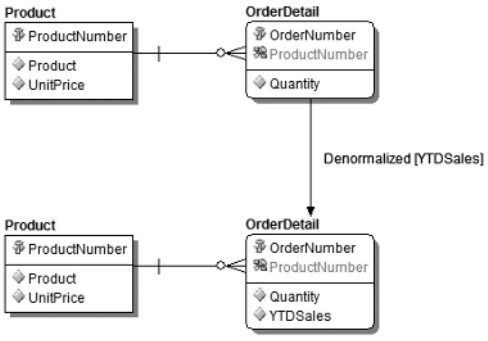


FIGURE 1.6 Denormalization: deriving aggregate data to improve performance



This is the normalized version:

```
OrderDetail(OrderNumber, ProductNumber, Quantity)
Product(ProductNumber, Product, UnitPrice)
Denormalized:
OrderDetail(OrderNumber, ProductNumber, Quantity)
Product(ProductNumber, Product, UnitPrice, YTDSales)
```

The important issue here is how you'll maintain the [YTDSales] column. This depends on your user requirements. If the data needs to be 100 percent accurate at all times, then you have no choice but to implement a DML trigger. On the other hand, if it is not important for the value to be up-to-date in real time, then you could schedule an after-hours T-SQL job that would update the [YTDSales] column at the end of each day.

So when deciding upon the potential level of denormalization in your database design, you need to counterbalance the benefits versus the overhead of maintaining the denormalized data so as not to lose *referential integrity*. You can achieve this only with thorough knowledge of your actual data and the specific business requirements of the users. And understanding how SQL Server's engine works doesn't hurt either!

Designing the Physical Entities

Once you have designed the logical database model, it is now time to turn your attention to the physical implementation of that model. The physical database design generally involves the following processes:

- Determining the data types to be used by the attributes
- Designing the physical table to represent the logical entities
- Designing the entity integrity mechanisms
- Designing the relational integrity mechanisms

I always believe it's useful to know where you're heading. So, it is constructive to see what your end goal is when designing the physical database model. SQL Server Management Studio easily enables you to draw a database diagram of your physical database design that shows the physical tables that make up the database, the attributes, and the properties of the tables and the *relationship*s between them (see Exercise 1.1).

EXERCISE 1.1

Generating a Database Diagram

1. Open SQL Server Management Studio, and connect using Windows Authentication.
2. In Object Explorer, expand *Server* ➢ Databases ➢ AdventureWorks ➢ Database Diagrams.

EXERCISE 1.1 *(continued)*

3. If prompted, select Yes when asked whether you want to create one or more of the support objects required to use database diagramming.
4. Right-click the Database Diagrams folder, and select New Database Diagram.
5. Select all tables in the Add Table dialog box, and click the Add button.
6. Click the Close button.
7. Examine the physical database model in the database diagram window.

When implementing the physical model, you always need to take into account the RDBMS engine you are using. The same logical database model might end up being implemented differently on SQL Server 2005 versus Microsoft Access or even SQL Server 2000.

Real World Scenario

Database Design Is an Iterative Process

In 2000 I was involved in analyzing a database used in the Australian wool industry. As usual, the database had a number of performance problems. In this particular instance, after examining the database design, I found that no constraints were defined at all. Upon further investigation, I determined this database had been upgraded originally from SQL Server 6.5 to SQL Server 7.0 and, finally, to the then-current SQL Server 2000. At no stage did anyone examine the database design to see whether they could leverage any new features of the latest version of SQL Server.

Your database design should be flexible and constantly reevaluated to ensure you are maximizing the potential of the SQL Server database engine on which you are running. In the case of upgrading to SQL Server 2005, you should be looking at redesigning your entities to take advantage of features such as the XML data type, support for large object data types, persisted columns, partitions, and so on.

Designing Attributes

Every RDBMS will have its own combination of *data type*s it natively supports. The major new additions for SQL Server 2005 are native support for XML data and the "dreaded" common language runtime (CLR) user-defined data types. In fact, SQL Server 2005 has a number of options as far as user-defined data types are concerned, but you will examine them in Chapter 2. You are primarily concerned with native system supplied data types here.

Understanding Types of Data

The modern database engine has to support a number of types of data in today's demanding business environments:

- Structured
- Semistructured
- Unstructured

It is important to understand the characteristics and differences of data types so as to be able to correctly choose the appropriate data type for the attributes in your database design.

Structured Data

Structured data can be organized in semantic groups or entities. Similar entities are grouped together by using relations or classes because they have the same descriptions or attributes. Structured data tends to have a well-defined, well-known, and typically rigid schema. This is the type of data that an RDBMS typically excels at storing and managing and traditionally is what developers have all been using over the last decade.

Structured data provides various advantages. You can normalize the data into various tables and thereby reduce redundancy, as discussed. Structured data also tends to lend itself to efficient storage and optimized access and is easy to define data integrity and business rules against.

Semistructured Data

Semistructured data represents data whose schema is not well defined or may evolve over a period of time. Although semistructured data is often organized in semantic entities where similar entities are grouped together, entities in the same group may not have the same attributes. XML is a widely used form of semistructured data. It excels as a means of storing emails and business-to-business (B2B) documents that tend to be semistructured.

Semistructured data is suitable, as indicated, when the structure of the data is not known completely or is likely to change significantly in the future. However, Extensible Markup Language (XML) also has a few disadvantages. Every element of data has to be marked with tags (which represent the schema), and this increases storage space; therefore, it is nowhere near as efficient as structured data types. Searching semistructured data is a little more complex because the schema is potentially different for each record.

Unstructured Data

Unstructured data has no schema whatsoever, so it is typically unpredictable. Examples of unstructured data include free-form text, Microsoft Office documents, and images.

The advantage of unstructured data is that you can change the format of the data at any time without any major database redesign. However, it is difficult to search using "traditional" techniques. SQL Server provides support for such Binary Large Objects (BLOBs) and has a "revamped" full-text indexing search engine to facilitate searches on free-form text.

Choosing Data Types and Sizes

From Databases 1.01: "Choose the smallest data type possible that can contain the range of values for that particular attribute." It's as simple as that. Mostly, anyway! You still need to take into account how SQL Server 2005 stores data rows when creating your table schemas, but generally this advice is sound.

Table 1.1 shows all the system-supplied native data types supported by SQL Server 2005, the range of values they support, and how much space they consume on disk.

TABLE 1.1 Native SQL Server 2005 Data Types

Data Type	Range	Storage
BIT	0 to 1	1 byte/eight BIT fields.
TINYINT	0 to 255	1 byte.
SMALLINT	–32,768 to 32,767	2 bytes.
INT	–2,147,483,648 to 2,147,483,647	4 bytes.
REAL	–.40E+38 to –1.18E–38, 0, and 1.18E–38 to 3.40E+38	4 bytes.
SMALLMONEY	–214,748.3648 to 214,748.3647	4 bytes.
SMALLDATETIME	January 1, 1900, through June 6, 2079	4 bytes.
BIGINT	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	8 bytes.
MONEY	–922,337,203,685,477.5808 to 922,337,203,685,477.5807	8 bytes.
DATETIME	January 1, 1753, through December 31, 9999	8 bytes.
TIMESTAMP	Generally used as a means of version-stamping rows	8 bytes.
FLOAT(*n*)	–1.79E+308 to –2.23E–308, 0, and 2.23E–308 to 1.79E+308	Variable. Depends on (*n*): **(*n*) Value** **Storage** 1–24 4 bytes 25–53 8 bytes

TABLE 1.1 Native SQL Server 2005 Data Types *(continued)*

Data Type	Range	Storage
UNIQUEIDENTIFIER	Globally unique identifier (GUID) value matching xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx mask	16 bytes.
DECIMAL(*p, s*) NUMERIC(*p, s*)	Depends on fixed precision (*p*) and scale (*s*); –10^38 +1 through 10^38 –1	Variable. Depends on (*p*): **Precision (*p*)** **Storage** 1–9 5 bytes 10–19 9 bytes 20–28 13 bytes 29–38 17 bytes
BINARY(*n*)	1 to 8,000 bytes	(*n*) bytes.
CHAR(*n*)	1 to 8,000 characters	(*n*) bytes.
NCHAR(*n*)	1 to 4,000 Unicode characters	(*n* * 2) bytes.
VARBINARY(*n*)	1 to 8,000 bytes	Variable. Storage is the actual data length plus 2 bytes.
VARCHAR(*n*)	1 to 8,000 characters	Variable. Storage is the actual data length plus 2 bytes.
NVARCHAR(*n*)	1 to 4,000 Unicode characters	Variable. Storage is the actual data length plus 2 bytes.
SQL_VARIANT	1 to 8,000 bytes	Variable. Maximum storage of 8,016 bytes.
VARCHAR(MAX) NVARCHAR(MAX) VARBINARY(MAX) XML IMAGE TEXT NTEXT	Up to 2GB	Variable. Maximum storage of 2,147,483,647 bytes.

