SQL Queries

Succinctly®

by Nick Harrison

Syncfusion | Technology Resource Portal

SQL Queries Succinctly

By
Nick Harrison

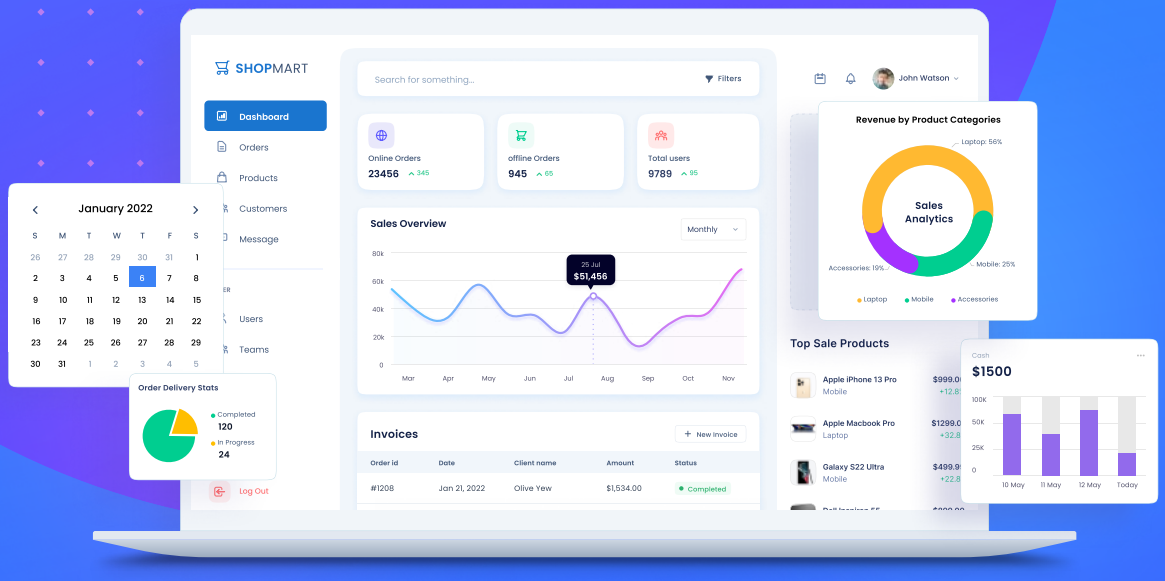
Foreword by Daniel Jebaraj

2501 Aerial Center Parkway
Suite 200
Morrisville, NC 27560
USA

Technical Reviewer: James McCaffrey
Copy Editor: Courtney Wright
Acquisitions Coordinator: Hillary Bowling, online marketing manager, Syncfusion, Inc.
Proofreader: Jacqueline Bieringer, content producer, Syncfusion, Inc.

THE WORLD'S BEST
UI COMPONENT SUITE
FOR BUILDING
POWERFUL APPS

4.6 out of 5 stars



GET YOUR FREE .NET AND JAVASCRIPT UI COMPONENTS

syncfusion.com/communitylicense

- 1,700+ components for mobile, web, and desktop platforms
- Support within 24 hours on all business days
- Uncompromising quality
- Hassle-free licensing
- 28000+ customers
- 20+ years in business

Trusted by the world's leading companies

Apple IBM SIEMENS intel VISA

Syncfusion

Table of Contents

The Story behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to “enable AJAX support with one click,” or “turn the moon to cheese!”

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and “Like” us on Facebook to help us spread the word about the *Succinctly* series!

About the Author

Nick Harrison has more than 20 years of development experience and has worked with .NET since the first Community Technical Preview (CTP) went golden. He believes that .NET is a wonderful development environment that gets better with each update.

Nick also stays active in the local development community as a frequent speaker at local user groups and code camps. He is an author for the technical journal Simple-Talk and an occasional blogger on the blog community Geeks with Blogs.

In 2007, he met the true love of his life and was fortunate enough to start a family with Tracy along with Morgan Faye. To this day, they remain a profound source of joy and a strong counterbalance to the chaos of daily life.

Chapter 1 Introduction

The role of SQL

SQL is the language of data. Data is the language of business because every business runs off data. We can say that SQL is the language of business, or at least it is the intermediary language for those of us who straddle the line between technology and business.

It's hard to imagine a business application without a database. Whether you are dealing with Online Transaction Processing (OLTP), Online Analytical Processing (OLAP), or ad hoc queries, SQL is the key to getting data into and out of the database. To the uninitiated or even the casual user, SQL can seem like magic. SQL is a fourth-generation language (4GL). This means that, unlike most programming languages, we don't tell the database how to do what needs to be done. We describe what needs to be done and leave the database to decide how best to do what we requested.

SQL allows us to describe the data we need. The database determines which indexes to use, how to join the data in the various tables involved, how to read from the disks, how to isolate dirty data, and so on.

What will be covered

Properly initiated, we can work this magic ourselves.

We will cover how to get data into the database, update records that are already there, and delete data that is no longer needed. We will slice and dice the data any way you can imagine. We will see how to follow relationships across tables and how to apply filters to find the relevant data. We will use built-in aggregate functions to summarize data and look at solving common problems with date and time values, as well as various ways to navigate hierarchical data, such as org charts, sales hierarchies, etc.

Often the hardest part of crafting a query is understanding the data model being used, so we will finish by exploring the data dictionary. This holds all the data that the database tracks about the data being stored. We will use these views to get details about every column in every table in any database. This can substantially shorten the learning curve for learning a new data model.

The focus for this book will be ANSI SQL, but many vendors have varying levels of support for the ANSI standards. All of the samples given in this book have been tested against SQL Server 2012. Most of the examples will work with SQL Server 2008R2. The few cases that will require a new version are noted as the sample is explained.

You can download a free version of SQL Server 2016 Express here.

What won't be covered

SQL in general is a big topic. Here we will focus on using the language itself. Several topics on the edge will, out of necessity, not be covered, even though each of these topics is very important on its own.

We will ignore most issues of database administration. We will skip over hardware and infrastructure concerns, and ignore the critical issues of redundancy and fault tolerance. We will mention performance only briefly, to showcase how a query could be rearranged to improve performance and discuss some common problems that can cause the database to work harder than it should in order to get the data we are interested in.

Security is everyone's concern, but in this book, we will leave this concern on the wayside. Just know that it's everyone's responsibility to protect the data. Defense in depth requires validations and appropriate constraints in the prompts on the user interface for business applications, validations in the business logic that cannot be bypassed in the user interface, and appropriate constraints in the database. These are important, but will not be directly addressed here.

Database constraints and checks are the domain of data modeling, which we will also be skipping. In each of our discussions, we will be working with a data model, but we won't delve deeply into how the data model was created or which alternative structures could have been used. In many cases, we will work with a less-than-optimal data model, acknowledging that we often don't get to work with an ideal data model. The reality is that we are often stuck with the database that we have, which has often grown and mutated beyond its original design.

Note: We will stress the importance of constraints and checks as appropriate, but still accept the reality that we mostly have to live with the database we have, and that we rarely get to design a new database from scratch.

Summary

In this introductory chapter we have looked at the nature of SQL and its importance in modern business applications. We have seen an overview of what we will cover in this book, as well as a partial list of some of the topics we will be skipping. Even though we are skipping these topics, they are still important. I urge you to consider this your first step in learning SQL and follow up on some of these side topics that we're skipping.

Now let's start with some of the simplest forms the standard SQL operations can take.

Chapter 2 Basic CRUD

CRUD is a common acronym that refers to the basic data manipulation language statements **`CREATE`**, **`READ`**, **`UPDATE`**, and **`DELETE`**; or **`INSERT`**, **`SELECT`**, **`UPDATE`**, and **`DELETE`** (but ISUD does not sound nearly as cool).

We will look at each of these types of statements in turn. In this chapter we will go over the basics for working with each of these statements. Chapter 3 will cover some more advanced scenarios with these basic statements. The rest of this book will focus on all the variations and complexities associated with reading or selecting data. After all, it's with **`SELECT`** statements that we slice and dice our data.

For our discussion of basic CRUD statements, we will often use the simple data model shown in Figure 2-1 relating employees and departments:

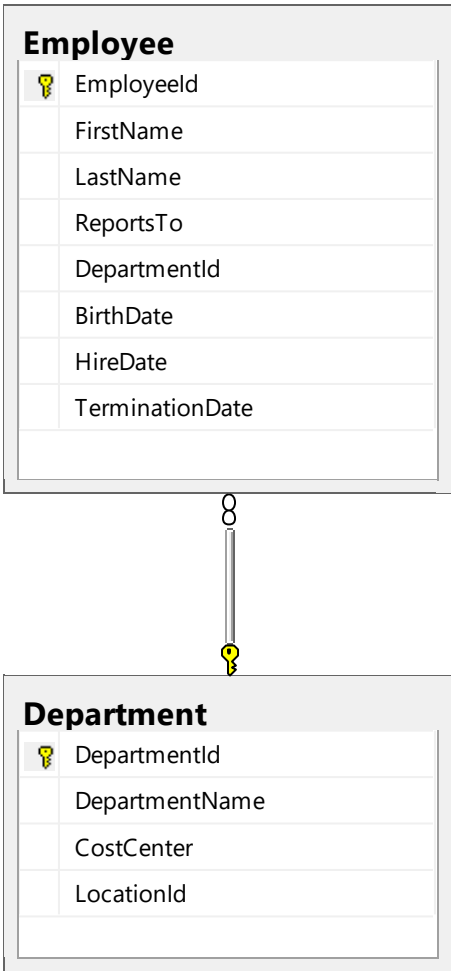


Figure 2-1: Our simple data model

The SQL to generate these tables is shown in Code Listing 2-1.

Code Listing 2-1: Creating the Employee and Department tables

```
CREATE TABLE [dbo].[Employee](
      [EmployeeId] [int] IDENTITY(1,1) NOT NULL,
      [FirstName] [varchar](50) NOT NULL,
      [LastName] [varchar](50) NOT NULL,
      [ReportsTo] [int] NULL,
      [DepartmentId] [int] NULL,
      [BirthDate] [datetime] NULL,
      [HireDate] [datetime] NOT NULL,
      [TerminationDate] [datetime] NULL,
 CONSTRAINT [PK_Employee] PRIMARY KEY CLUSTERED ([EmployeeId] ASC)
)

CREATE TABLE [dbo].[Department](
      [DepartmentId] [int] IDENTITY(1,1) NOT NULL,
      [DepartmentName] [varchar](50) NOT NULL,
      [CostCenter] [varchar](50) NULL,
      [LocationId] [int] NULL,
 CONSTRAINT [PK_Department] PRIMARY KEY CLUSTERED ([DepartmentId] ASC)
)
ALTER TABLE [dbo].[Employee]  WITH CHECK ADD  CONSTRAINT
[FK_Employee_Department] FOREIGN KEY([DepartmentId])
REFERENCES [dbo].[Department] ([DepartmentId])
```

Insert statements

We get data into the database with **`INSERT`** statements. In general, these will be handled with business logic in your application code. We generally don't do a lot of ad hoc inserting into the database. Business logic in your application may handle referential integrity and other constraints that may not be fully defined in the database. You may also have auditing and tracking mechanisms in your application that should not be bypassed.

Business applications are really SQL-level firewalls controlling access to the database. Business logic can implement rules that cannot easily be defined with database constraints, and can define process flows that may change over time or with different business areas, making them impractical to embed in the database structures. Regardless of the reasons, you probably will have rules that need to be followed, but are implemented only in the application logic.

Tip: Ideally, all of these constraints would be baked into the database. In reality, there will be many constraints that the database will know nothing about and can do nothing to help ensure that they are adhered to.

If you find an **`INSERT`** statement that you need to run regularly, it will be in your best interests to get it properly integrated into your application code, and not rely on getting it right every time someone needs to run it manually.

With all the warnings out of the way, let's look at the basic **INSERT** statement; we will use the **Employee** table.

The general form for the **INSERT** statement is:

Code Listing 2-2: The basic INSERT statement

```
INSERT INTO dbo.Employee
        ( FirstName ,
          LastName ,
          ReportsTo ,
          DepartmentId ,
          BirthDate ,
          HireDate ,
          TerminationDate )
VALUES  ( '' , -- FirstName - varchar(50)
          '' , -- LastName - varchar(50)
          0 , -- ReportsTo - int
          1 , -- DepartmentId - int
          GETDATE() , -- BirthDate - datetime
          GETDATE() , -- HireDate - datetime
          GETDATE()  -- TerminationDate - datetime
          )
```

The key thing to get right here is making sure the column order in the **INSERT** clause matches the column order in the **VALUES** clause. If you skip a column in the **INSERT** clause, you must skip it in the **VALUES** clause as well. If a column is skipped, you will not be inserting a value into that column.

Note: Because we have defined a foreign key from the Employee table to the Department table, we need to give a valid Department value. We can leave the DepartmentId blank (with a NULL value), and it will still pass the referential constraints, but if we assign it a value, it must match a value in the Department table. If we want to ensure that we always get a valid value for the Department, and it is not left blank, we would need to mark the column as Not Null, as well as set the foreign key relationship.

Default values

If we don't specify the value for a column, it will get the default value for that column. If you don't specify a default value for a column and don't specify a value during the insert, the column will get a value of NULL.

Note: NULL is different from being left blank. For text fields like First and Last Name, you could specify the value as '' to leave it blank, but it still has a value. But if you don't specify a value and there are no default values associated with the column, the value will default to NULL.

Now let's look at some of the details for the **Employee** table definition. We might have a default department that new employees get added to. If this is the case, and we don't specify the department, the newly inserted record will be associated with that department. If you do specify a value for a column with a default value, the value you specify will take precedence.