When choosing appropriate data types for your attributes, keep the following recommendations in mind:

- Generally avoid using the FLOAT and REAL data types because they are imprecise. Remember, floating-point data is approximate and consequently cannot be represented exactly. In other words, what you put in might not be what you get out.

- The DATETIME data type includes the time, down to an accuracy of 3.33 milliseconds. The SMALLDATETIME data type includes it down to 1 second. Be careful when working with these data types because you always need to take the time component into account. The best solution if you are interested in storing just the date is to zero out the time component programmatically, such as with a DML trigger.

I am pretty sure an American National Standards Institute (ANSI) DATE data type has been one of the most requested features for SQL Server in the last 10-plus years. Not XML.... Not CLR.... The first person who explains to me satisfactorily why Microsoft has not yet implemented this basic feature will get lunch and drinks on me!

- Be careful with using the VARCHAR(MAX), NVARCHAR(MAX), and VARBINARY(MAX) data types because they might slow down performance. You will examine this in more detail shortly when you learn how SQL Server 2005 stores data rows.
- A single BIT field will still take up a byte, because SQL Server has to byte-align all fields. You might be better off implementing such an attribute as a more meaningful CHAR(1) field. For example, you might require a [Sex] field in the [Customer] table, which you are considering to implement as a BIT field. However, if it is the only BIT field in the table, you might consider implementing it as a CHAR(1) field because it will take up the same amount of space and the M and F values will be easier to interpret.
- The NCHAR, NVARCHAR, and NTEXT data types all use 2 bytes to store data because they use Unicode encoding (the Unicode UCS-2 character set), which is useful for storing multi-lingual character data. However, it does take up twice the amount of space, which can affect performance.
- Use the UNIQUEIDENTIER data type to store GUIDs, which, although not guaranteed to be unique, can be considered unique because the total number of unique keys (2,128, or 3.4028 1038) is so large that the possibility of duplicate numbers being generated twice is slight. Some developers use GUIDs as pseudorandom numbers. Be careful of using UNIQUEIDENTIFIER columns with clustered indexes because they have limited value, being 16 bytes in length. You use the NEWID() SQL Server system function to generate the GUID value.
- You should use the VARBINARY data type to store BLOBs such as Microsoft Word and Adobe Acrobat documents, images such as JPEG files, and the like.
- Use the VARCHAR and NVCHAR data types to store free-form text. The problem with free-form text is that it cannot be efficiently queried by using traditional SQL Server B-Tree indexes. In this case, you should investigate how full-text search and full-text indexes have been implemented in SQL Server 2005.

The NTEXT, TEXT, and IMAGE data types will be removed in a future version of Microsoft SQL Server. Use NVARCHAR, VARCHAR, and VARBINARY instead.

The system-supplied native data types have not really changed significantly since SQL Server 2000. You can still take advantage of user-defined data types. The major new additions to SQL Server 2005 are native support for XML and CLR user-defined data types.

XML Data Type

Sigh…how many arguments have I had over this one? As with all technology, it's all in the implementation and not the technology itself. The benefits of XML are many. XML provides a text-based means to describe and apply a hierarchical structure to information, be it a record, list, or tree. Being based on text, XML is simultaneously computer- and human-readable and unencumbered by licenses or restrictions. It supports Unicode and so natively supports multiple languages. All these features make it a great mechanism for the following:

- Data exchange as a transport mechanism
  - B2B
  - Business-to-consumer (B2C)
  - Application-to-application (A2A)
- Document management
  - Office XML documents
  - Extensible HTML (XHTML)
- Messaging
  - Simple Object Access Protocol (SOAP)
  - Really Simple Syndication (RSS)
- Middle-tier collaboration

The million-dollar question of course is, what do you do with XML inside SQL Server? When working with XML data, you need to choose whether you should use the native XML data type, use some BLOB data type, or shred the XML into relational columns. Your considerations depend on a number of factors:

- Whether you need to preserve document order and structure, or *fidelity*. The various data types will store the XML data differently.
- Whether you want to further query your XML data at a finer grain. The frequency of querying the XML data will be a factor.
- Whether you need modify your XML data at a finer grain. You will also need to consider how often the XML data will be modified.
- Whether you want to speed up XML queries by indexing. You can create XML indexes on XML fields. SQL Server can index all tags, values, and paths for an XML field. I will cover this in more detail in Chapter 2.
- Whether you have a schema for your XML data.
- Whether your database solution requires system catalog views to administer your XML data and schemas.

If you shred an XML document and then later reconstitute it, SQL Server does not guarantee that it will be identical.

SQL Server 2005 implements the ISO SQL-2003 standard XML data type and supports both untyped and typed XML. Microsoft recommends using the untyped XML in these cases:

- When you do not have a schema for your XML data
- When you do not want SQL Server to validate the XML data, although you have the schema

Microsoft recommends using typed XML data in these cases:

- When you have schemas for your XML data and you want SQL Server to validate the XML data accordingly
- When you want to take advantage of storage and querying compilation that comes with the typed XML data

So, have I answered the question? Well...not exactly. I think of XML as being similar to "nuclear weaponry"—just because you have it doesn't mean you should use it. It depends on your business requirements more than anything else. But if you require ad hoc modeling of semistructured data, where an object's properties are sparsely populated, the schema is changing, or the multivalues properties do not fit into your traditional relational schema, then you should strongly consider using XML.

T-SQL User-Defined Data Types

SQL Server 2005 allows you to create T-SQL user-defined data types, which are basically aliases to existing system-supplied native data types. Most new database solutions don't implement these *alias types* because "they do not bring anything new to the table." However, the following are still good reasons to take advantage of them:

- When you are porting a database solution from another RDBMS that uses a data type that SQL Server does not natively support, they allow the database creation scripts to run seamlessly. Alternatively, use them if you are implementing a database solution on different RDBMS engines, such as when creating a [BOOLEAN] alias type so as to be compatible with Microsoft Access.
- They are a great way of standardizing data types for commonly used attributes, such as [Name], [LastName], and [PostCode], to ensure compatibility.

If you create any user-defined data types inside the **model** system database, those data types will be automatically generated in any new databases that you subsequently create. This represents a great technique of implementing "standards" inside an organization.

The partial syntax for creating a user-defined data type is as follows:

```
CREATE TYPE [ schema_name. ] type_name
{
  FROM base_type
  [ ( precision [ , scale ] ) ]
  [ NULL | NOT NULL ]
 | EXTERNAL NAME assembly_name [ .class_name ]
} [ ; ]
```

You should no longer use the `sp_addtype` system stored procedure to create user-defined data types. This system stored procedure will be removed in a future version of Microsoft SQL Server. Use `CREATE TYPE` instead.

So if you decided to create a user-defined data type for sex inside your database, you would execute the following T-SQL code:

```
CREATE TYPE [dbo].[Sex]
FROM CHAR (1) NOT NULL ;
GO
```

So let's go through a very simple example of creating a T-SQL user-defined data type (see Exercise 1.2).

EXERCISE 1.2

Creating a T-SQL User-Defined Data Type

1. Open SQL Server Management Studio, and connect using Windows Authentication.
2. In Object Explorer, expand *Server* ➢ System Databases ➢ tempdb ➢ Programmability ➢ Types.
3. Right-click User-Defined Data Types, and click New User-Defined Data.
4. Enter **Sex** in the Name text box.
5. Select Char in the Data Type drop-down list.
6. Change the Length value to 1.
7. Click the Allow NULLS check box.
8. Click the OK button.