To set the default value for the **DepartmentId** column to 1, we can define that column like this:

Code Listing 2-3: Specifying the default department

```
ALTER TABLE dbo.Employee ADD DEFAULT (1) FOR DepartmentId,
```

You may also want to default the **HireDate** to today's date:

Code Listing 2-4: Specifying the default hire date

```
ALTER TABLE dbo.Employee  ADD DEFAULT  (GETDATE()) FOR HireDate
```

Tip: The important thing to remember here is that even if you don't specify a value during the INSERT statement, the new record may have a value if there is a default value associated with the column.

Identity columns

You may have noticed that in our standard syntax, we did not mention the **EmployeeId** column. This is a special case for the default value. It's a common best practice to associate the primary key for a table with an identity column. This is how SQL Server handles keeping track of valid values, the next value, etc. We have a few important rules to bear in mind when dealing with identity columns:

- You can assign the **identity** property to columns when they are created either as part of the initial table creation, or when the table is altered by adding a new column.
- Once a column has been added, you cannot alter it to add the **identity** property to it; it needs to be added when the table is created or the column is added.
- You can only have one identity column per table.
- You cannot explicitly set the value for an identity column. We will see the exception to this in a moment.
- You also cannot update the value for an identity column.

This helps ensure data integrity for this table and any table that depends on it, but it can make it harder to import data from an alternate source, such as copying data from one environment to another.

Note: DBAs have developed intricate strategies for dealing with the identity columns on tables in a replication environment. For the most part, you will not be

affected by these issues unless you are the DBA. We just need to be aware that many of these strategies will result in gaps in the identity values.

For various reasons, you may find that you have a legitimate need to insert a specific value into an identity column. Despite what we have said so far, it turns out that you can directly change this value, but you have to jump through some hoops to do so, and you need to be careful because you could easily compromise data integrity or create future primary key violations.

Code Listing 2-5: Overriding the identity column during insert

```
SET IDENTITY_INSERT dbo.Employee ON;

INSERT  INTO dbo.Employee
        ( EmployeeId, FirstName, LastName )
VALUES  ( 200, 'Nick', 'Harrison' );


SET IDENTITY_INSERT dbo.Employee OFF;
```

Tip: Always make sure to explicitly turn the identity insert as soon as you have finished. Otherwise you could put your data integrity at risk.

Constraints

Constraints can be defined to limit the values that can go in a column. This helps ensure data integrity, but it can also cause confusion during inserts if you are not following the rules defined in the constraints.

Let's revisit the **`Employee`** table and add a constraint that says that the **`TerminationDate`** must be after the **`HireDate`**.

Code Listing 2-6: Constraining when we can terminate an employee

```
ALTER TABLE dbo.Employee  ADD CONSTRAINT ck_termination_date
CHECK ( TerminationDate > HireDate);
```

Now if we try the original **`INSERT`** statement that we saw earlier with the standard syntax, we will get an error, because an employee cannot be hired and terminated at the same time.

Code Listing 2-7: INSERT statement violating constraints

```
INSERT INTO dbo.Employee
        ( FirstName ,
          LastName ,
          ReportsTo ,
          DepartmentId ,
          BirthDate ,
```

```
          HireDate ,
          TerminationDate )
VALUES  ( 'New' , -- FirstName - varchar(50)
          'Employee' , -- LastName - varchar(50)
          0 , -- ReportsTo - int
          0 , -- DepartmentId - int
          GETDATE() , -- BirthDate - datetime
          GETDATE() , -- HireDate - datetime
          GETDATE()  -- TerminationDate - datetime
          )

Msg 547, Level 16, State 0, Line 1
The INSERT statement conflicted with the CHECK constraint
"ck_termination_date". The conflict occurred in database "Playground",
table "dbo.Employee".
The statement has been terminated.
```

Tip: If you are designing a database, define all reasonable constraints you can. This will ensure that validation rules cannot easily be violated. If you are using a pre-existing data model, be aware that many constraints will be missing, so you may have to work harder with application-based business logic to ensure data integrity.

Triggers

Triggers are bits of logic that will fire in the database without any direct intervention from you. They are like events for the database. For example, we can have a trigger for insert that will be called whenever a record is inserted into that table. This can be used to further support referential integrity, to set up auditing of changes to key tables, or to enforce business rules in the database.

Tip: Enforcing business rules through triggers in the database means that the business rule cannot be bypassed, but it is easy to overlook. When the rule changes, you may have trouble remembering that the rule was implemented in a trigger because they easily go unnoticed. Also, if multiple applications are using the same database, they may not all agree on these business rules, so their implementation would be best met in the application code.

Looking back at the **`Employee`** table we have been using, we may want to define an **`EmployeeAudit`** table that would track whenever anyone modifies data in this table. Triggers could be used to easily handle such tracking.

Start by creating a new table called **`EmployeeAudit`** with the same columns as the **`Employee`** table. For this new table, the **`EmployeeId`** will not be the primary key. It will also not be an identity column. Instead, we will add a new column called **`EmployeeAuditId`**, which will be the

primary key as well as the identity column. Let's also add a couple of new columns for **AuditAction** and **AuditDate**.

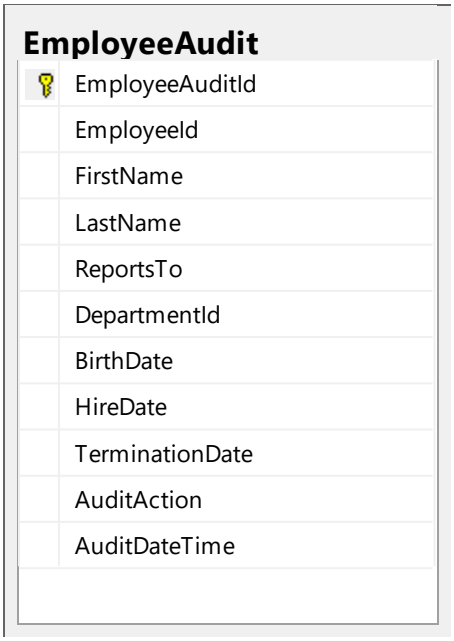


Figure 2-2: A simple audit table

Tip: Even though we want to define as many constraints as is practical in the Employee table, we will want to disable most of the constraints on the Audit table. This way we can log everything that happens to the Employee table, even if a constraint is temporarily disabled.

Code Listing 2-8: Creating the EmployeeAudit table

```
CREATE TABLE [dbo].[EmployeeAudit](
   [EmployeeAuditId] [int] IDENTITY(1,1) NOT NULL,
   [EmployeeId] [int] NOT NULL,
   [FirstName] [varchar](50) NOT NULL,
   [LastName] [varchar](50) NOT NULL,
   [ReportsTo] [int] NULL,
   [DepartmentId] [int] NULL,
   [BirthDate] [datetime] NULL,
   [HireDate] [datetime] NOT NULL,
   [TerminationDate] [datetime] NULL,
   [AuditAction] [char](1) NOT NULL,
   [AuditDateTime] [datetime] NOT NULL,
   CONSTRAINT [PK_EmployeeAudit_1] PRIMARY KEY CLUSTERED
([EmployeeAuditId] ASC)
)
```

Now that we have our **Audit** table in place, let's look at what the insert trigger for the **Employee** table might look like.

Code Listing 2-9: Trigger to audit inserts into the Employee table

```
CREATE TRIGGER trEmployeeInsert ON dbo.Employee
  FOR INSERT
AS
    INSERT  INTO dbo.EmployeeAudit
            ( EmployeeId ,
              FirstName ,
              LastName ,
              ReportsTo ,
              DepartmentId ,
              BirthDate ,
              HireDate ,
              TerminationDate ,
              AuditAction ,
              AuditDateTime )
            SELECT  Inserted.EmployeeId ,
                    Inserted.FirstName ,
                    Inserted.LastName ,
                    Inserted.ReportsTo ,
                    Inserted.DepartmentId ,
                    Inserted.BirthDate ,
                    Inserted.HireDate ,
                    Inserted.TerminationDate,
                    'I' ,  -- I is for insert
                    GETDATE()
            FROM    Inserted;
```

Note: We will explore the syntax for this insert statement more fully in the next chapter. The SQL INSERT statement combined with a SELECT statement is one technique you can use to support logging when the insert operation adds multiple records into a table.

The **`Inserted`** table mentioned in the **`Trigger`** definition is a memory-only table that SQL Server maintains for us. We cannot modify this table by adding an index or manipulating any of the data, but it can be very helpful in tracking what was inserted and taking appropriate action.

Now when we insert a record into the **`Employee`** table, we will get a copy of what was inserted in the **`EmployeeAudit`** table.

Code Listing 2-10: Insert triggers may change the number of rows affected by the INSERT statement

```
INSERT INTO dbo.Employee
        ( FirstName ,
          LastName ,
          ReportsTo ,
          DepartmentId ,
```

```
          BirthDate ,
          HireDate ,
          TerminationDate )
VALUES  ( 'New' , -- FirstName - varchar(50)
          'Employee' , -- LastName - varchar(50)
          0 , -- ReportsTo - int
         1 , -- DepartmentId - int
          GETDATE() , -- BirthDate - datetime
          GETDATE() , -- HireDate - datetime
          GETDATE() + 60  -- TerminationDate - datetime
          )


(1 row(s) affected)

(1 row(s) affected)
```

If you run this command in Management Studio, you will see that two rows were affected: the new record in the **`Employee`** table, and a new record in the **`EmployeeAudit`** table.

Note: With triggers, you may have a single INSERT statement affect more than one record, potentially records in different but related tables.

Let's now look at how to enforce business rules in a trigger. Suppose your company has a rule that all employees must be at least 18 years old. You could potentially implement this rule with a trigger like this:

Code Listing 2-11: Defining a trigger to enforce business logic

```
CREATE TRIGGER trEmployeeInsertVerifyAge ON dbo.Employee
    AFTER INSERT
AS
    IF EXISTS ( SELECT  *
                FROM    Inserted
                WHERE   DATEDIFF(YEAR, Inserted.BirthDate, GETDATE()) <
18 )
        BEGIN
            RAISERROR ('New hires must be at least 18 years old', 16,
10);
            ROLLBACK TRANSACTION;
            RETURN;
        END;
```

Now if we run the **`INSERT`** statement, we will get an error complaining about the age of the new employee.

Code Listing 2-12: Insert trigger defines business rules that cannot be bypassed

```
INSERT INTO dbo.Employee
        ( FirstName ,
          LastName ,
          ReportsTo ,
          DepartmentId ,
          BirthDate ,
          HireDate ,
          TerminationDate )
VALUES  ( 'New' , -- FirstName - varchar(50)
          'Employee' , -- LastName - varchar(50)
          0 , -- ReportsTo - int
         1 , -- DepartmentId - int
          GETDATE() , -- BirthDate - datetime
          GETDATE() , -- HireDate - datetime
          GETDATE() + 60  -- TerminationDate - datetime
          )

Msg 50000, Level 16, State 10, Procedure trEmployeeInsertVerifyAge, Line
8
New hires must be at least 18 years old
Msg 3609, Level 16, State 1, Line 1
The transaction ended in the trigger. The batch has been aborted.
```

Because we rolled back the transaction, there will not be a record in either the **`Employee`** or **`EmployeeAudit`** tables.

> ***Note: This is not a good way to verify the age of the employee. There are many scenarios where this simple test would incorrectly fail, as well as incorrectly pass. Determining the age is a little bit more complex, but for the purpose of showcasing a trigger, let's keep it simple.***

Select statements

We read data from the database the **`SELECT`** statement. In its simplest form, we have:

Code Listing 2-13: The basic SELECT statement

```
SELECT fields
FROM table
WHERE conditions ARE true
```

We will see a few variations on this pattern here, and then spend the rest of the book exploring all the variations you could ever want to see.

Single table

As long as we are only considering a single table, the query stays relatively simple.

Code Listing 2-14: Querying from a single table

```
SELECT  EmployeeId ,
        FirstName ,
        LastName ,
        ReportsTo ,
        DepartmentId ,
        BirthDate ,
        HireDate ,
        TerminationDate
FROM    dbo.Employee
WHERE   EmployeeId < 100;
```

This will give us a list of all employee records with an **`EmployeeId`** value that is less than 100. This is not necessarily the same as the first 100 employees. Had any been deleted, there would be gaps in the **`EmployeeId`** sequence, and you would not get 100 records.

To get the first 100 employee records, we can run a slightly different query.

Code Listing 2-15: Getting the first 100 employee records

```
SELECT  TOP 100 EmployeeId ,
        FirstName ,
        LastName ,
        ReportsTo ,
        DepartmentId ,
        BirthDate ,
        HireDate ,
        TerminationDate
FROM    dbo.Employee
ORDER BY EmployeeId
```

Any column from the **`Employee`** table can be used to filter the records returned in the **`WHERE`** clause.

Code Listing 2-16: Filtering records in the WHERE clause

```
SELECT  EmployeeId ,
        FirstName ,
        LastName ,
        ReportsTo ,
        DepartmentId ,
        BirthDate ,
        HireDate ,
```

```
        TerminationDate
FROM    dbo.Employee
WHERE   ReportsTo = 51486
        AND ( HireDate < '01/01/2000'
              OR BirthDate < '01/01/1980' )
        AND ( TerminationDate IS NULL
              OR TerminationDate > GETDATE() )
ORDER BY EmployeeId;
```

We can combine these various filters any way we want to. Here we will get a list of employees who report to a specific employee and were either hired before the year 2000, or were born before 1980, and are either not terminated or scheduled to be terminated in the future.

Inner join

It is common to need to combine multiple tables to get the data that you need. The simplest way to combine data from two tables is with an inner join. With an inner join, the corresponding data used to define the join must be in both tables.

We could join the **Employee** and **Department** tables to a list of employees who have been terminated this year, and their departments:

Code Listing 2-17: Joining two tables using the foreign key

```
SELECT  FirstName ,
        LastName ,
        TerminationDate, DepartmentName ,
        CostCenter ,
        LocationId
FROM    dbo.Employee
        INNER JOIN dbo.Department ON Department.DepartmentId =
Employee.DepartmentId
WHERE   TerminationDate > DATEADD(yy, DATEDIFF(yy, 0, GETDATE()), 0);
```

If the employee did not have a **DepartmentId** set, or the **DepartmentId** was not in the **Department** table, that employee record would not be returned. Any department that does not have a record matching the filters in the **WHERE** clause will also be omitted.

Note: Because we added referential constraints through the foreign key definition between Employee and Department, any DepartmentId listed in the Employee table will be in the Department table, but not all Departments may be listed in the Employee table. Hopefully every department has not fired someone this year.

Note: Don't worry about the date functions in the WHERE clause. We will explore all things date-related in Chapter 6. The end result of these functions is to get the first day of the current year.

Outer join

Outer joins can be a little bit more complicated. To appreciate the differences associated with the various forms that an outer join can take, let's add a couple of new tables to consider.

Let's call them **`Left`** and **`Right`**:

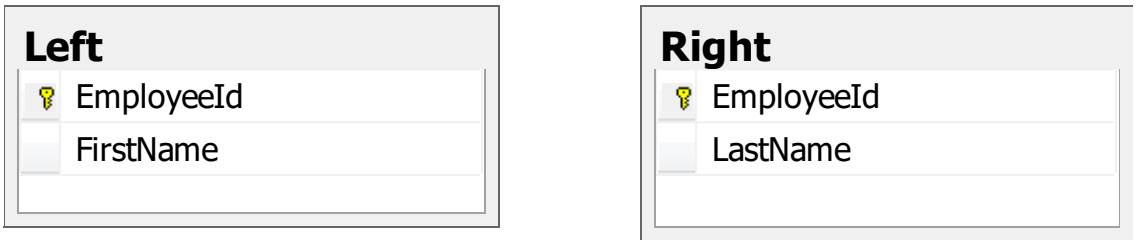


Figure 2-3: Contrived sample tables to showcase the different types of outer joins

We will join these two tables on the **`EmployeeId`**. Since we are explicitly showcasing what happens when both tables don't have all of the same records, we left off the foreign key constraints, and neither of these primary keys are set up as an identity column.

> ***Note: In actual practice you would never set up a couple of tables like this, but this makes it easy to showcase data scenarios that legitimately may crop up, but under more arcane business scenarios than we want to deal with here.***

Let's load these tables with some sample fictitious data. For the table **`Left`**:

EmployeeId	FirstName
1	Fred
2	Barney
3	Buzz
4	Han
5	Wedge

Result Set 2-1: Data in table Left

And for table **`Right`**:

EmployeeId	LastName
1	Flintstone
2	Rubble
6	Organo

EmployeeId	LastName
7	Jetson
8	Parker

Result Set 2-2: Data in table Right

With this data loaded, we can see that we will have some matching records and some records that will not match.

A simple inner join will return the following records:

Code Listing 2-18: An inner join between Left and Right

```
SELECT  *
FROM    dbo.[Left]
        JOIN [Right] ON [Right].[EmployeeId] = [Left].[EmployeeId];
```

Note: If the join is not explicitly specified, the join type is an inner join.

EmployeeId	FirstName	EmployeeId	LastName
1	Fred	1	Flintstone
2	Barney	2	Rubble

Result Set 2-3: Only records with matches in both tables are shown

We get all of the matching records. A left outer join will return the following records:

Code Listing 2-19: Left outer join between left and right

```
SELECT  *
FROM    dbo.[Left]
        LEFT OUTER JOIN [Right] ON [Right].[EmployeeId] =
[Left].[EmployeeId];
```

This is all the records from the first table in the join plus the matching records from the second table. For the records without a matching record in the second table, we fill in the blanks with NULL.

EmployeeId	FirstName	EmployeeId	LastName
1	Fred	1	Flintstone
2	Barney	2	Rubble

EmployeeId	FirstName	EmployeeId	LastName
3	Buzz	NULL	NULL
4	Han	NULL	NULL
5	Wedge	NULL	NULL

Result Set 2-2: All records from the Left table plus any matching records from the Right table

A right outer join will return the following records:

Code Listing 2-20: Right outer join between Left and Right

```
SELECT  *
FROM    dbo.[Left]
        RIGHT OUTER JOIN [Right] ON [Right].[EmployeeId] =
[Left].[EmployeeId]
```

This is all the records from the second table plus the matching records from the first table in the join.

EmployeeId	FirstName	EmployeeId	LastName
1	Fred	1	Flintstone
2	Barney	2	Rubble
NULL	NULL	6	Organo
NULL	NULL	7	Jetson
NULL	NULL	8	Parker

Result Set 2-1: All of the records from Right, and any matches from Left

A full outer join will return the following records.

Code Listing 2-21: A full outer join between Left and Right

```
SELECT  *
FROM    dbo.[Left]
        FULL OUTER JOIN [Right] ON [Right].[EmployeeId] =
[Left].[EmployeeId];
```

This is all the records from both tables matching the ones that have matches, filling in the blanks from either table with NULL values.

EmployeeId	FirstName	EmployeeId	LastName
1	Fred	1	Flintstone
2	Barney	2	Rubble
3	Buzz	NULL	NULL
4	Han	NULL	NULL
5	Wedge	NULL	NULL
NULL	NULL	6	Organo
NULL	NULL	7	Jetson
NULL	NULL	8	Parker

Result Set 2-2: Records from either table matching where possible

A cross join will return every combination from both tables. In this case, it will return 25 records.

Code Listing 2-22: A cross join between the Left and Right tables

```
SELECT  *
FROM    dbo.[Left]
        CROSS  JOIN [Right]
```

Tip: Very rarely will you want to use a cross join because of the number of records that it can return; plus, it can often return meaningless data. We will see a couple of practical examples a little later on.

Update statements

UPDATE statements allow us to change data in the database. Unlike the **INSERT** statement that we saw earlier, **UPDATE** statements do not result in a new record; they operate on an existing record. The data in the original record is replaced with the new values you specify.

An example of a simple **UPDATE** statement is:

Code Listing 2-23: The general UPDATE statement

```
UPDATE  dbo.Employee
SET     FirstName = 'New Value' ,
        LastName = 'New Last Name' ,
        ReportsTo = 1 ,
```

```
        DepartmentId = 0 ,
        BirthDate = '7/4/1776' ,
        HireDate = '1/20/2017' ,
        TerminationDate = NULL
WHERE   EmployeeId = 404;
```

We can update any number of columns in the set clause. The updated values can be based on other column values, can be the result of calculations, or maybe the result of manipulating their original values.

In the **WHERE** clause, we can filter down to the specific records that we want to update. Updating by the primary key is very common. You know that you will only update a single record.

Tip: The WHERE clause from an UPDATE statement can be used in a SELECT clause to know ahead of time how many records will be affected.

Let's look at a few examples.

Suppose you want to change the **Employee** table so that all employees who report to the person with ID **34149**, now report to the person with ID **35624**:

Code Listing 2-24: A basic UPDATE statement moving employees from one manager to another

```
UPDATE  dbo.Employee
SET     ReportsTo = 35624
WHERE   ReportsTo = 34149;
```

We may want to delay firing by 30 days for anyone who hasn't already been terminated:

Code Listing 2-25: Adjusting the termination date based on the WHERE condition

```
UPDATE  dbo.Employee
SET     TerminationDate = DATEADD(DAY, 30, TerminationDate)
WHERE   TerminationDate > GETDATE();
```

Identity column

You may find that you need to change the existing values for existing identity columns. Unfortunately, we cannot update the value of an identity column. Even though we can use **Set IdentityInsert** to insert a specific value, there is no **Set Identity_Update**.

We may still need to be able to change the value for the identity column, but we have a few extra steps to go through.

1. Create a new staging table based on the table that we want to update, but don't make the primary key an identity column.

2. Insert the values that need to be updated into this new table, specifying whatever value you need for the primary key.
3. Delete the records that you wanted to update from the original table.
4. Turn **`Identity INSERT`** on.
5. Insert the records from the staging table back into the original table.
6. Turn **`Identity INSERT`** off.

 ***Note: These are a lot of steps, but they're not difficult to do. In the next chapter, we will see an easy way to move a set of records from one table to another.***

Triggers

Triggers for updating a table are similar to insert triggers with just a couple of differences. Conceptually, we can think of an update as a **`DELETE`** followed by an **`INSERT`**. The values for the record being updated are inserted into the table after the original record is deleted. As we have already seen, an **`UPDATE`** statement can and often does affect many records.

In the body of the trigger, we get two tables that SQL Server maintains for us: the **`Inserted`** table that we saw in the **`INSERT`** trigger, and a **`Deleted`** table. In the case of updates, the **`Inserted`** table will track the new values for the records after they have been updated, and the **`Deleted`** table will track the original values as they were before the update.

We can join these two tables to track how the values change over time.

Code Listing 2-26: Tracking changes with an update trigger

```
CREATE  TRIGGER trEmployeeUpdate ON dbo.Employee
    FOR UPDATE
AS
    SELECT  CONVERT(VARCHAR(10), Deleted.EmployeeId) + ' becomes '
            + CONVERT(VARCHAR(10), Inserted.EmployeeId) AS EmployeeId ,
            Deleted.FirstName + ' becomes ' + Inserted.FirstName AS
FirstName ,
            Deleted.LastName + ' becomes ' + Inserted.LastName AS
LastName ,
            CONVERT(VARCHAR(10), Deleted.ReportsTo) + ' becomes '
            + CONVERT(VARCHAR(10), Inserted.ReportsTo) AS ReportsTo ,
            CONVERT(VARCHAR(10), Deleted.DepartmentId) + ' becomes '
            + CONVERT(VARCHAR(10), Inserted.DepartmentId) AS DepartmentId
,
            CONVERT(VARCHAR(23), Deleted.BirthDate, 126) + ' becomes '
            + CONVERT(VARCHAR(23), Inserted.BirthDate, 126) AS BirthDate
,
            CONVERT(VARCHAR(23), Deleted.HireDate, 126) + ' becomes '
            + CONVERT(VARCHAR(23), Inserted.HireDate, 126) AS HireDate ,
            CONVERT(VARCHAR(23), Deleted.TerminationDate, 126) + '
becomes '
```

```
            + CONVERT(VARCHAR(23), Inserted.TerminationDate, 126) AS
TerminationDate
    FROM    deleted
            JOIN inserted ON Inserted.EmployeeId = Deleted.EmployeeId;
```

The only thing complicated here is the data manipulation to concatenate all the original and new values as strings.

Tip: In actual usage, you would probably want to take the output of this join and store into a logging table. You probably would never want a trigger to directly return a Result Set; this is for illustration purposes only.

Any business rules that you implement as an **INSERT** trigger should also be implemented as an **UPDATE** trigger. SQL Server makes this is easy. In the **CREATE TRIGGER** statement, specify that it is for both **INSERT** and **UPDATE**.

Code Listing 2-27: Implementing business logic with update triggers

```
CREATE TRIGGER trEmployeeInsertVerifyAge ON dbo.Employee
    AFTER INSERT, UPDATE
AS
    IF EXISTS ( SELECT  *
                FROM    Inserted
                WHERE   DATEDIFF(YEAR, Inserted.BirthDate, GETDATE()) <
18 )
        BEGIN
            RAISERROR ('New hires must be at least 18 years old', 16,
10);
            ROLLBACK TRANSACTION;
            RETURN;
        END;
```

There are also some domain-specific business rules that are relevant for updates and not inserts that you may want to consider, for example:

- Prevent updates after an employee has been terminated.
- Prevent changes to a loan after closing.
- Prevent updates to an order after it has been shipped.

Delete statements

Sometimes we have to remove data from the database. We do this with the **DELETE** statement, which deletes the identified records from the database.

A sample **DELETE** statement in its simplest form is:

Code Listing 2-28: The basic DELETE statement

```
DELETE FROM dbo.Employee
WHERE EmployeeId = 44068
```

Just like with **UPDATE** statements, the **WHERE** clause for a **DELETE** statement can be used in a **SELECT** statement to verify the records that will be affected. Anything that you can do in the **WHERE** clause for a **SELECT** or **UPDATE** statement, you can also do in a **DELETE** statement.

Tip: Always run a Select with the same Where clause before deleting records to verify how many records will be removed by the Delete.

In some environments you may not use **DELETE** statements. For some businesses, data may never be deleted but simply marked as no longer active. In such cases, the **DELETE** statement will take the form of a soft delete, implemented as an **UPDATE** statement. Generally, this will involve updating the record that you want to delete and setting a value for a column indicating that the record has been deleted, such as **DeletedDate** and maybe **DeletedBy**.

Referential constraints

One of the problems that can arise from deleting records is the havoc that it can play with referential integrity. For example, you delete a record from the **Department** table while there are still employee records associated with the deleted department. As long as the foreign key relationships are properly defined, the database will not allow the delete.

Code Listing 2-29: Referential constraints blocking a delete

```
DELETE FROM dbo.Department WHERE DepartmentId = 1

The DELETE statement conflicted with the REFERENCE constraint
"FK_Employee_Department". The conflict occurred in database "SQL", table
"dbo.Employee", column 'DepartmentId'.
```

If the foreign key relationships are not properly defined, you are at risk for data integrity problems.

Tip: If you cannot properly define a foreign key in the database and allow the database to guarantee referential integrity, run reports periodically to ensure that your data is still clean.

Cascading deletes

A foreign key can be defined with an option to **On Delete Cascade**. This means that when a record is deleted, every record in related tables will be deleted as well. Obviously this can be dangerous, but can also simplify some data management tasks.

Naturally, we would not want to enable cascade delete between **Department** and **Employee**. It would not make sense to delete an employee record when a department is deleted.