CLR User-Defined Data Types

A new feature of SQL Server 2005 is the support for CLR user-defined data types. The capability of going beyond the traditional native data types of SQL Server is a powerful feature as far as extensibility is concerned, but you need to consider the impact on performance, because the data types will perform slower. You will also probably need to write your own functions to manipulate your CLR user-defined data types because the native SQL Server 2005 functions will have limited functionality.

Generally, you should try to use the native data types provided by SQL Server 2005. Consider implementing CLR user-defined data types when you have complex requirements such as a need for multiple elements and/or certain behavior; however, again, you could potentially use the XML data type. CLR user-defined data types are well suited to the following:

- Geospatial data
- Custom date, time, currency, and extended numeric data
- Complex data structures such as arrays
- Custom encoded or encrypted data

Having said all that, you will probably never have to implement CLR user-defined data types, which is probably for the best (but I sleep better at nights knowing they are there).

To create a CLR user-defined data type, follow this procedure:

1. Code and build the assembly that defines the CLR user-defined data type using a language that supports the Microsoft .NET Framework, such as Microsoft Visual C# and Microsoft Visual Basic .NET.
2. Register the assembly in SQL Server 2005 using the T-SQL `CREATE ASSEMBLY` statement, which will copy the assembly into the database.
3. Create the CLR user-defined data type within the database by using the T-SQL `CREATE TYPE` statement I covered previously.

By default, to reduce the surface area of attack, the CLR integration feature of SQL Server 2005 is turned off. You will need to enable it to create CLR user-defined data types.

Turning on CLR Integration in SQL Server 2005 is done through the Surface Area Configuration tool which we cover in mode detail in Chapter 4. Let's turn on the CLR functionality of your SQL Server 2005 instance in Exercise 1.3.

EXERCISE 1.3

Turning on CLR Integration in SQL Server 2005

1. Open SQL Server Surface Area Configuration.
2. Click Surface Area Configuration for Features.
3. Click CLR Integration.
4. Click the Enable CLR Integration check box.
5. Click the OK button.

Designing Domain Integrity

Domain (or column) integrity controls the values that are valid for a particular column. You can enforce *domain integrity* through a number of mechanisms:

- The data type assigned to that column
- Whether the column definition allows NULL values
- Procedural code in an insert or update DML trigger
- A foreign key constraint that references a set of primary key values in the parent table
- A check constraint defined on that column

You should no longer use database rule objects to enforce domain integrity because they are being deprecated in later versions of SQL Server. They are available only for backward compatibility in SQL Server 2005.

Basically, you should use a check *constraint* to maintain domain integrity whenever possible because, like the other constraints, they are fast.

The partial syntax for creating a primary key or unique constraint is as follows:

```
ALTER TABLE [database_name .[ schema_name ] . |
schema_name . ] table_name
[ WITH { CHECK | NOCHECK } ]
ADD CONSTRAINT constraint_name
CHECK (logical_expression)
```

So if you decided to create a check constraint that would ensure that the [Passport-Number] column of the [Customer] table matched a pattern, you would execute the following T-SQL code:

```
ALTER TABLE [Customer]
ADD CONSTRAINT [CK_Customer(CustomerNumber)]
CHECK ([PassportNumber]
LIKE '%[0-9][0-9][0-9][0-9][0-9][0-9][0-9]') ;
GO
```

The problem with using check constraint statements is that they are limited in the following ways:

- They can reference other columns only in the same table.
- They cannot call subqueries directly.

So for more complex domain integrity requirements, you will have to resort to some of the other mechanisms of enforcement.

Designing Entities

Once you have determined the appropriate data types for the attributes of your entities, you are ready to create the physical database model. This process typically involves the following:

- Defining the entities
- Defining entity integrity
- Defining referential integrity

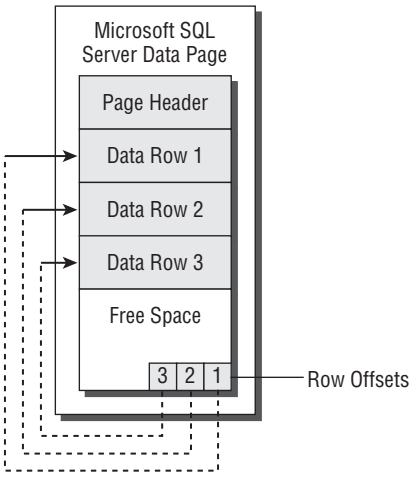
By now you should be done with most of the design work, so it's more a case of being familiar with the T-SQL syntax for creating a table and the various options. Well…not quite. You still might need to redesign your entity to take into account how the SQL Server 2005 storage engine physically stores the data rows in your database files, because this can have a dramatic impact on performance.

Without being overly concerned with database files, extents, and pages (although it would help dramatically, so off you go to research those terms more!), it is sufficient to say that the basic unit of input/output (I/O) in SQL Server 2005 is an 8KB page.

Understanding How SQL Server Stores Data Rows

SQL Server 2005 stores data rows sequentially after a 96-byte header of the page, as shown in Figure 1.7. A row offset table starts at the end of the page and contains one entry for each row located on that page. Each entry keeps track of the beginning of the row relative to the start of the page. The entries in the row offset table are in reverse sequence from the rows on the page.

FIGURE 1.7 SQL Server data page structure



Generally speaking, a row cannot span a page boundary. So, a table can store a maximum of 8,060 bytes per row. However, SQL Server 2005 relaxes this restriction for the `VARCHAR`, `NVARCHAR`, `VARBINARY`, `SQL_VARIANT`, and CLR user-defined data types. SQL Server 2005 does this by taking advantage of a special IN_ROW_DATA allocation unit that is used to store these overflow columns and tracks them via a special 24-byte pointer in the original data row.

When the row wants to grow beyond the page limit, SQL Server automatically moves one or more of these variable-length columns to the ROW_OVERFLOW_DATA allocation unit. If subsequent DML operations shrink the data row, SQL Server dynamically moves the column to the original data row. Consequently, you need to take into account the percentage of rows likely to overflow, the frequency they will be queried (as they will be slower), and the frequency at which these rows will be modified (because this will also slow performance).

You might be better off redesigning your database design by storing these columns in a separate table that has a one-to-one relationship with the original table design. It is also a good idea to calculate the *page density*, or how much of your 8KB page SQL Server 2005 will be using, because it might indicate a need to redesign your database. Ideally, you want to see a small percentage of the page not being used by SQL Server. In Exercise 1.4, you'll learn how to calculate page density.

EXERCISE 1.4

Calculating Page Density

1. Calculate the record size of the table, assuming the maximum for variable length data types, using the data type lengths in Table 1.1.

Column	Data Type
WhaleID	INT
WhaleName	NVARCHAR(20)
WhaleGender	CHAR(1)
Transmitter	UNIQUEIDENTIFIER
DateTagged	SMALLDATETIME
WhaleSpecies	VARCHAR(20)
LastSighting	SMALLDATETIME

2. Divide 8,092 by the record size from step 1.
3. Round the result from step 2 *down* to an integer to find the number of rows that will fit per page.
4. Multiply the record size calculated from step 1 by the integer from step 3 to determine how much space will be consumed by the data rows.

EXERCISE 1.4 *(continued)*

5. Subtract the figure from step 4 from 8,092 to determine the amount of space on each data page that SQL Server will not be able to use.

So, if your record size is 3,000 bytes in size, you would have more than 2,000 bytes of free space on each page that SQL Server 2005 could not use. This would represent more than 25 percent wasted space on each page, which is quite a high figure. In this case, I would talk to the developers to see how you could redesign the table schema.

Defining Entities

SQL Server 2005 supports up to two billion tables per database and 1,024 columns per table. The number of rows and total size of a table is virtually unlimited. As discussed, the maximum number of bytes per row is 8,060, although this restriction is for the `VARCHAR(n)`, `NVARCHAR(n)`, `VARBINARY(n)`, and `SQL_VARIANT` data types, which can take advantage of overflow pages. Don't forget that the lengths of each one of these columns must still fall within the limit of 8,000 bytes, but their combined widths may exceed the 8,060-byte limit in a table.