Tip: If the foreign key column can be NULL, it is not a good candidate for cascade delete. If a record can exist without the foreign table, it should not be deleted when the foreign table is deleted.

In an order-tracking database, we may want to add cascading deletes to automatically delete **OrderDetail** records when the **Order** record is deleted. Without cascading deletes, you would have to delete the **OrderDetail** records before you could delete the **Order** record.

Code Listing 2-30: Constraints preventing deletes

```
DELETE  FROM dbo.[Order]
WHERE   OrderId = 1;

The DELETE statement conflicted with the REFERENCE constraint
"FK_LoanCode_LoanCodeType". The conflict occurred in database "SQL",
table "dbo.OrderDetail", column 'OrderId'.
```

If we delete an **Order** record without first deleting the associated **OrderDetail** record, we get the reference constraint violation shown in Code Listing 2-30.

If we define the constraint enabling cascade deletes:

Code Listing 2-31: Alter table to cascade delete

```
ALTER TABLE dbo.OrderDetail  WITH CHECK ADD  CONSTRAINT
[FK_OrderDetail_Order] FOREIGN KEY([OrderId])
REFERENCES dbo.[Order] ([OrderId])
ON UPDATE CASCADE
ON DELETE CASCADE;
```

Now if we execute the same **DELETE** statement, we won't get an error even if there are multiple records in the **OrderDetail** table.

Code Listing 2-32: Cascade delete bypasses constraint violations

```
DELETE  FROM dbo.[Order]
WHERE   OrderId = 1;
```

The message that we get back simply states that one record was affected, even though all of the related **OrderDetail** records were also deleted.

Tip: Be careful when cascade deletes are enabled—more records than reported may be deleted.

The importance of transactions

Deletes are permanent under normal circumstances. Once you issue a **DELETE** statement, the record is gone unless it was deleted as part of a transaction.

A database transaction represents a logical unit of work. Logical units of work have four properties, collectively known as ACID:

- **Atomicity** means that the transaction cannot be split into parts. Either all statements of the transaction succeed, or none of it succeeds.
- **Consistency** means that at the end of the transaction, all of the data is in a consistent state with all of the constraints and data integrity rules adhered to. If not, the transaction would have been rolled back to the state it was in before the transaction started.
- **Isolation** means that modifications made inside the transaction are kept isolated from outside the transaction until the transaction is complete. Also, changes made outside of the transaction are kept separate from inside the transaction until the transaction is complete.
- **Durability** means that once the transaction is complete, all of the changes made as part of the transaction are permanent. These changes have been made and stored on disk so that they would survive even a system failure.

Transactions help ensure data integrity and that business logic can be evaluated in a known state. They also provide a safety net for **UPDATE** and **DELETE** statements. If you start a transaction before executing a **DELETE** or **UPDATE** statement and these statements do not affect the number of records expected, you can roll back the transaction and undo any SQL statements executed.

Code Listing 2-33: DELETE as part of a transaction

```
-- Show the contents of the Order table before the Transaction
Begins.
SELECT * FROM dbo.[Order]
BEGIN TRANSACTION
-- Delete the contents of a single order.
DELETE  FROM dbo.[Order]
WHERE   OrderId != 1;
-- Show the contents of the Order table after deleting an Order.
SELECT * FROM dbo.[Order]
-- Rollback will discard all changes made since the transaction.
started
ROLLBACK
-- Show and confirm that the Order table is back to where we
started from.
SELECT * FROM dbo.[Order]
```

Summary

In this chapter we have seen the basics of the essential CRUD operations. We saw how to create data with an **`INSERT`** statement, how to read data with a **`SELECT`** statement, and how to use the **`UPDATE`** and **`DELETE`** statements to round out these operations.

We covered a lot of material, but if you have looked at the SQL in a typical application, you know that we have barely scratched the surface. In the next chapter we will pull the covers back a bit and explore versions of the basic operations that are just a bit more complex.

Chapter 3 More Advanced CRUD

If we knew all of the requirements upfront and could design the database to take them into account, our lives would be much simpler. In the real world, we don't know all of the requirements upfront. Even the requirements that we are given upfront will change over time, and eventually, many of the initial requirements may no longer be valid or may be unrecognizable given enough time.

The database is designed in the beginning, using the best information and best practices that we have at the time. Over time, this well-designed database will deteriorate into violating any number of best practices, usually as a result of the changing requirements. No one sets out to create a badly designed database, but anyone who has worked with the same data model for a decent length of time will tell you all sorts of things that they wished were different or that they would do differently if they could start from scratch.

Note: A lot of time can be spent complaining about a bad design or wondering why anyone would come up with the design that you currently have. Such efforts are rarely productive.

Fancy inserts

Select-based inserts

We saw this syntax briefly when we looked at auditing with insert triggers. Instead of explicitly giving the input values for an **INSERT** statement, we can provide a **SELECT** statement that will provide this data. This will insert one record for every record in the specified **SELECT** statement.

An example of a **SELECT**-driven insert is:

Code Listing 3-1: Select-driven insert

```
INSERT  INTO dbo.Department
        ( DepartmentName ,
          CostCenter ,
          LocationId )
        ( SELECT    'Accounting' ,
                    '6MRTG99RAR3N6BI443CRY0M3' ,
                    0);
```

This query will insert a single record because the associated select returns only a single record. Because we did not specify a table name, only the values explicitly given in the **SELECT** clause will be returned. Our **SELECT** statement can be as complex as any **SELECT** statement we will see, meaning that we can do some amazing things here.

This is a very powerful technique. With it you can easily move data from one table to another, as we did with the insert trigger. We can also use it to fill in missing records where constraint was missing so that you can add it back.

We have a couple of rules to follow for the **SELECT** statement:

- The columns in the **SELECT** must match the columns and data types from the **INSERT** statement.
- The **SELECT** statement cannot include an **Order By** statement. The database will complain because there is no need to sort the records being inserted.

Now let's revisit the auditing insert trigger from the last chapter.

Code Listing 3-2: Insert trigger revisited

```
CREATE TRIGGER trEmployeeInsert ON dbo.Employee
    FOR INSERT
AS
    INSERT INTO dbo.EmployeeAudit
            ( EmployeeId ,
              FirstName ,
              LastName ,
              ReportsTo ,
              DepartmentId ,
              BirthDate ,
              HireDate ,
              TerminationDate ,
              AuditAction ,
              AuditDateTime )
            SELECT  Inserted.EmployeeId ,
                    Inserted.FirstName ,
                    Inserted.LastName ,
                    Inserted.ReportsTo ,
                    Inserted.DepartmentId ,
                    Inserted.BirthDate ,
                    Inserted.HireDate ,
                    Inserted.TerminationDate,
                  'I' ,  -- I is for insert
                  GETDATE()
            FROM    Inserted;
```

This query builds on the fact that the **Inserted** table has the same structure as the table associated with the trigger. Additionally, the **Inserted** table has a record for each record that was inserted. Most of the time, this will be a single record, but if it happened to have two records, or even a thousand records, this query will handle keeping the audit table up-to-date.

As another example, we can look at what might happen if the foreign key constraints between the **Employee** and **Department** tables was accidently dropped. By the time we discovered this problem, we may have **Employee** records referring to **DepartmentIds** that do not exist. We

would like to correct this problem by adding the missing records to the **Department** table so that we can enable the constraint and keep our data clean.

We can run a query like this to find the missing **Department** records:

Code Listing 3-3: Select to identify missing records to insert

```
SELECT DISTINCT DepartmentId
FROM    dbo.Employee
WHERE   DepartmentId NOT IN
      ( SELECT    DepartmentId
      FROM      dbo.Department );
```

Don't worry about this **SELECT** syntax yet—we will cover it shortly. For now, just know that it will produce a list of the **DepartmentIds** that are not in the **Department** table.

Before we can insert these records, we will need to update the database to allow us to insert a value into the **Identity** column. After we insert these records, we want to disable explicitly inserting values into the identity columns. The full code to handle inserting the missing records will be:

Code Listing 3-4: Enabling identity insert to give specific DepartmentId for the insert

```
SET IDENTITY_INSERT dbo.Department ON;

INSERT  INTO dbo.Department
        ( DepartmentId ,
          DepartmentName )
        ( SELECT DISTINCT
                  DepartmentId ,
                  'Missing Department Added by Data Cleanup'
DepartmentName
          FROM      dbo.Employee
          WHERE     DepartmentId NOT IN ( SELECT    DepartmentId
                                          FROM      dbo.Department ));

SET IDENTITY_INSERT dbo.Department OFF;
```

Note: We can infer the missing DepartmentId values based on the usage in the Employee table, but this does not give us any guidance on what the DepartmentName should be. We want to supply a standard name that will obviously not be the real name, and that we can easily track and go back to correct as a second step in the data cleanup.

Since we can explicitly set the value of an identity column during the insert, you may wonder about explicitly updating its value. After all, this is also a common scenario. Suppose that after an operational reorganization, you need to update the identity column for every employee hired

after a particular date. Turns out this not as simple as issuing an **UPDATE** statement; there are many steps to go through for what conceptually should be a relatively straightforward task.

> ***Note: In actual practice, never do this. The primary key to a table should never have any meaning outside of uniquely identifying a record. If you need a number that has a business meaning, this should be a separate column with no database meaning. In this case, you probably needed an EmployeeId column and an EmployeeNumber column. The Id column is used to maintain the data in the database, while the Number column may have whatever business meaning is appropriate.***

The steps to update the value of an identity column are:

1. Create a new table with the same structure as the one we want to update, without specifying an identity column. We will call this the **Staging** table.
2. Insert the records that need to be updated into this new table setting the identity column as you want it updated.
3. Delete the records from the original table that were just copied.
4. Allow inserts into the identity column on the original table.
5. Insert the records from the **Staging** table back into the original table.
6. Disable inserts into the identity column.
7. Drop the **Staging** table.

The full code to handle this type of change looks like this:

Code Listing 3-5: Complete process for updating existing identity column values

```
SELECT  EmployeeId + 1000 AS EmployeeId ,
        FirstName ,
        LastName ,
        ReportsTo ,
        DepartmentId ,
        BirthDate ,
        HireDate ,
        TerminationDate
INTO    EmployeeStaging
FROM    dbo.Employee
WHERE   HireDate > DATEADD(MONTH, -3, GETDATE());

DELETE  FROM dbo.Employee
WHERE   HireDate > DATEADD(MONTH, -3, GETDATE());

SET IDENTITY_INSERT dbo.Employee ON;
INSERT  INTO dbo.Employee
        ( EmployeeId ,
          FirstName ,
          LastName ,
          ReportsTo ,
          DepartmentId ,
          BirthDate ,
```

```
          HireDate ,
          TerminationDate )
        ( SELECT    EmployeeId ,
                    FirstName ,
                    LastName ,
                    ReportsTo ,
                    DepartmentId ,
                    BirthDate ,
                    HireDate ,
                    TerminationDate
          FROM      dbo.EmployeeStaging);
SET IDENTITY_INSERT dbo.Employee OFF;
```

Practical uses of Cartesian products

We touched on this briefly in the last chapter when we talked the various ways to join tables. A cross join will return every possible combination from the tables being joined. If you have two records in one table and four records in another table, the result of the cross join will include eight records.

To see how this could be useful, let's add a new table. The **`Ordinal`** table will look like this:

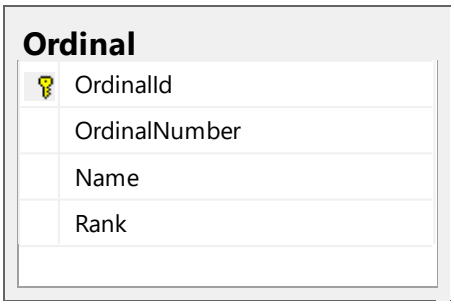


Figure 3-1: Ordinal table

With this table, we can easily produce a query that will return any number of records, incorporating details from whatever table we want to cross join with it. For example, we could simulate eight employee records for each department using the following query:

Code Listing 3-6: Forcing an eight-way Cartesian product on the Employee table

```
SELECT  'Employee' FirstName ,
        Name LastName ,
        DepartmentId
FROM    dbo.Department
        CROSS JOIN dbo.Ordinal
WHERE   OrdinalNumber <= 8;
```

Which will give us the following results:

FirstName	LastName	DepartmentId
Employee	One	1
Employee	Two	1
Employee	Three	1
Employee	Four	1
Employee	Five	1
Employee	Six	1
Employee	Seven	1
Employee	Eight	1
Employee	One	2
Employee	Two	2
Employee	Three	2
Employee	Four	2
Employee	Five	2
Employee	Six	2
Employee	Seven	2
Employee	Eight	2

Result Set 3-1: Results of the Cartesian query

Wrap this in an **`INSERT`** statement, and we will insert eight new employee records for each department. This could be useful for loading dummy data into a database to perform load testing or anytime substantial data needs to be staged.

Tip: Red Gate has a great tool for generating data: the SQL Data Generator. You can use it to easily generate hundreds of thousands of records for any table.

You could also use this technique to create 12 budget records for each department. This would be one for each month of the year, making it easier to verify that every possible record is created from the beginning and subsequent processing can focus on updating existing records.

Tip: This technique can be useful anytime you want to ensure that you have accounted for inserting all possible records.

Fancy updates

The updates we saw in the previous chapter all focused on a single table. As a record was updated, the new values were either explicitly given or determined by manipulating other values from the record being updated. This is often all that is needed, but not always—sometimes you need something fancier.

Correlated updates

A correlated update allows you to define an **`UPDATE`** statement in terms of a **`SELECT`** statement and use any of the columns from any of the tables joined in the **`SELECT`** statement to determine the update values.

To show how this works, let's revisit the order-tracking tables from the last chapter.

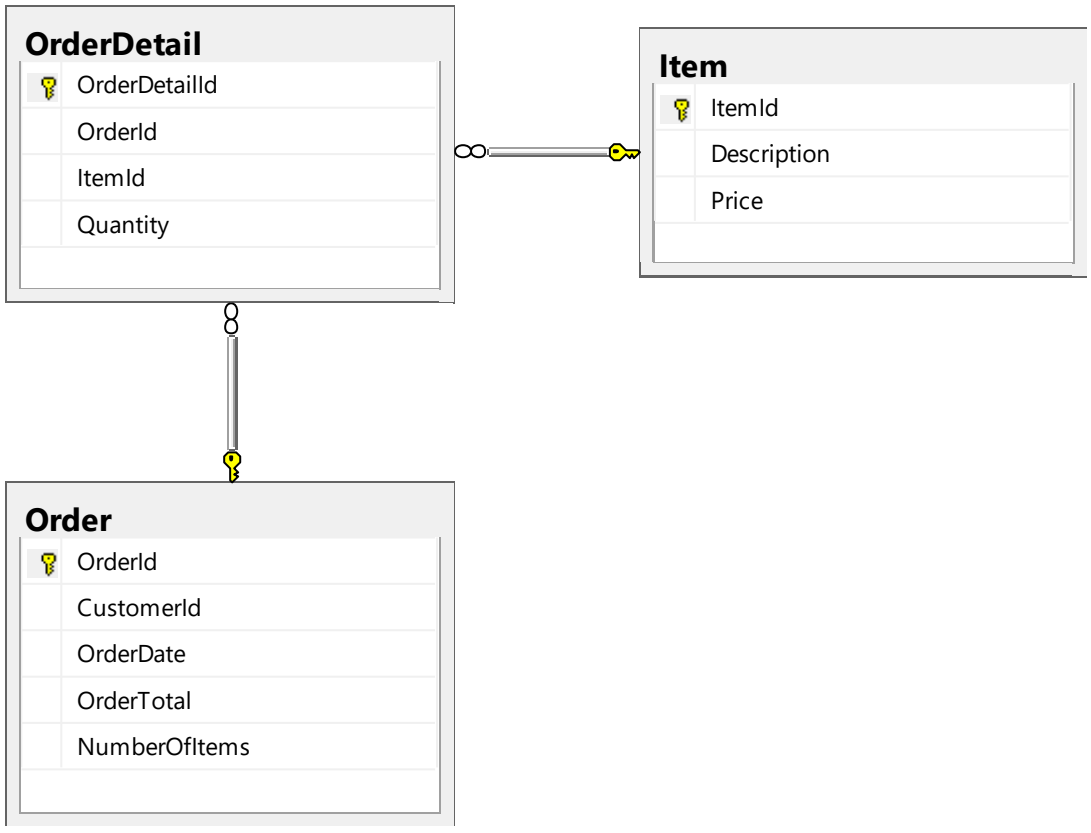


Figure 3-2: A classic order-tracking data model

We have an **`Order`** table and an **`OrderDetail`** table. These are staples for any e-commerce system. The **`Order`** table includes a roll up from the **`OrderDetail`** table.

Tip: Summarizing the detail data in the Order record is a common performance trick, even though it violates several rules for good database design. Sometimes good design takes a back seat to performance.

We can update these values with a query like this:

Code Listing 3-7: Correlated update

```
DECLARE @OrderID INT;
SET @OrderID = 41867
UPDATE  dbo.[Order]
SET     NumberOfItems = Summary.NumberOfItems ,
        OrderTotal = Summary.OrderTotal
FROM    dbo.[Order]
        JOIN ( SELECT   SUM(Quantity) NumberOfItems ,
                        SUM(Quantity * Price) OrderTotal ,
                        OrderId
               FROM     dbo.OrderDetail
                        JOIN dbo.Item ON Item.ItemId = OrderDetail.ItemId
               GROUP BY OrderId ) Summary ON Summary.OrderId =
[Order].OrderId
WHERE   [Order].OrderId = @OrderID;
```

This query is a little bit more complicated than is typical in a correlated query because the query being used cannot directly include a **`GROUP BY`**, so we had to move the **`GROUP BY`** to a subquery.

In this example, we define the variable **`@OrderId`** to constrain the **`UPDATE`** to a single **`Order`** record. Without the **`WHERE`** clause on the **`UPDATE`** statement, every **`Order`** record would be updated. This may often be what you want, but for the first pass it's a good idea to update a single record and ensure that you get the expected results before forcing an update on every record. Adding a variable is another good practice because it keeps the query free of literal values, and makes it easier to spot where to change the literal value if you need to change the query to update a different **`Order`** record.

Note: Don't worry about the GROUP BY syntax in the query. We will cover this syntax in detail in the next chapter, along with aggregate functions like SUM.

A subquery is a little bit of SQL magic where we can define a **`SELECT`** statement and treat the result set as a temporary table that exists only as long as the containing query is running. This can be very useful, but can lead to queries that are difficult to read, depending on how complex the derived table's definition is.

Common table expressions

Common table expressions (CTEs) are similar to subqueries, but much more powerful. Like subqueries, a CTE is a named query that exists only for the duration of the single statement that comes right after its definition. What makes CTEs more powerful is that they can be referenced multiple times in the same query. A CTE can also reference itself. Such recursive queries open up some very interesting possibilities, which we will explore in Chapter 5.

The basic syntax for a CTE is:

Code Listing 3-8: The basic structure of a CTE

```
WITH CTE AS (SELECT statement)
```

For the **SELECT** statement, we can do anything that we can do in any **SELECT** statement. So we can rework Code Listing 3-7 with a CTE like this:

Code Listing 3-9: Basic syntax for common table expressions (CTE)

```
WITH    Summary  (NumberOfItems, OrderTotal, OrderID)
          AS ( SELECT   SUM(Quantity) NumberOfItems ,
                        SUM(Quantity * Price) OrderTotal ,
                        OrderId
               FROM     dbo.OrderDetail
                        JOIN dbo.Item ON Item.ItemId = OrderDetail.ItemId
               GROUP BY OrderId)
    UPDATE  dbo.[Order]
    SET     NumberOfItems = Summary.NumberOfItems ,
            OrderTotal = Summary.OrderTotal
    FROM    dbo.[Order]
            JOIN Summary ON Summary.OrderId = [Order].OrderId
    WHERE   [Order].OrderId = @OrderId;
```

The **WITH** clause names the CTE and specifies the names for the columns that will be included in the result set. You don't have to include the list of columns as long as all of the columns are explicitly named in the query.

Tip: If you specify the list of columns in the WITH clause, the number of columns must match the number of columns in the SELECT statement, but the names do not need to match. The names in the WITH clause will be the ones used outside of the CTE, not the names in the SELECT statement.

We can include multiple CTEs in a query. Each CTE is separated by commas. We could rewrite the **UPDATE** statement using two CTEs:

Code Listing 3-10: Multiple CTEs in a single query

```
WITH    Units ( NumberOfItems, OrderID )
          AS ( SELECT   SUM(Quantity) NumberOfItems ,
                        OrderId
               FROM     dbo.OrderDetail
               GROUP BY OrderId),
        Volume ( OrderTotal, OrderID )
          AS ( SELECT   SUM(Quantity * Price) OrderTotal ,
                        OrderId
               FROM     dbo.OrderDetail
                        JOIN dbo.Item ON Item.ItemId = OrderDetail.ItemId
               GROUP BY OrderId)
```

```
    UPDATE  dbo.[Order]
    SET     NumberOfItems = Units.NumberOfItems ,
            OrderTotal = Volume.OrderTotal
    FROM    dbo.[Order]
            JOIN Units ON Units.OrderID = [Order].OrderId
            JOIN Volume ON Volume.OrderID = [Order].OrderId
    WHERE   [Order].OrderId = @OrderID;
```

Here we have two CTEs, **Units** and **Volume**. In actual practice you would not want to separate these into two CTEs, but this showcases how to include multiple CTEs in a query.

We have only scratched the surface of what is possible with CTEs, and we will explore them more fully in Chapter 5.

Merge statement

With the **MERGE** statement, you can replace a lot of messy code with something much more maintainable. Unfortunately, the syntax takes some getting used to because it supports several different clauses to support multiple types of data modifications.

The general syntax for the **MERGE** statement is:

Code Listing 3-11: Basic MERGE statement with an update

```
MERGE TargetTable
USING SourceTable
ON TargetTable.TargetId = SourceTable.TargetId
WHEN MATCHED THEN
  UPDATE SET ColumnName = value;
```

We can also include a common table expression. The syntax also includes clauses for not matching the source and not matching the target.

We can rewrite the **UPDATE** statement using the **MERGE** statement along with a helpful CTE:

Code Listing 3-12: MERGE statement using a CTE

```
WITH    OrderSummary
          AS ( SELECT   OrderId ,
                        SUM(Quantity) AS NumberOfItems ,
                        SUM(Price) AS OrderTotal
               FROM     dbo.OrderDetail
                        JOIN dbo.Item ON Item.ItemId = OrderDetail.ItemId
               GROUP BY OrderId)
    MERGE dbo.[Order]
    USING OrderSummary
    ON [Order].OrderId = OrderSummary.OrderId
```

```
    WHEN MATCHED THEN
        UPDATE SET
               NumberOfItems = OrderSummary.NumberOfItems ,
               OrderTotal = OrderSummary.OrderTotal ;
```

Here the **Target** table is the **Order** table. The **Source** table is the CTE. This may look overly complicated for a single **UPDATE** statement, but we aren't limited to a single one. We can incorporate any number of **WHEN** clauses:

Code Listing 3-13: MERGE statement handling multiple business rules

```
WITH    OrderSummary
          AS ( SELECT   OrderId ,
                        SUM(Quantity) AS NumberOfItems ,
                        SUM(Price) AS OrderTotal
               FROM     dbo.OrderDetail
                        JOIN dbo.Item ON Item.ItemId = OrderDetail.ItemId
               GROUP BY OrderId)
    MERGE dbo.[Order]
    USING OrderSummary
    ON [Order].OrderId = OrderSummary.OrderId
    WHEN MATCHED THEN
        UPDATE SET
               NumberOfItems = OrderSummary.NumberOfItems ,
               OrderTotal = OrderSummary.OrderTotal
    WHEN MATCHED AND CustomerId = 340419 THEN
        UPDATE SET
               NumberOfItems = 0 ,
               OrderTotal = 0
    WHEN NOT MATCHED BY SOURCE THEN
        UPDATE SET
               NumberOfItems = 0 ,
               OrderTotal = 0 ;
```

Once you are comfortable with the syntax for the **MERGE** statement, you see that this gives us a very concise format to express multiple business rules in a single readable statement.

Fancy deletes

Deletes are rarely fancy, and we don't want them to be fancy, since they are destructive to existing data. Even though we like boring, simple **DELETE** statements, we occasionally need to get a bit fancier than the delete by primary key that we saw in the last chapter.

Select feeding a delete

Let's start by revisiting the problems we had deleting a **Department** record with associated **Employee** records because we didn't want to enable cascade deletes for these two tables. Without cascade deletes, we get the following the following error.

Code Listing 3-14: A reference constraint raised by a DELETE statement

```
DELETE FROM dbo.Department
WHERE DepartmentId = 1

The DELETE statement conflicted with the REFERENCE constraint
"FK_Employee_Department". The conflict occurred in database "SQL", table
"dbo.Employee", column 'DepartmentId'.
```

So how do we solve this problem without cascading deletes?

Tip: If a record could exist without the foreign key value (such as an employee without a department), it is not a good candidate for cascade deletes.

We need to first delete all the records in the related table before deleting the target records in the main table. This is relatively straightforward if we want to delete a single record; it just requires two **DELETE** statements in the right order.

Code Listing 3-15: Explicitly delete related records before deleting the foreign record

```
DELETE  FROM dbo.Employee
WHERE   DepartmentId = 1;

DELETE  FROM dbo.Department
WHERE   DepartmentId = 1;
```

But what if you have a group of **Department** records to delete?

Code Listing 3-16: Deleting more than one department at a time

```
DELETE  FROM dbo.Department
WHERE    LocationId = 95
```

Clearly we don't want to handle these deletes one at a time. We have no idea how many **Department** records are in our target location. Fortunately, we can write a query to get the list of **Department** records that we want to delete and feed that to the **DELETE** statement.

Code Listing 3-17: Identifying the departments to be deleted based on the location

```
SELECT  DepartmentId
FROM    dbo.Department
```

```
WHERE   LocationId = 95;
```

This will give us a list of the departments to be deleted. Now we can feed this list to the **DELETE** statement.

Code Listing 3-18: Deleting related records and primary records together

```
DELETE  FROM dbo.Employee
WHERE   DepartmentId IN ( SELECT    DepartmentId
                          FROM      dbo.Department
                          WHERE     LocationId = 95 );

DELETE  FROM dbo.Department
WHERE   LocationId = 95;
```

Regardless of how many **Department** records are going to be deleted, these two queries will handle the deletes.

Note: This is a great example of thinking in terms of sets instead of thinking procedurally.

A classic procedural approach to solving this problem would tackle the departments one at a time iteratively, perhaps even looping through a cursor. This is substantially more work than is necessary.

It is much simpler and more efficient to think in terms of sets. The inner **SELECT** statement creates a set of the departments that we want to delete. We then delete the contents of the set as a whole (conceptually) instead of one at a time.

Merge statement

We already talked about the **MERGE** statement as a fancy update; we can also use it to handle inserts and deletes. We can handle these deletes with the following query:

Code Listing 3-19: Deleting the related records as a MERGE statement

```
MERGE dbo.Employee
USING dbo.Department
ON Employee.DepartmentId = Department.DepartmentId
WHEN MATCHED AND LocationId = 95 THEN
    DELETE;
```

This is much easier than the previous queries to handle the delete.