You can use the `CREATE TABLE` T-SQL statement to create entities in SQL Server 2005. For all the SQL Server 2005 exams, you should familiarize yourself with the `CREATE TABLE` syntax and the various options available.

```
CREATE TABLE
  [ database_name . [ schema_name ] . | schema_name . ]
table_name
   ( { <column_definition> | <computed_column_definition> }
    [ <table_constraint> ] [ ,...n ] )
  [ ON { partition_scheme_name ( partition_column_name ) |
 filegroup
    | "default" } ]
  [ { TEXTIMAGE_ON { filegroup | "default" } ]
[ ; ]
```

The <column_definition> options allow you to control the columns that make up the table schema and their data types. It is recommended that you always explicitly control nullability instead of relying on any default behavior of SQL Server or your integrated development environment (IDE).

```
<column_definition> ::=
column_name <data_type>
  [ COLLATE collation_name ]
  [ NULL | NOT NULL ]
  [
```

```
    [ CONSTRAINT constraint_name ] DEFAULT
constant_expression ]
   | [ IDENTITY [ ( seed ,increment ) ]
[ NOT FOR REPLICATION ]
  ]
  [ ROWGUIDCOL ] [ <column_constraint> [ ...n ] ]
```

The <data type> options control the schema to which the data type belongs. If the type_schema_name is not specified, SQL Server will reference the type_name in the following order:

- The SQL Server system data type
- The default schema of the current user in the current database
- The [dbo] schema in the current database

```
<data_type> ::=
[ type_schema_name . ] type_name
  [ ( precision [ , scale ] | max |
    [ { CONTENT | DOCUMENT } ] xml_schema_collection ) ]
```

The <column_constraint> options allow you to define the various data integrity types through constraints. I'll cover these various constraints later in this chapter. A lot of developers prefer using the ALTER TABLE statement after the initial CREATE TABLE statement because it is easier to read and to possibly reflect their design methodology.

```
<column_constraint> ::=
[ CONSTRAINT constraint_name ]
{   { PRIMARY KEY | UNIQUE }
    [ CLUSTERED | NONCLUSTERED ]
    [
      WITH FILLFACTOR = fillfactor
     | WITH ( < index_option > [ , ...n ] )
    ]
    [ ON { partition_scheme_name ( partition_column_name )
 | filegroup | "default" } ]
 | [ FOREIGN KEY ]
    REFERENCES [ schema_name . ] referenced_table_name
[ ( ref_column ) ]
    [ ON DELETE { NO ACTION | CASCADE | SET NULL |
SET DEFAULT } ]
    [ ON UPDATE { NO ACTION | CASCADE | SET NULL |
SET DEFAULT } ]
    [ NOT FOR REPLICATION ]
 | CHECK [ NOT FOR REPLICATION ] ( logical_expression )
}
```

The <computed_column_definition> options allow you to create a column that is based on a valid T-SQL expression. One of the more important new features to improve performance in SQL Server 2005 is the capability of persisting a computed column in your table schema.

```
<computed_column_definition> ::=
column_name AS computed_column_expression
[ PERSISTED [ NOT NULL ] ]
[
  [ CONSTRAINT constraint_name ]
  { PRIMARY KEY | UNIQUE }
    [ CLUSTERED | NONCLUSTERED ]
    [
      WITH FILLFACTOR = fillfactor
     | WITH ( <index_option> [ , ...n ] )
    ]
  | [ FOREIGN KEY ]
    REFERENCES referenced_table_name [ ( ref_column ) ]
    [ ON DELETE { NO ACTION | CASCADE } ]
    [ ON UPDATE { NO ACTION } ]
    [ NOT FOR REPLICATION ]
  | CHECK [ NOT FOR REPLICATION ] ( logical_expression )
  [ ON { partition_scheme_name ( partition_column_name )
    | filegroup | "default" } ]
]
```

Deciding whether to persist a computed *field* depends entirely on your database solution and environment. You should take a number of considerations into account, however:

- The extra space taken up by the computed field
- How frequently users will query the computed field
- Whether the computed field will be part of the result set, the SARG, or both
- The same considerations of density and selectivity that you would have with an index
- How frequently the data that the computed field is based on changes
- How volatile the table is in general

For an online transaction processing (OLTP) environment where your data changes frequently, you'll be less inclined to persist computed fields because of the overhead on the database engine of having to maintain the computed values.

Furthermore, you can create indexes on computed fields to improve performance when searching on the computed field, but you need to meet a number of requirements:

- The computed field expression is deterministic and precise.
- The computed field expression cannot evaluate to the IMAGE, NTEXT, or TEXT data type.
- All functions referenced by the computed field have the same owner as the table.
- A number of SET options are met.

For a complete list of requirements, look up the "Creating Indexes on Computed Columns" topic in SQL Server 2005 Books Online.

The <table_constraint > options are similar to the <column_constraint> options discussed previously:

```
< table_constraint > ::=
[ CONSTRAINT constraint_name ]
{
  { PRIMARY KEY | UNIQUE }
    [ CLUSTERED | NONCLUSTERED ]
        (column [ ASC | DESC ] [ ,...n ] )
    [
      WITH FILLFACTOR = fillfactor
      |WITH ( <index_option> [ , ...n ] )
    ]
    [ ON { partition_scheme_name (partition_column_name)
      | filegroup | "default" } ]
  | FOREIGN KEY
       ( column [ ,...n ] )
    REFERENCES referenced_table_name [ ( ref_column
[ ,...n ] ) ]
    [ ON DELETE { NO ACTION | CASCADE | SET NULL |
SET DEFAULT } ]
    [ ON UPDATE { NO ACTION | CASCADE | SET NULL |
SET DEFAULT } ]
    [ NOT FOR REPLICATION ]
  | CHECK [ NOT FOR REPLICATION ] ( logical_expression )
}
```

The <index_option> options allow you to control some fine-tuning mechanism for indexes that might be implemented during table creation. Generally, you should leave the defaults alone unless you have a specific requirement.

```
<index_option> ::=
{
  PAD_INDEX = { ON | OFF }
 | FILLFACTOR = fillfactor
 | IGNORE_DUP_KEY = { ON | OFF }
 | STATISTICS_NORECOMPUTE = { ON | OFF }
 | ALLOW_ROW_LOCKS = { ON | OFF}
 | ALLOW_PAGE_LOCKS ={ ON | OFF}
}
```

So in this particular example, you would execute the following T-SQL code to create the [Customer] table:

```
CREATE TABLE [Customer] (
   [CustomerNumber] INT     NOT NULL,
   [FirstName]      VARCHAR(20)  NULL,
   [LastName]       VARCHAR(20)  NULL,
[PassportNumber] CHAR(8)      NULL,
[Address]        VARCHAR(50)  NULL,
[Region]          VARCHAR(20)  NULL,
[PostCode]       VARCHAR(10)  NULL,
[Country]         VARCHAR(20)  NULL
)
```

Designing Entity Integrity

Simplified *entity (or table) integrity* in relational database theory requires that all rows in a table be uniquely identified via an identifier known as a *primary key*. A primary key can potentially be a single column or a combination of columns.

Sometimes several columns could potentially be the primary key, such as in the case of a [Customer] table that has both a [CustomerNumber] column and a [PassportNumber] column, both of which are guaranteed to have a value and be unique. These columns are referred to as *candidate keys*.

When deciding on your primary key, you need to determine whether you are going to use a natural key or a surrogate key. A *natural key* is a candidate key that naturally has a logical relationship with the rest of the attributes in the row. A *surrogate key*, on the other hand, is an additional key value that is artificially generated by SQL Server, typically implemented as integer column with the identity property.

The advantage of using natural keys is that they already exist, and since they have a logical relationship with the rest of the attributes, indexes defined on them will be used by both user searches and join operations to speed up performance. Obviously, you don't need to add a new, unnatural column to your entity, which would take up additional space.

The main disadvantage of natural keys is that they might change if your business requirements change. For example, if you have a primary key based on the [CustomerNumber] field and it subsequently changes from a numeric to an alphanumeric field, then you will need to change both the data type of the [CustomerNumber] field and all related tables where the [CustomerNumber] field is used as a foreign key. This might not be an easy task for a high-availability SQL Server solution running in a 24/7 environment.

Another, potentially important consideration might exist if you have a compound or composite natural key or a very wide key. This can have a dramatic impact on performance if you have chosen to create a clustered index on such a wide natural key, abecause it will also impact the size of all your nonclustered indexes. Likewise, join operations using a nonclustered index on a wide natural key will not perform as well as a smaller surrogate key.

The main advantage of a surrogate key is that it acts as an efficient index, which is created by the developer typically as an integer field, managed by SQL Server through the identity property, and never seen by the user. Therefore, a surrogate key makes a good potential candidate for a clustered index because of its small size. Since the users don't actually work with the surrogate key, you do not need to *cascade* update operations as well.

You will find a number of people who believe that a primary key value, once inserted, should never be modified. Like Fox Mulder, "I want to believe...." In any case, the concept of a surrogate key works well in this case, because users never work with it directly.

You can implement a primary key, or the effect of a primary key, in a number of ways. The techniques in SQL Server include the following:

- Using a primary key constraint
- Using a unique constraint
- Using a unique index
- Using an insert and update DML trigger that verifies that the primary key value being modified is unique

All of these will have the same effect, ensuring that entity integrity is maintained by disallowing duplicates in the primary key column or columns. However, you should generally use a primary key constraint. Maintaining entity integrity programmatically through procedural

code in triggers or elsewhere is considered error prone (in other words, *you* might goof it up!), involves more work, and is slower, since it works at a "higher" level of the SQL Server engine. Constraints are so much easier to work with, and they perform quicker…so use them!

You can define a primary key constraint only on a column that does not allow NULLs. If you need to allow one NULL value in your primary key, you will need to resort to a unique constraint instead.

The other advantage of using a primary key constraint is that it highlights the "importance" of the field on which it's defined, so many third-party and Microsoft applications will be able to derive the primary key from the primary key constraint, as opposed to relying on naming conventions or indexes.

The partial syntax for creating a primary key or unique constraint is as follows:

```
ALTER TABLE [database_name .[ schema_name ] . | schema_name . ] table_name
ADD CONSTRAINT constraint_name
PRIMARY KEY | UNIQUE [ CLUSTERED | NONCLUSTERED ] (column_name)
[ WITH ( index_options )]
```

So if you decided to create a primary key on the [CustomerNumber] column and a candidate key on the [PassportNumber] column of the [Customer] table, you would execute the following T-SQL code:

```
ALTER TABLE [Customer]
ADD CONSTRAINT [PK_Customer(CustomerNumber)]
PRIMARY KEY (CustomerNumber) ;
GO
ALTER TABLE [Customer]
ADD CONSTRAINT [UQ_Customer(PassportNumber)]
UNIQUE (PassportNumber) ;
```

By default, SQL Server 2005 will create a clustered index for the primary key constraint in the previous T-SQL script. This might not be optimal for your particular table. Generally, I recommend that you don't rely on such default behavior and always explicitly script all such options.

It is a highly recommended practice to have a primary key defined on all tables in a relational database. The main reason, of course, is that it will ensure entity integrity; however, it also makes life a lot easier for developers—and especially for contractors coming onto your site to help sort out the mess the developers have made. Ha! Be aware that there can also be performance issues and other unexpected implications in the future.

Real World Scenario

Importance of Primary Key Constraints

Just this week in Sydney, I had to analyze a large travel agent database solution that had been developed overseas. There were performance issues, as always.... By examining the `sysindexes` table, I determined that out of the 700-plus tables in the database, only 60 percent of them had primary keys defined.

Of course, the client had been trying to implement transactional replication, which relies upon the primary key being defined. Without a primary key, SQL Server can replicate tables only via a snapshot, which can be particularly slow and expensive. One of my recommendations was to contact the developers and ask them to provide scripts that would create the "missing" primary keys.

So in this particular instance, a primary key constraint not only had an impact on entity integrity but also on performance. You should always have a good reason not to implement a primary key constraint. A very, very, very good reason!

Designing Entity Relationships

Relationships…I've implemented many. "The fundamental assumption of the relational database model is that all data is represented as mathematical *n*-ary relations, an *n*-ary relation being a subset of the Cartesian product of *n* sets." What the…???

Simply put, relationships naturally come about as a consequence of the normalization process and the resultant multiple entities. It is critical that you enforce and implement these relationships correctly in any SQL Server database solution. Developers, DBAs, and users are always primarily concerned about performance. Although performance is important, data integrity and especially referential integrity should take precedence.

Maintaining Referential Integrity

Referential integrity ensures that the relationships between the primary keys (referenced table) and foreign keys (referencing table) are always maintained; otherwise, you end up with *orphaned records*. Consequently, a row in a referenced table cannot be deleted, and a primary key can't be changed, if a foreign key refers to that row.

In some database implementations, orphaned records are acceptable. For example, it might be unimportant who you sold a particular item to, but it is important to keep a record of the transaction for accounting or taxation purposes—think drug or arms dealers, or perhaps a database used to collect web traffic for a website where user session information quickly multiplies.

Another important decision point is whether you want SQL Server to automatically *cascade update* and delete operations. This feature has been available at the database engine level since SQL Server 2000.

A great way to lose referential integrity is to implement referential integrity at the application layer. It just plain doesn't work—trust me!

Real World Scenario

IT in ET

In 2001 I was contracted by the United Nations to analyze a civil registry system for a developing nation that had been developed by overseas developers. (Wow! I got *developing*, *developed*, and *developers* in the one sentence. In any case, it was no laughing matter.)

The developers had implemented what little referential integrity there was in the application layer. Needless to say, the database had substantial referential integrity loss. This resulted because of a poor application design, lack of database constraints, and no security, which allowed users to modify the data directly. The end result translated to a potential waste of money in excess of $300,000 USD, a system that was "almost useless," and a mammoth data cleansing undertaking that took me months to perform, with no guarantee of 100 percent success.

All of this could easily have been avoided by using a foreign key constraint with cascading actions.

So when deciding on what technique you should use for referential integrity, you have two options at the database engine level in SQL Server 2005: constraints or programmatic objects.

Maintaining Referential Integrity Using Foreign Key Constraints

Since SQL Server 2000, the database engine has had the capability to automatically cascade update and delete operations. Consequently, given the ease of implementation and the performance, you should generally implement referential integrity between two tables through a foreign key constraint.

Another advantage of using a foreign key constraint over other means is that various database modeling and reporting tools can automatically determine the relationships, instead of trying to determine relationships through naming conventions or indexes.

The partial syntax for creating a foreign key constraint is as follows:

```
ALTER TABLE [database_name .[ schema_name ] .
| schema_name . ] table_name
[ WITH { CHECK | NOCHECK } ]
ADD CONSTRAINT constraint_name
```

```
FOREIGN KEY (column_name)
REFERENCES [ schema_name . ] referenced_table_name
[ ( ref_column ) ]
[ ON DELETE { NO ACTION | CASCADE | SET NULL |
 SET DEFAULT } ]
[ ON UPDATE { NO ACTION | CASCADE | SET NULL |
 SET DEFAULT } ]
```

If you decided to create a foreign key on the [CustomerNumber] column [Orders] table that references the [Customer] table, you would execute the following T-SQL code:

```
ALTER TABLE [Orders]
ADD CONSTRAINT [FK_Orders(CustomerNumber)]
FOREIGN KEY (CustomerNumber)
REFERENCES [Customer](CustomerNumber)
ON UPDATE CASCADE
ON DELETE CASCADE ;
```

Notice that this cascades both delete and update operations in this particular case. This decision is purely based on your business requirements.

A common mistake I see being made time and time again is a lack of indexes on foreign key columns. Unlike with the primary key constraint, SQL Server 2005 does not create any index on the foreign key column. Considering, as discussed, that join operations can be expensive for the database engine and can run frequently in most SQL Server solutions, it is highly recommended that you index your foreign keys. The real art here is to determine the best type of index to use!

One major limitation with constraints is that they apply only within a database. So if your referential integrity requirements are between two tables that reside in different databases, you cannot use a foreign key constraint. You will have to implement such a cascading action through a DML trigger. Another major limitation of the foreign key constraint is that it does not support foreign keys with multiple cascade paths and cycles. Likewise, you will most likely have to implement cascading actions in relationships that are potentially cyclic through a DML trigger.

Maintaining Referential Integrity Using Procedural Code

In certain scenarios, as highlighted previously, you will not be able to implement referential integrity through a foreign key constraint. Instead, you will have to take advantage of the programmable objects supported by SQL Sever 2005.