Summary

We have taken the complexity for each of the basic statements up a notch. A common theme that we have seen in each of these scenarios is how a query can be used to drive and control more complex data manipulation scenarios.

We have also introduced a couple of new concepts: the common table expression and the **`MERGE`** statement. Both of these are powerful tools to add to our tool belt.

Now we will turn our attention to more complex **`SELECT`** statements and explore various ways to slice and dice data.

Chapter 4 Slicing and Dicing Data

Aggregate functions

After we have applied the appropriate joins to get related data and filters to filter out the data that we are not interested in, we may want to run some calculations to summarize the relevant data. Aggregate functions perform their calculations against a set of values, returning a single value.

The most common aggregate functions you are likely to use include:

Table 4-1: List of aggregate functions

AVG	Returns the average value for the column specified.
COUNT	Returns the count of the number of records in the identified set.
MAX	Returns the largest value for the column specified.
MIN	Returns the smallest value for the column specified.
SUM	Returns the sum of the values for the column specified.
StdDev	Returns the statistical standard deviation for the column specified.
VAR	Returns the statistical variance for the column specified.

Unless you are a statistician, you will probably mainly use the top five from the list, but these functions can combine to provide some interesting insights into your data. Here we will focus on the nonstatistical functions.

Note: Null values are ignored when evaluating any aggregate function.

Group By

The **GROUP BY** statement is used with the aggregate functions to group the result set by one or more columns. Any column from the **SELECT** clause that is not used in an aggregate function must be included in the **GROUP BY** clause. You can include any additional columns that you might need, but it rarely makes sense to group by a column that will not be displayed: the results would not make sense.

A simple example of a SQL statement using aggregate functions and a **GROUP BY** statement is:

Code Listing 4-1: General syntax for an aggregate function

```
SELECT  DepartmentId ,
        MIN(HireDate) ,
        MAX(HireDate)
FROM    dbo.Employee
GROUP BY DepartmentId;
```

This query will give a list of **DepartmentId**s with the first and last **HireDate** for each department.

Sum and Count

The **SUM** aggregate function only makes sense for numeric data types. It doesn't make sense to add up dates or strings, but any numeric data type can be summed.

COUNT is applicable regardless of the data type, except for text, image, or ntext, because columns of these types are not stored with the rest of the record. In general, we don't care about which column is specified for the **COUNT** function; we just count the number of records.

Both of these functions can also take a distinct modifier, which means that instead of operating on every record in the result set, they will operate on the unique values in the result set. If you do not specify a modifier, then **ALL** will be assumed.

Code Listing 4-2: Query showing the difference between COUNT and COUNT DISTINCT

```
SELECT  COUNT(1) NumberOfEmployees ,
        COUNT(DISTINCT DepartmentId) NumberOfDepartments
FROM    dbo.Employee;
```

Even though **SUM** requires numeric input, we can get useful information against nonnumeric data. We can use SQL Server's **CASE** statement to give us some numeric values to add up.

Code Listing 4-3: Using the CASE statement to provide a numeric value to add up

```
SELECT  SUM(CASE WHEN HireDate < '1/1/1970' THEN 1
                 ELSE 0
            END) HiredBeforeThe70s ,
        SUM(CASE WHEN HireDate BETWEEN '1/1/1970' AND '12/31/1979' THEN 1
                 ELSE 0
            END) HiredDuringThe70s ,
        SUM(CASE WHEN HireDate BETWEEN '1/1/1980' AND '12/31/1989' THEN 1
                 ELSE 0
            END) HiredBeforeThe80s ,
        SUM(CASE WHEN HireDate BETWEEN '1/1/1990' AND '12/31/1999' THEN 1
                 ELSE 0
            END) HiredBeforeThe80s
```

```
FROM    dbo.Employee;
```

We use the multiple case statements to compare the **HireDate** and return either a 0 or a 1. When we add 0 it will have no effect, but when we add a 1 we will increase the count for that date range.

This will return a single record with four counts:

Hired Before the 70s	Hired in the 70s	Hired in the 80s	Hired in the 90sT
312	185	169	185

Result Set 4-1: Breaking hiring data into bands based on hire date

This is a handy way to break the data up into intervals.

To get a better feel for how this works, let's drop the aggregate functions and look at some sample rows.

Code Listing 4-4: Using the CASE statement without SUM

```
SELECT  (CASE WHEN HireDate < '1/1/1970' THEN 1
              ELSE 0
         END) HiredBeforeThe70s ,
        (CASE WHEN HireDate BETWEEN '1/1/1970' AND '12/31/1979' THEN 1
              ELSE 0
         END) HiredDuringThe70s ,
        (CASE WHEN HireDate BETWEEN '1/1/1980' AND '12/31/1989' THEN 1
              ELSE 0
         END) HiredBeforeThe80s ,
        (CASE WHEN HireDate BETWEEN '1/1/1990' AND '12/31/1999' THEN 1
              ELSE 0
         END) HiredBeforeThe80s
FROM    dbo.Employee;
```

Hired Before the 70s	Hired in the 70s	Hired in the 80s	Hired in the 90sT
0	0	0	1
1	0	0	0
0	0	0	1
0	1	0	0

Hired Before the 70s	Hired in the 70s	Hired in the 80s	Hired in the 90sT
1	0	0	0
0	0	0	1
0	1	0	0
1	0	0	0

Result Set 4-2: Only one column in each record will have a value

Each row returned will have at most one column with a value of 1. The other columns will be 0. If all of the columns are 0, then that particular employee was hired after the 90s. This provides a powerful way to summarize data quickly.

We can also combine a **`Group By`** to get this summarized by department:

Code Listing 4-5: Summarizing the metrics by department

```
SELECT  DepartmentName ,
        SUM(CASE WHEN HireDate < '1/1/1970' THEN 1
                 ELSE 0
            END) HiredBeforeThe70s ,
        SUM(CASE WHEN HireDate BETWEEN '1/1/1970' AND '12/31/1979' THEN 1
                 ELSE 0
            END) HiredDuringThe70s ,
        SUM(CASE WHEN HireDate BETWEEN '1/1/1980' AND '12/31/1989' THEN 1
                 ELSE 0
            END) HiredBeforeThe80s ,
        SUM(CASE WHEN HireDate BETWEEN '1/1/1990' AND '12/31/1999' THEN 1
                 ELSE 0
            END) HiredBeforeThe80s
FROM    dbo.Employee
        INNER JOIN dbo.Department ON Department.DepartmentId =
Employee.DepartmentId
GROUP BY DepartmentName;
```

Department Name	Hired Before the 70s	Hired in the 70s	Hired in the 80s	Hired in the 90sT
Accounting	50	36	29	31
BusinessSales	7	3	4	3
ConsumerSales	5	2	1	2
Corporate Care	44	25	24	25

Department Name	Hired Before the 70s	Hired in the 70s	Hired in the 80s	Hired in the 90sT
CorporateSales	6	4	2	2
Customer	23	7	13	18
InternationalSales	6	8	2	3
Marketing	29	19	20	16
NationalSales	5	2	5	4
Prepaid Customer	46	17	21	26
Sales	6	6	3	4
Service	15	17	10	8
Technical	27	18	14	13
TechnicalSales	13	6	6	7
Web	27	11	13	22

Result Set 4-3: Summarizing the hiring statistics by department

Min and Max

The **MIN** and **MAX** aggregate functions work on a wide range of data types. You can find the **MIN** and **MAX** for **varchars**, **DateTime**, and any numeric types. **MIN** and **MAX** do not make sense against a bit field or image. These functions can be used to find the oldest and youngest employees in each department. When applied to **Varchars** or **NVarchar**, these functions will do an alphabetic comparison, which can also provide useful insight into your data.

We can also combine **MIN** and **MAX** with the other aggregate functions to get more insight into your data, provided that the other aggregate functions are in a nested query or a common table expression.

Note: You cannot perform an aggregate function on an expression containing an aggregate or a subquery.

We can use this technique to pivot data in the result set. This allows us to convert the rows in the result set to columns in a new result set. Let's see how we might construct a summary for a monthly sales report.

Going back to the **Order** tables that we have been working with, let's step through what a sales report might look like.

Code Listing 4-6: Building a sales report

```
SELECT  MAX(CASE OrderSummary.OrderDate
              WHEN 1 THEN ( OrderSummary.OrderTotal )
              ELSE 0
```

```
            END) AS JanuaryOrders ,
        MAX(CASE OrderSummary.OrderDate
              WHEN 2 THEN ( OrderSummary.OrderTotal )
              ELSE 0
            END) AS FeburaryOrders ,
        MAX(CASE OrderSummary.OrderDate
              WHEN 3 THEN ( OrderSummary.OrderTotal )
              ELSE 0
            END) AS MarchOrders
FROM    ( SELECT    DATEPART(MONTH, OrderDate) AS OrderDate ,
                    SUM(OrderTotal) AS OrderTotal
          FROM      dbo.[Order]
          GROUP BY  DATEPART(MONTH, OrderDate) ) OrderSummary;
```

Note: Don't worry about the DatePart function yet. We will cover all of the date and time functions in Chapter 7.

The results from this query may look like this, depending on how the company did that quarter:

January Orders	February Orders	March Orders
18206	5212	14586

Result Set 4-4: Sales report rollup and pivoted

To see the impact of the **MAX** aggregate functions, let's review the results of the query without them. Without the **MAX** aggregate functions, we get a result set with 12 records.

January Orders	February Orders	March Orders
18206	0	0
0	5212	0
0	0	14586
0	0	0
0	0	0
0	0	0
0	0	0
0	0	0

0	0	0
0	0	0
0	0	0
0	0	0

Result Set 4-5: Sales report before the rollup

Each record will have a valid value in one column at most. The **MAX** aggregates will collapse all of these records into a single record, showing the valid record for each column.

Tip: We could add a Group By clause to get a summary breakdown for any level in the Sales hierarchy (such as Group, Region, Territory, etc.) that we are interested in.

Having

The **HAVING** clause comes after any **GROUP BY** statements your query may include. The **HAVING** clause allows you to provide filters based on the results on the aggregate function calls.

A simple example of a SQL statement including a **HAVING** clause is:

Code Listing 4-7: General syntax for a HAVING clause

```
SELECT  CustomerId ,
        COUNT(1)
FROM    dbo.[Order]
GROUP BY CustomerId
HAVING  COUNT(1) > 20;
```

With this query we can easily identify frequent customers.

The **HAVING** clause is not often used and is often overlooked, partly because we generally need to report on the results of the aggregate functions, and not filter based on their output. We can also get the same results without the **HAVING** clause.

Code Listing 4-8: Simulating the effects of a HAVING clause

```
SELECT  OrderCounts.CustomerId ,
        OrderCounts.NumberOfOrders
FROM    ( SELECT    CustomerId ,
                    COUNT(1) NumberOfOrders
          FROM      dbo.[Order]
          GROUP BY  CustomerId ) OrderCounts
WHERE   OrderCounts.NumberOfOrders > 20;
```

Tip: Even though it's easily overlooked, keep the HAVING clause in mind. When needed, this clause will simplify the query syntax.

Sorting

Once we have identified the records we are interested in, we often need to sort the data to put it into context and help ensure that we do not overlook relevant data by keeping the related records near each other in the result set. If you are looking at a detail sales records for a division, you want all of the detail records to be together.

There are just a few rules for when and how you can sort. Columns of type ntext, text, image, geography, geometry, and xml cannot be used in an **ORDER BY** clause. This is primarily because there is no clear sequence strategy for these data types.

The **ORDER BY** clause cannot be included in views, inline functions, derived tables, subqueries, etc., because these structures are used in larger queries that would generally handle sorting that would contradict the sorting defined elsewhere.

Note: The exception to this restriction is if a TOP, OFFSET, or FETCH clause is specified. We will discuss OFFSET and FETCH shortly. The TOP clause is straightforward—you simply specify the number of records to return. With any of these clauses included, the sort order will influence which records will be returned.

Changing directions

If you do not specify the sort direction, the direction will default to ascending.

Code Listing 4-9: Sorting by the number of orders ascending

```
SELECT  CustomerId ,
        COUNT(1) NumberOfOrders
FROM    dbo.[Order]
GROUP BY CustomerId
HAVING  COUNT(1) > 20
ORDER BY NumberOfOrders;
```

This query will show the customer with the highest number of orders last, while the query in Code Listing 4-10 will show the customer with the most number of orders first:

Code Listing 4-10: Sorting by the number of orders descending

```
SELECT  CustomerId ,
        COUNT(1) NumberOfOrders
```

```
FROM    dbo.[Order]
GROUP BY CustomerId
HAVING  COUNT(1) > 20
ORDER BY NumberOfOrders DESC;
```

Based on your needs, you will generally know the direction that you want to sort, but sometimes it may not be so clear cut. We can make the details of the **`ORDER BY`** more dynamic. We can specify the conditional case statement evaluated for each record returned to determine how that record should be sorted.

Let's look at a slightly contrived scenario. Revisiting the **`Department`** table, we may want to sort by the **`DepartmentName`** unless the **`LocationId`** is less than 5, in which case we want to sort by the **`CostCenter`**.