A DML trigger is ideal for maintaining complex referential integrity requirements that cannot be met by a foreign key constraint. When writing such triggers, your developers should be conscious that the triggers will be firing for all update, insert, and delete operations, so performance is paramount. Make sure they implement the trigger efficiently!

SQL Server 2005 supports a number of triggers, which you will examine in Chapter 2. At this stage, it is sufficient to concentrate on how to maintain referential integrity through a DML after trigger.

The partial syntax for creating a DML after trigger is as follows:

```
CREATE TRIGGER [ schema_name . ]trigger_name
ON { table | view }
[ WITH <dml_trigger_option> [ ,...n ] ]
{ FOR | AFTER | INSTEAD OF }
{ [ INSERT ] [ , ] [ UPDATE ] [ , ] [ DELETE ] }
[ WITH APPEND ]
[ NOT FOR REPLICATION ]
AS { sql_statement [ ; ] [ ,...n ] |
EXTERNAL NAME <method specifier [ ; ] > }

<dml_trigger_option> ::=
  [ ENCRYPTION ]
  [ EXECUTE AS Clause ]
```

So if you had to cascade a delete operation between the [Customer] table and a related [Orders] table that happens to be in a separate database (called [SalesDB] in this instance), you would write the following T-SQL code:

```
CREATE TRIGGER [trg_Customer_CascadeDelete]
ON [Customer]
AFTER DELETE
AS
DELETE [SalesDB].[dbo].[Orders]
WHERE [CustomerNumber]
IN (SELECT [CustomerNumber] FROM inserted);
GO
```

Summary

The first technique you learned in this chapter was how to create a logical database design through the process of normalization. You learned the importance of 3NF in your initial database design. I then introduced the benefit of denormalization, which brings redundancy into your logical database. I discussed the trade-offs that denormalized data takes extra space and needs to be maintained.

You then examined the issues you face when choosing the appropriate data types for the attributes of your entities. You learned the various data types natively supported by SQL Server 2005 and examined the new XML and CLR user-defined data types in detail.

Next, I discussed domain integrity as well as the different SQL Server objects that help maintain the domain of a particular attribute.

You then learned how to physically implement your entities by using the CREATE TABLE statement. A further explanation of how SQL Server stores data rows physically clarified the impact your database design has on minimizing the space wasted on the SQL Server data pages.

I then discussed the importance of entity integrity and talked about how to implement it primarily via a primary key constraint.

You then learned the importance of referential integrity in a relational database and how to implement it depending on your requirements.

Exam Essentials

Understand 3NF. You should understand the process of normalization and be able to normalize a set of entities to 3NF.

Know how to appropriately use the XML data type. Make sure you know how SQL Server 2005 implements the XML data type and where it is appropriate to use the XML data type in your database design.

Understand the syntax for creating tables. SQL Server 2005 supports a rich number of options when creating tables. Ensure you understand the various options and how to use them.

Know the different ways of implementing domain integrity. You can implement domain integrity using different techniques. Make sure you know which technique is appropriate for the given requirements.

Know the different ways of implementing referential integrity. You can employ a number of mechanisms to achieve referential integrity. Make sure you understand the different mechanisms and when to choose the correct implementation.

Review Questions

1. You are designing the [Product] table for a large warehouse database that will stock more than 100,000 different product IDs, called SKUs. Performance is paramount in this table. You have the possibility of amalgamating with other warehouse databases in the future, so you have decided to implement a surrogate key. The table should keep track of [SKU], [ProductName], and other related fields. What T-SQL commands should you run? (Choose all that apply.)

A. Run:

```
CREATE TABLE [Product] (
SKU VARCHAR(10) NOT NULL,
Product VARCHAR(20) NOT NULL,
...
)
```

B. Run:

```
ALTER TABLE [Product]
ADD CONSTRAINT [UQ_Product(SKU)]
UNIQUE (SKU)
```

C. Run:

```
ALTER TABLE [Product]
ADD CONSTRAINT [PK_Product(ProductId)]
PRIMARY KEY (ProductId)
```

D. Run:

```
CREATE TABLE [Product] (
ProductID INT IDENTITY (1,1),
SKU VARCHAR(10) NOT NULL,
Product VARCHAR(20) NOT NULL,
...
)
```

E. Run:

```
CREATE TABLE [Product] (
ProductID UNIQUEIDENTIFIER DEFAULT NEWID(),
SKU VARCHAR(10) NOT NULL,
Product VARCHAR(20) NOT NULL,
...
)
```

F. Run:

```
ALTER TABLE [Product]
ADD CONSTRAINT [PK_Product(SKU)]
PRIMARY KEY (SKU)
```

G. Run:

```
ALTER TABLE [Product]
ADD CONSTRAINT [UQ_Product(ProductId)]
UNIQUE (ProductId)
```

2. You are designing a database for the electricity market in Australia. The market has a [DispatchInterval] field that will be used as a primary key and that uses the following mask: YYYYMMDD###. The last three digits correspond to five-minute intervals, so there are 288 values a day. What data type should you use?

A. `INT`

B. `BIGINT`

C. `NUMERIC(11,0)`

D. `FLOAT`

3. You are developing a civil registry database for the Divided Nations nongovernmental organization that needs to keep track of demographic information about the population for the past 200 years. The main table contains millions of rows, so performance is critical. What data type should you use for the [DateOfBirth] field?

A. Use a `CHAR(10)` data type to store dates using the DD-MM-YYYY format.

B. Use the `SMALLDATE` data type.

C. Use the `DATETIME` data type.

D. Use a CLR user-defined data type.

4. What objects in SQL Server 20005 can be used to maintain domain integrity? (Choose all that apply.)

A. `PRIMARY KEY CONSTRAINT`

B. `FOREIGN KEY CONSTRAINT`

C. `CHECK CONSTRAINT`

D. Data type

E. DML trigger

F. `NONCLUSTERED INDEX`

G. Nullability (`NULL`/`NOT NULL`)

5. You are designing a Sales database where the tables contain millions of records and there are thousands of users, so performance is paramount. One of the most common reports will be returning Store, Product, and SUM(Quantity). You decide to denormalize the following database schema to eliminate the need for join operations:

```
Product(ProductId, Product, Price)
Store(StoreId, Store, Address, PostCode, State, Country)
Sales(SalesId, StoreId, SalesDate, Status)
SalesDetail(SalesId, ProductId, Quantity)
```

What should you do?

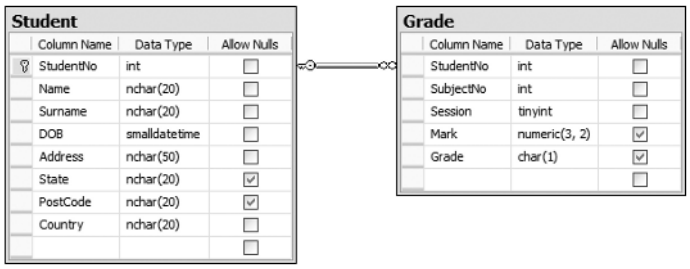
A. Add the [Product] column to the [SalesDetail] table.

B. Add the [Store] column to the [Sales] table.

C. Add the [Store] column to the [SalesDetail] table.

D. Add the [Product] column to the [Sales] table.

6. You are designing a database for a university and want to ensure that query performance will be optimal between the following two tables that will be frequently joined:



What should you do to improve query performance?

A. Create an index on the [Student].[StudentNo] column.

B. Create a unique constraint on the [Student].[StudentNo] column.

C. Create an index on the [Grade].[StudentNo] column.

D. Create a unique constraint on the [Grade].[StudentNo] column.

7. You are designing a table for a large multinational weapons manufacturer that needs to store the Coordinated Universal Time (UTC) date and time, down to the millisecond, of when a missile is launched. What is the easiest data type to implement for such a field?

A. Use a CLR user-defined data type.

B. Use the `DATETIME` data type.

C. Use a T-SQL user-defined data type.

D. Use a `BINARY` data type.

8. You are designing a large sales database where performance is critical. You know that your sales personnel will be running queries constantly throughout the working hours to find out the year-to-date sales for a particular product. They want the data returned to be up-to-date. You have more than 690,000 products and generate approximately 10,000 invoices daily. Your database design is as follows:

```
CREATE TABLE [Product] (
      [ProductId] SMALLINT NOT NULL,
      [Product] VARCHAR(20) NOT NULL,
      [Price]     MONEY NOT NULL,
      [StockLevel] INT NOT NULL
)

CREATE TABLE [Invoice] (
      [InvoiceNumber]   INT NOT NULL,
      [InvoiceDate] DATETIME NOT NULL,
      [CustomerId] INT NOT NULL
)

CREATE TABLE [InvoiceDetails] (
      [InvoiceNumber]   INT NOT NULL,
      [ProductId] SMALLINT NOT NULL,
      [SalesQuantity]   TINYINT NOT NULL,
      [SalesPrice] MONEY NOT NULL
)
```

What two actions should you perform?

A. Use `ALTER TABLE [Invoice] ADD [YTDSales] INT`.

B. Use `ALTER TABLE [Product] ADD [YTDSales] INT`.

C. Maintain [YTDSales] using a DML trigger.

D. Maintain [YTDSales] using a T-SQL script that is scheduled to run after-hours.

9. You need to decide which data type to use to store legal documents that natively have an XML format. The legal documents will need to be able to be searched. Government regulations stipulate that the legal documents cannot be altered by any party once submitted. What data type should you use?

A. Use a `CHAR(8000)` data type.

B. Use a `TEXT` data type.

C. Use a `VARCHAR(MAX)` data type.

D. Use an `XML` data type.

10. You need to ensure that you do not lose referential integrity between these two tables that have a one-to-one relationship and that SQL Server automatically cascades update but not delete operations:

```
CREATE TABLE [Product] (
      [WareHouseId] TINYINT NOT NULL,
      [ProductId] INT NOT NULL,
      [ProductName] VARCHAR(20) NOT NULL,
      [Price] NCHAR(10) NOT NULL,
      [StockLevel] TINYINT NOT NULL
)
GO
ALTER TABLE [Product]
ADD CONSTRAINT [PK_Product]
PRIMARY KEY (WareHouseId, ProductId)

CREATE TABLE [ProductDetails] (
      [WarehouseId[ TINYINT NULL,
      [ProductId] INT NULL,
      [ProductDescription] XML NULL,
      [ProductPhoto] VARBINARY(MAX) NULL
)
GO
```

What should you execute?

A. Execute:

```
ALTER TABLE [ProductDetails
ADD CONSTRAINT [FK_ProductDetails_Product]
FOREIGN KEY (ProductId)
REFERENCES Product (ProductId)
ON UPDATE CASCADE
ON DELETE CASCADE
```

B. Execute:

```
ALTER TABLE [ProductDetails]
ADD CONSTRAINT [FK_ProductDetails_Product]
FOREIGN KEY (WarehouseId, ProductId)
REFERENCES Product (WareHouseId, ProductId)
ON UPDATE CASCADE
ON DELETE CASCADE
```

C. Execute:

```
ALTER TABLE [ProductDetails]
ADD CONSTRAINT [FK_ProductDetails_Product]
FOREIGN KEY (WarehouseId, ProductId)
REFERENCES Product (WareHouseId, ProductId)
ON UPDATE CASCADE
ON DELETE NO ACTION
```

D. Execute:

```
ALTER TABLE [ProductDetails]
ADD CONSTRAINT [FK_ProductDetails_Product]
FOREIGN KEY (ProductId)
REFERENCES Product (ProductId)
ON UPDATE CASCADE
ON DELETE NO ACTION
```

11. What normal form should you typically aim for when designing the initial relational database design?

A. First normal form (1NF)

B. Second normal form (2NF)

C. Third normal form (3NF)

D. Fourth normal form (4NF)

E. Fifth normal form (5NF)

12. Whenever a programmer is fired from the company and their record deleted from the [Employee] table of the [HumanResources] database, you need to ensure that all related records are deleted from child tables in the [HumanResources] database. What is the quickest way to achieve this?

A. Use a check constraint.

B. Use a rule.

C. Use a DML trigger that will automatically delete the related records.

D. Use a foreign key constraint with the cascade delete option to automatically cascade delete operations.

13. What objects in SQL Server 2005 can be used to main entity integrity? (Choose all that apply.)

A. PRIMARY KEY CONSTRAINT

B. FOREIGN KEY CONSTRAINT

C. CHECK CONSTRAINT

D. UNIQUE CONSTRAINT

E. DML trigger

14. You need to decide which data type to use to store technical documents that natively have an XML format. The technical documents will often be searched, so performance is important. The schema of the technical documents is expected to change over time. What data type should you use?

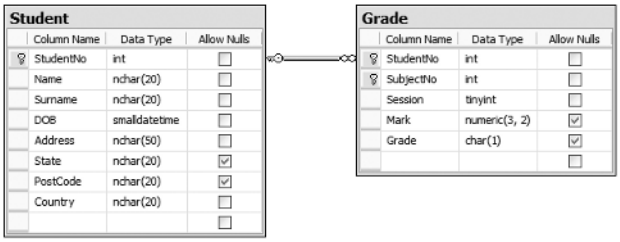
A. Use a VARCHAR(MAX) data type.

B. Use a CHAR(8000) data type.

C. Use a TEXT data type.

D. Use an XML data type.

15. You have designed your logical database model as shown here. What T-SQL statements should you execute to create the tables? (Choose all that apply in the correct order.)



A. Execute:

```
ALTER TABLE [Grade]
ADD CONSTRAINT [FK_Grade_Student]
FOREIGN KEY(SubjectNo)
REFERENCES [Student] (StudentNo)
```

B. Execute:

```
CREATE TABLE [Student] (
    [StudentNo] INT NOT NULL,
    [Name] NCHAR(20) NOT NULL,
    [Surname] NCHAR(20)NOT NULL,
    [DOB] SMALLDATETIME NOT NULL,
    [Address] NCHAR(50) NOT NULL,
    [State] NCHAR(20) NOT NULL,
    [PostCode] NCHAR(20) NOT NULL,
    [Country] NCHAR(20) NOT NULL
)
```

C. Execute:

```
CREATE TABLE [Student] (
    [StudentNo] INT NOT NULL,
    [Name] NCHAR(20) NOT NULL,
    [Surname] NCHAR(20)NOT NULL,
    [DOB] SMALLDATETIME NOT NULL,
    [Address] NCHAR(50) NOT NULL,
    [State] NCHAR(20) NULL,
    [PostCode] NCHAR(20) NULL,
    [Country] NCHAR(20) NOT NULL
)
```

D. Execute:

```
CREATE TABLE [Grade] (
    [StudentNo] INT ,
    [SubjectNo] INT ,
    [Session] TINYINT ,
    [Mark] NUMERIC(3, 2),
    [Grade] CHAR (1)
)
```

E. Execute:

```
ALTER TABLE [Student]
ADD CONSTRAINT [UQ_Student]
UNIQUE (StudentNo)
```

F. Execute:

```
ALTER TABLE [Grade]
ADD CONSTRAINT [PK_Grade]
PRIMARY KEY (StudentNo, SubjectNo)
```

G. Execute:

```
CREATE TABLE [Grade] (
    [StudentNo] INT NOT NULL,
    [SubjectNo] INT NOT NULL,
    [Session] TINYINT NOT NULL,
    [Mark] NUMERIC(3, 2) NULL,
    [Grade] CHAR (1) NULL
)
```

H. Execute:

```
ALTER TABLE [Grade]
ADD CONSTRAINT [PK_Grade]
PRIMARY KEY (StudentNo)
```

I. Execute:

```
ALTER TABLE [Student]
ADD CONSTRAINT [PK_Student]
PRIMARY KEY (StudentNo)
```

J. Execute:

```
ALTER TABLE [Grade]
ADD CONSTRAINT [PK_Grade]
PRIMARY KEY (SubjectNo)
```

16. You want to ensure that the [Name], [LastName], and [PhoneNumber] fields are implemented in a standard way across all new SQL Server database solutions in your enterprise. What should you do? (Choose two.)