Code Listing 4-11: Deciding how to sort record by record

```
SELECT  DepartmentId ,
        DepartmentName ,
        CostCenter ,
        LocationId
FROM    dbo.Department
ORDER BY CASE WHEN LocationId < 5 THEN DepartmentName
              ELSE CostCenter
         END;
```

DepartmentId	Department Number	Cost Center	LocationId
8	ConsumerSales	1R8JASDAFAD	5
3	Prepaid Customer	3SXY6JLYJM1	5
7	Database	80O3SAS8SP6	2
12	Prepaid Customer	DNM1OUVCS	5
14	Express Marketing	1SNXK5DFDSF	1
5	Inside Accounting	VVCELPQURP	3
2	InternationalSales	IDAP8Q32CB	4
6	Outside Accounting	NI04JEDVP8RE	1
10	Phone Marketing	I8J5XNIYOE9KF	2
15	Service	K4DFDFSGGH	1
4	Service Hardware	FWR9NEF5X	1

DepartmentId	Department Number	Cost Center	LocationId
13	Service Software	WGFB3HWMD	4
9	TechnicalSales	SU476Y3GFGF	2
11	Web	DZYFWH4JXIJJ	3
1	Web Marketing	E5Z3BSG80R1	2

Result Set 4-6: Sorting differently based on the location

Note: If you want to specify the direction for a conditional sort, you specify it after the end in the case statement. A conditional order by cannot change the direction of the order by, only which column to use.

Multiple sorts

You can sort by multiple columns; each column can have a different direction, and each column could potentially be a conditional **`ORDER BY`**. The individual columns specified must be unique. It does not make sense to order by the same column twice. The sequence of the columns specified matter. The result set retrieved is sorted by the first column specified, and then this result is sorted by the second column specified, etc.

Code Listing 4-12: Sorting by multiple columns

```
SELECT  DepartmentId ,
        DepartmentName ,
        CostCenter ,
        LocationId
FROM    dbo.Department
ORDER BY LocationId ,
        DepartmentName ,
        CostCenter DESC;
```

In this query, we sort by the **`LocationId`**, then the **`DepartmentName`**, and then the **`CostCenter`**. The impact of the last column specified in the order by can be seen in the last two records in the following result set.

DepartmentId	Department Name	Cost Center	LocationId
14	Express Marketing	1SNXK5DFDSF	1
6	Outside Accounting	NI04JEDVP8RE	1
15	Service	K4DFDFSGGH	1
4	Service Hardware	FWR9NEF5X	1

DepartmentId	Department Name	Cost Center	LocationId
7	Database	80O3SAS8SP6	2
10	Phone Marketing	I8J5XNIYOE9KF	2
9	TechnicalSales	SU476Y3GFGF	2
1	Web Marketing	E5Z3BSG80R1	2
5	Inside Accounting	VVCELPQURP	3
11	Web	DZYFWH4JXIJJ	3
2	InternationalSales	IDAP8Q32CB	4
13	Service Software	WGFB3HWMD	4
8	ConsumerSales	1R8JASDAFAD	5
12	Prepaid Customer	DNM1OUVCS	5
3	Prepaid Customer	3SXY6JLYJM1	5

Result Set 4-7: Sorting by multiple columns

Offset and Fetch

OFFSET and **FETCH** clauses can be used to implement a paging strategy. This provides a filter on the data based on the sorting sequence. An example SQL **SELECT** including **OFFSET** and **FETCH** in your **ORDER BY** follows:

Code Listing 4-13: A simple example using OFFSET and FETCH

```
SELECT  EmployeeId ,
        FirstName ,
        LastName ,
        ReportsTo ,
        DepartmentId
FROM    dbo.Employee
ORDER BY DepartmentId ,
        LastName ,
        FirstName
        OFFSET 20 ROWS FETCH NEXT 10 ROWS ONLY;
```

Note: OFFSET and FETCH were not added until SQL Server 2012.

The values for the **OFFSET** and **FETCH** can be numeric constants like you see in Code Listing 4-13, or could be locally declared variables, or even subqueries. You can query the number of records to be returned from a configuration table for configuration settings.

Code Listing 4-14: Pulling the parameters for FETCH from a configuration table

```
SELECT  EmployeeId ,
        FirstName ,
        LastName ,
        ReportsTo ,
        DepartmentId
FROM    dbo.Employee
ORDER BY DepartmentId ,
        LastName ,
        FirstName
        OFFSET 20 ROWS FETCH NEXT ( SELECT  PageSize
                                    FROM    dbo.AppSettings
                                    WHERE   AppSettingID = 1 ) ROWS ONLY;
```

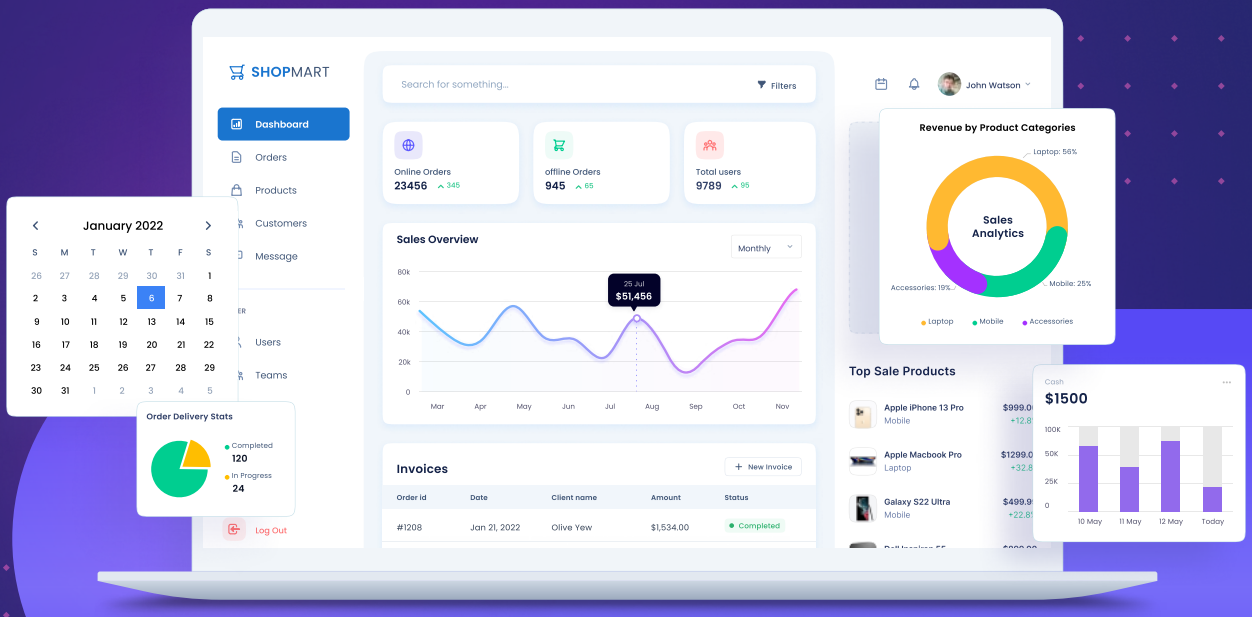
Summary

In this chapter we have seen various ways to slice and summarize data. The aggregate functions open up a lot of possibilities to get fresh insights into your data. We have also seen various ways that sorting can be used to draw the focus to relevant data.

Now we will turn our attention to a new type of table join from what we saw in Chapter 2. Next we will explore the implications of tables selecting from themselves.

THE WORLD'S BEST UI COMPONENT SUITE FOR BUILDING

POWERFUL APPS



GET YOUR FREE .NET AND JAVASCRIPT UI COMPONENTS

syncfusion.com/communitylicense

4.6 out of 5 stars

- 1,700+ components for mobile, web, and desktop platforms
- Support within 24 hours on all business days
- Uncompromising quality
- Hassle-free licensing
- 28000+ customers
- 20+ years in business

Trusted by the world's leading companies

Syncfusion

Chapter 5 Selecting From Yourself

In Chapter 2 we stepped through the various options for joining different tables. We looked at how inner joins, outer joins, and cross joins differ. In this chapter we will look at joining tables to themselves.

Joining the same table multiple times

Sometimes with a normalized data model, you will find that you need to join a table multiple times to get full details. Let's extend the **`Employee`** model to track multiple phone numbers. It doesn't make sense to add a new column for each type of phone number you want to track. Instead, we will add a new table that will have a separate record for each phone number that we want to track. This simplifies adding new types of phone numbers, but querying to get all of the phone numbers can be a bit more involved.

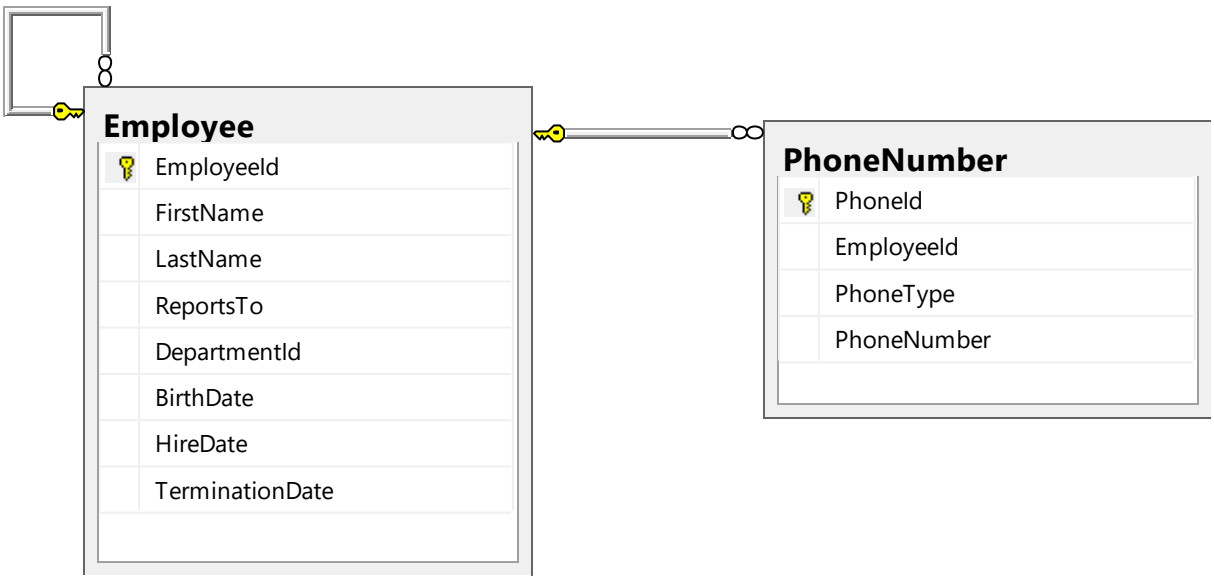


Figure 5-1: Data model for employees and their phone numbers

Selecting various phone numbers

This data model is a good design, and is commonly implemented, but is not without its problems. Our initial attempt to retrieve the home phone number for all employees might look like this:

Code Listing 5-1: Initial join between Employee and PhoneNumber

```
SELECT  Employee.EmployeeId ,
```

```
        FirstName ,
        LastName ,
        PhoneType ,
        PhoneNumber
FROM    dbo.Employee
        JOIN dbo.PhoneNumber ON PhoneNumber.EmployeeId =
Employee.EmployeeId
WHERE   PhoneType = 4
ORDER BY LastName ,
        FirstName
```

Looking through the result set, we can quickly see the problems with this query. As an inner join, employees without a home phone are missing. The other problem is that this data model allows multiple home phone numbers.

Tip: To ensure that an employee does not have duplicate PhoneTypes, you could add a unique constraint on the EmployeeId, PhoneType combination. However, if you already have many duplicates, this may not be an option.

Resolving the first issue is straightforward: convert the inner join to an outer join.

Code Listing 5-2: Outer join between the Employee and PhoneNumber Tables

```
SELECT  Employee.EmployeeId ,
        FirstName ,
        LastName ,
        PhoneType ,
        PhoneNumber
FROM    dbo.Employee
        LEFT OUTER JOIN dbo.PhoneNumber
                ON PhoneNumber.EmployeeId = Employee.EmployeeId
                AND PhoneType = 4
ORDER BY LastName ,
        FirstName
```

Note: We have to also move the PhoneType check from the WHERE clause to the JOIN clause for it to be included as an outer join condition.

Eliminating the redundant phone numbers is a bit more challenging. We want to get a single record for each **`EmployeeId`**. This is where a **`TOP`** expression can be useful. With this expression, we can explicitly specify how many records should be returned in the result set. We want to ensure that there is only a single matching record pulled from the **`PhoneNumber`** table.

For this to work, we need a correlated subquery. This means that the subquery will reference data contained in the outer query. This also means that the subquery will be executed once for every record from the containing query. Otherwise the **`TOP`** expression would not make sense,

and would always return only a single row. This also means that we'll need a new type of **JOIN** logic to join to this subquery.

Code Listing 5-3: Joining Employee and PhoneNumber tables with a CROSS APPLY

```
SELECT  Employee.EmployeeId ,
        FirstName ,
        LastName ,
        PhoneType ,
        PhoneNumber
FROM    dbo.Employee
        CROSS  APPLY ( SELECT TOP 1
                              PhoneNumber ,
                              PhoneType ,
                              EmployeeId
                       FROM      dbo.PhoneNumber
                       WHERE     PhoneType = 4
                                 AND EmployeeId = Employee.EmployeeId ) p
ORDER BY EmployeeId;
```

Note: Anytime you have a subquery like the one in Code Listing 5-3, you must give it an alias so that you can refer to it at other places in the query. In this case, you won't have to refer to it later, but you still must provide an alias; in this case, p.

CROSS APPLY is similar to the **CROSS JOIN** that we used earlier to create a Cartesian product, except we know that we will have only a single record in the second table. This is not exactly what we want either, because any **Employee** records that have no **PhoneNumber** would be excluded from the list. Fortunately, the **CROSS APPLY** has a partner operation called the **OUTER APPLY**, which will allow us to show every record exept the ones missing data from the correlated subquery. Just like outer joins, the missing data will be replaced with NULL values or whatever value is specified as a default for that column.