A. Use a T-SQL user-defined data type for the [Name], [LastName], and [PhoneNumber] fields.

B. Create the user-defined types in the **tempdb** database.

C. Use a CLR user-defined data type for the [Name], [LastName], and [PhoneNumber] fields.

D. Create the user-defined data types in the model database.

17. You are capacity planning your database design and need to determine how much space will be required for the [ProductDescription] table. You predict that the table will contain 6,900,000 products. The table schema is as follows:

```
CREATE TABLE [ProductDescription] (
    [ProductId] INT,
    [EnglishPrice] SMALLMONEY,
    [EnglishDescription] NCHAR(450),
    [RussianPrice] SMALLMONEY,
    [RussianDescription] NCHAR(450),
    [GermanPrice] SMALLMONEY,
    [GermanDescription] NCHAR(450)
)
```

How much space will the table consume?

A. 55,200,000KB

B. 27,600,000KB

C. 18,400,000KB

D. 13,800,000KB

18. You are designing a sales database and want ensure that a salesperson cannot issue an invoice for more products than currently in stock. Your database design is as follows:

```
CREATE TABLE [Product] (
    [ProductId] SMALLINT NOT NULL,
    [Product] VARCHAR(20) NOT NULL,
    [Price]     MONEY NOT NULL,
    [StockLevel] INT NOT NULL
)

CREATE TABLE [Invoice] (
    [InvoiceNumber]   INT NOT NULL,
    [InvoiceDate] DATETIME NOT NULL,
    [CustomerId] INT NOT NULL
)

CREATE TABLE [InvoiceDetails] (
    [InvoiceNumber]   INT NOT NULL,
    [ProductId] SMALLINT NOT NULL,
    [SalesQuantity]   TINYINT NOT NULL,
    [SalesPrice] MONEY NOT NULL
)
```

What should you use to ensure that salespeople do not issue an invoice for more products than currently in stock?

A. Use a check constraint.

B. Use a DML trigger.

C. Use a foreign key constraint.

D. Use a primary key constraint.

19. You need to ensure that whenever an employee is deleted from the [Employee] table of the [HumanResources] database, all related records are deleted from the [Marketing] database. What is the quickest way to achieve this?

A. Use a check constraint.

B. Use a DML trigger that will automatically delete the related records.

C. Use a foreign key constraint with the cascade delete option to automatically cascade delete operations.

D. Use a rule.

20. You are helping to design a database for a Swedish bank called Darrenbank. Tobias, the chief information officer (CIO), explains they require a field called [CreditRisk] in the [Customer] table. This [CreditRisk] field is based on a complex calculation that involves 69 fields from the [Customer] table. The [CreditRisk] field will generate a value ranging from 1 to 5. The [Customer] has more than 200 fields and contains more than 307,000 records. Tobias has indicated that the data changes "infrequently" and that "performance is important." How do you implement the [CreditRisk] field?

A. Choose all that apply.

B. Create an index on the [CreditRisk] field.

C. Create the [CreditRisk] field.

D. Create the [CreditRisk] field as a computed field.

E. Create the [CreditRisk] field as a persisted, computed field.

F. Create a DML trigger to maintain the [CreditRisk] field.

Answers to Review Questions

1. D, C, B. Option D correctly implements an efficient surrogate key. Option C correctly implements the surrogate key, and Option B correctly implements the natural key. Option E is inappropriate because it uses the 16-byte `UNIQUEIDENTIFIER`, which is generally a poor choice for a primary key because it slows down performance. Option A is incorrect because it does not implement a surrogate key. If the database needs to be amalgamated in the future, it is a simple matter of altering the table to make a compound primary key to facilitate amalgamation.

2. B. The `BIGINT` data type is the smallest data type that can hold [DispatchInterval]. The INT data type is not large enough. The `NUMERIC(11,0)` data type consumes a byte more than `BIGINT`. The `FLOAT` data type is imprecise and is not inherently a cardinal.

3. C. The `DATETIME` data type will be able to accommodate your domain. The `SMALLDATE` data type will not accommodate your domain because it does not allow dates before 1900. The `CHAR(10)` data type will consume two more bytes than the `DATETIME` data type, which will impact performance. A CLR user-defined data type will not perform as well as a native SQL Server data type in this instance.

4. B, C, D, E, G. Foreign key constraints, check constraints, data types, nullability, and DML triggers all can help maintain domain integrity, or the set of values that are valid for a column. Primary key constraints help maintain entity integrity. Nonclustered indexes generally improve performance.

5. C. By adding the [Store] column to the [SalesDetail] table, you have eliminated the need for the query to access the [Sales] and [Store] tables.

6. C. Creating an index on the [Grade].[StudentNo] column will improve join performance because SQL Server 2005 does not automatically create indexes on foreign keys. Creating an index on the [Student].[StudentNo] field will worsen performance because there is already an index on that field via the primary key constraint, which means more indexes for SQL Server 2005 to maintain. creating a unique constraint on the [Student].[StudentNo] or [Grade].[StudentNo] column will prevent valid data from being inserted into the [Grade] table.

7. A. None of the SQL Server–supplied data types stores the date, time, and time zone information natively. Although a `BINARY` data type could in fact potentially store such information, it would be difficult to implement and use. A CLR user-defined data type would easily be able to store this customized data.

8. B, C. You should add the [YTDSales] column to the [Product] table because there will be only one instance of maintaining the denormalized data and no need for a join operation. Although maintaining performance is critical, the sales personnel needs the information to be accurate. Performance should not be impacted because you are changing only a small percentage of rows throughout the day on a static table. The DML trigger would have to increment only the existing [YTDSales] value, creating a minimal impact on performance.

9. C. Since the document cannot be changed, you cannot use the XML data type because it does not guarantee that the inserted XML document will be the same as the retrieved XML document. A `CHAR(8000)` data type takes up too much needless space, and the table will grow beyond SQL Server's page limit. The `TEXT` data type is being deprecated and should not be used.

10. C. Option C correctly implements referential integrity using the compound foreign key constraint and cascades the update operation only. Options A and D do not implement the compound foreign key correctly. Option B does not cascade the operations correctly.

11. C. When designing out initial database design, you should typically aim for 3NF.

12. D. You should always use a foreign key constraint over any other means to maintain referential integrity if possible because they are the quickest and easiest to implement. You could use a DML trigger, but that would involved more coding, would be slower, and would require more skills from your developers.

13. A, D, E. You can use a primary key constraint, unique constraint, and DML trigger to enforce uniqueness and therefore entity integrity. Foreign key constraints enforce mainly referential integrity; check constraints can maintain domain integrity.

14. D. The XML data type will give you the most flexibility, allowing the documents to be indexed to improve query performance and facilitate a changing schema. None of the other data types supports any schema natively, not any indexing within the SQL Server 2005 engine.

15. C, G, I, F, A. Options C, G, I, F, and A correctly implement the database schema. Options B and D do not correctly implement the NULLs. Options H and J do not correctly implement the composite primary key. Option E implements a unique constraint, not a primary key constraint.

16. A, D. T-SQL user-defined data types were designed for implementing such "standards" in multiple databases. By creating them in the model database, you will ensure that all future user databases will have the new user-defined data types. You should use CLR user-defined data types for much more complex requirements. Use the **tempdb** database only as a temporary global workspace.

17. B. A data row will consume 2,716 bytes (4 + 4 + 900 + 4 + 900 + 4 + 900). A page has 8,092 free space. Therefore, a page can hold only two rows (8,092 / 2716 = 2.98). Consequently, the table will consume 3,450,000 pages (6,900,000 / 2) or 27,600,000KB (3,450,000 * 8KB).

18. B. You can write a DML trigger to look up a product's [StockLevel] from the [Product] table and ensure that the [SalesQuantity] being entered is less than that amount before allowing a record to be inserted into [InvoiceDetails]. A check constraint cannot look up a value in another table. A foreign key constraint can look up a value only in a related table. A primary key constraint maintains only entity integrity.

19. B. You can easily write a DML trigger to maintain cross-database referential integrity. A constraint works only within a database, so a foreign key constraint would not be appropriate in this particular instance.

20. D. Creating a persisted, computed field finds the best balance between improving performance without creating overhead on the system. Options B and E will be slower than creating a persisted computed column. There is no point in creating an index (Option A) because the data is not selective enough, even if it is searched on. Just creating a computed field (Option C) will not improve performance because each time the field is retrieved, a complex calculation will have to be done. Considering the extra space required (`TINYINT`), it is better to persist the field, and Tobias has indicated, in the subtle ways that Swedes do, that the data changes "infrequently."