So our final query to get the **Employee** records along with an associated home phumber, if they have one, is:

Code Listing 5-4: Joining the Employee and PhoneNumber tables with an OUTER APPLY

```
SELECT  Employee.EmployeeId ,
        FirstName ,
        LastName ,
        PhoneType ,
        PhoneNumber
FROM    dbo.Employee
        OUTER   APPLY ( SELECT TOP 1
                              PhoneNumber ,
                              PhoneType ,
                              EmployeeId
                       FROM      dbo.PhoneNumber
```

```
                    WHERE     PhoneType = 4
                              AND EmployeeId = Employee.EmployeeId ) p
ORDER BY EmployeeId;
```

Not too bad, but a query that needs to show an **Employee** record along with any home phone number information may also want to include other phone numbers, such as the cell phone, office phone, fax number, etc. Adding a correlated subquery for each of these types will add a lot of complexity to the final query and make it harder to follow.

This is where a table-valued function can be used. A table-valued function allows us to create a parameterized view or common table expression and returns a Result Set, or in this case, a single record. This means that it can be used anywhere that a table could be used, included as a correlated subquery.

Note: Even though our table-valued function will return a single record, it is still returning a Result Set.

Our table-valued function is easy to define because it looks a lot like our correlated subquery:

Code Listing 5-5: Defining the table-valued function

```
CREATE FUNCTION EmployeePhoneNumberByType
    (
      @EmployeeId INT ,
      @PhoneType INT
    )
RETURNS TABLE
AS
RETURN
    SELECT  TOP 1
            PhoneId ,
            EmployeeId ,
            PhoneType ,
            PhoneNumber
    FROM    dbo.PhoneNumber
    WHERE   EmployeeId = @EmployeeId
            AND PhoneType = @PhoneType;
```

It's also easy to use:

Code Listing 5-6: Using the table-valued function

```
SELECT  Employee.EmployeeId ,
        FirstName ,
        LastName ,
```

```
        Home.PhoneNumber AS Home,
            Office.PhoneNumber AS Office,
            cell.PhoneNumber AS Cell,
            fax.PhoneNumber AS Fax
FROM    dbo.Employee
        OUTER APPLY dbo.EmployeePhoneNumberByType(Employee.EmployeeId, 4)
AS Home
        OUTER APPLY dbo.EmployeePhoneNumberByType(Employee.EmployeeId, 1)
AS Office
        OUTER APPLY dbo.EmployeePhoneNumberByType(Employee.EmployeeId, 2)
AS Cell
        OUTER APPLY dbo.EmployeePhoneNumberByType(Employee.EmployeeId, 3)
AS Fax
ORDER BY EmployeeId;
```

Depending on the amount of data in your database, you might start seeing a problem with this approach at this point.

Note: Because we are calling this table-valued function in four separate OUTER APPLY statements, we are calling it four times for each record that is returned. As the number of records grows, this can be become very expensive.

If you do not have the volume of data that causes such a performance problem, this query may be all that you need, but if you do encounter performance issues, then we may need to get more creative to optimize this query for performance.

Tip: Don't optimize away clarity unless you are having a performance problem. A clever solution that is difficult to follow will be difficult to maintain, and could have errors that are hard to detect or resolve.

To optimize away the need to call a correlated subquery (or worse, four correlated subqueries) for each record, we will need a better way to limit ourselves to a single record without having to use the **TOP** clause. We may have to add a couple of additional subqueries to the mix, but as long as they are not correlated, we will see a nice performance improvement.

Code Listing 5-7: Optimized query eliminating the correlated subquery

```
SELECT  Employee.EmployeeId ,
        FirstName ,
        LastName ,
        PhoneNumber AS HomePhone
FROM    dbo.Employee
        LEFT OUTER JOIN ( ( SELECT  EmployeeId ,
                                    MAX(PhoneId) AS PhoneId
                            FROM    dbo.PhoneNumber
                            WHERE   PhoneType = 4
                            GROUP BY EmployeeId ) HomePhoneId
```

```
                            INNER JOIN dbo.PhoneNumber
                                ON HomePhoneId.PhoneId =
PhoneNumber.PhoneId )
                    ON PhoneNumber.EmployeeId = Employee.EmployeeId;
```

In performance testing on my local database loaded with 10,000 **Employee** records and 6,000 **PhoneNumber** records, the original query to return a home phone number took an average of 3,421 ms. The optimized version took an average of 48 ms.

This is a substantial improvement. Three seconds to run a query is rarely going to be acceptable, especially when it could be brought back down to a fraction of a second.

Now that we have seen that this optimization does actually produce performance gains, let's look at the details for how we work such magic.

Let's start with the nested join in the middle of the query.

Tip: When breaking a query apart, the middle is often the best place to start.

The first part, which we will call **HomePhoneId**, will give us the PhoneId for the last record with a **PhoneType** entered for each employee. This will allow us to filter down to a single record for each employee, but it does not give us the **PhoneNumber**, but rather the primary key to get the full record for that phone number.

Code Listing 5-8: Finding the primary key for the home PhoneNumber for each employee

```
SELECT  EmployeeId ,
        MAX(PhoneId) AS PhoneId
FROM    dbo.PhoneNumber
WHERE   PhoneType = 4
GROUP BY EmployeeId
```

Viewed by itself, we can see that this is a simple query using the **MAX** aggregate function and a single **GROUP BY**. Let's look at a few of the records returned.

EmployeeId	PhoneId
1	334071
2	315909
3	393938
4	393715
5	352673

EmployeeId	PhoneId
6	327152
7	385227
8	342555
9	383116
10	342805

Result Set 5-1: Finding the PhoneId for the home phone for each employee

When looking at this result set, we need to remember that there is no guarantee that the result will include a record for each **`EmployeeId`**. The **`MAX`** aggregate function will reduce multiple records to a single record, but it does not help with employee records with no home phone records.

If there is no record for an **`EmployeeId`** in the **`HomePhoneId`** subquery, we will not have anything to match against in the **`PhoneNumber`** table either. By nesting this join, we are able to deal with the result of the join as a whole outside of the scoping. In this case, we take the result of joining the subquery **`HomePhoneId`** with the **`PhoneNumber`** table, and then do an outer join to the **`Employee`** table. Without the nested join, we would have to do a two-level outer join, which isn't really possible because the first outer join will fill in the blanks with NULL values, leaving nothing to match against for the second-level join.

The nested join allows us to avoid this problem by treating both joins as one.

General selecting trees and graphs

Trees and *graphs* are the general names we give to any hierarchical table structures. This is a table that has a foreign key back to itself. As long as there are no cycles in the relationships, we call it a tree. If there are cycles in the relationships, it is a graph.

Both types of data structures have many uses and show up in various scenarios. We have already seen one in the **`Employee`** table. It includes a reference back to itself to show who each employee reports to.

Note: A hierarchical table may be used to track all manner of hierarchical data from org charts to sales hierarchies, to operational hierarchies, to nested questions on a questionnaire, or to a bill of materials. Many of these techniques are useful regardless of the purpose of the data stored.

Classic organization chart

Let's revisit the **Employee** table, this time paying attention to the **ReportsTo** column that we have ignored so far.

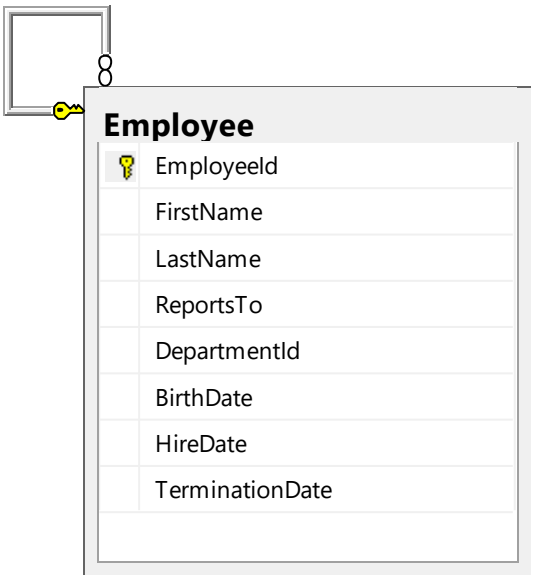


Figure 5-2: Revisiting the Employee table with the ReportsTo reference

The **ReportsTo** column forms a foreign key back to the **EmployeeId** for another record. If it's properly structured, only one record should be missing a value for the **ReportsTo** column. In most organizations, everyone has a boss except for the one at the top of the chart.

We can easily track any records that have been orphaned by earlier **DELETE** statements, leaving them with no one to report to.

Code Listing 5-9: Finding orphaned records

```
SELECT  EmployeeId ,
        FirstName ,
        LastName ,
        ReportsTo ,
        DepartmentId ,
        BirthDate ,
        HireDate ,
        TerminationDate
FROM    dbo.Employee
WHERE   ReportsTo IS NULL
        AND ( TerminationDate > GETDATE()
              OR TerminationDate IS NULL );
```

If an employee has been terminated, you may or may not care about who they once reported to.

Tip: Run a query like this periodically to track potential data integrity issues as soon as possible. If an Employee is missing its ReportsTo value, we need to supply a placeholder for tracking purposes.

Who's the boss?

To get the immediate supervisor for any given employee is relatively simple. Join the **`Employee`** table back to itself, aliasing one of them as **`Manager`**, and one of them as **`Employee`**.

Code Listing 5-10: Joining Manager to Employee

```
SELECT  Employee.EmployeeId ,
        Employee.FirstName EmployeeFirstName,
        Employee.LastName EmployeeLastName,
        Manager.EmployeeId ManagerId,
        Manager.FirstName ManagerFirstName,
        Manager.LastName ManagerLastName
FROM    dbo.Employee
        INNER JOIN dbo.Employee Manager ON Manager.EmployeeId =
Employee.ReportsTo
WHERE   Employee.TerminationDate IS NULL
ORDER BY ManagerLastName ,
        ManagerFirstName ,
        EmployeeLastName ,
        EmployeeFirstName;
```

Note: Anytime you reference the same table more than once, you will need to give each reference a unique name through an alias.

Sometimes you need more details than just the immediate supervisor. You may need to track the level a particular employee is in the hierarchy, see how many levels separate two employees, or report on more than two levels at a time.

In each of these cases, we need to take advantage of the recursive nature of hierarchical tables. Recursive common table expressions (CTEs) will allow us to exploit this power.

A recursive common table expression will have three components:

- **Anchor**: This will be the initial call to the query composing the CTE. The anchor will generally invoke the recursive call through a **`UNION`** statement.
- **Recursive call**: This will be a subsequent query that references the CTE being defined.
- **Termination check**: This check is often implied. The recursive calls end when no records are returned from the previous call, which requires that there not be any loops in the data. For example, we will have problems if an employee reports to themselves or to one of their subordinates. We will see shortly how to detect this problem and avoid it.

Now let's put these pieces together to show the direct reports and their level in the hierarchy.

Code Listing 5-11: Recursive CTE showing direct reports

```
WITH    DirectReports
          AS (
-- Anchor member definition
               SELECT   e.ReportsTo ,
                        e.EmployeeId ,
                        e.FirstName ,
                        e.LastName ,
                        CAST (' ' AS VARCHAR(50)) ManagerFirstName ,
                        CAST (' ' AS VARCHAR(50)) ManagerLastName ,
                        0 AS Level
               FROM     dbo.Employee AS e
               WHERE    e.ReportsTo IS NULL
               UNION ALL
-- Recursive member definition
               SELECT   e.ReportsTo ,
                        e.EmployeeId ,
                        e.FirstName ,
                        e.LastName ,
                        m.FirstName ,
                        m.LastName ,
                        d.Level + 1
               FROM     dbo.Employee AS e
                        INNER JOIN dbo.Employee AS m ON e.ReportsTo =
m.EmployeeId
                        INNER JOIN DirectReports AS d ON e.ReportsTo =
d.EmployeeId )
    SELECT  DirectReports.ReportsTo ,
            DirectReports.EmployeeId ,
            DirectReports.FirstName ,
            DirectReports.LastName ,
            DirectReports.ManagerFirstName ,
            DirectReports.ManagerLastName ,
            DirectReports.Level
    FROM    DirectReports
      ORDER BY LEVEL, managerlastname, lastname
```

ReportsTo	EmployeeId	First Name	Last Name	Manager First Name	Manager Last Name	Level
NULL	1	Nicole	Bartlett			0
1	13	Katina	Archer	Nicole	Bartlett	1

ReportsTo	EmployeeId	First Name	Last Name	Manager First Name	Manager Last Name	Level
1	2051	Gilberto	Arroyo	Nicole	Bartlett	1
1	4	Darnell	Calderon	Nicole	Bartlett	1
1	12	Lindsay	Conner	Nicole	Bartlett	1
1	7276	Elijah	Cruz	Nicole	Bartlett	1
1	4151	Blake	Duarte	Nicole	Bartlett	1
1	9	Daphne	Dudley	Nicole	Bartlett	1
1	5	Desiree	Farmer	Nicole	Bartlett	1
1	6	Holly	Fernandez	Nicole	Bartlett	1

Result Set 5-2: Who's the boss?

We can also incorporate the virtual **`Level`** column into the filters:

Code Listing 5-12: Filtering on the level

```
    SELECT  DirectReports.ReportsTo ,
            DirectReports.EmployeeId ,
            DirectReports.FirstName ,
            DirectReports.LastName ,
            DirectReports.ManagerFirstName ,
            DirectReports.ManagerLastName ,
            DirectReports.Level
    FROM    DirectReports
      WHERE DirectReports.Level BETWEEN 7 AND 9
      ORDER BY LEVEL, managerlastname, lastname
```

In addition to tracking the level, we may often want to get the full path to an individual employee. This is useful information to display and use, but more importantly, we can use it to detect cycles in our data:

Code Listing 5-13: Hierarchy showing the full path through the hierarchy

```
WITH    EmployeeReportingPath
          AS (
-- Anchor member definition
               SELECT   e.ReportsTo ,
                        e.EmployeeId ,
                        e.FirstName ,
                        e.LastName ,
```

```
                        '.' + CAST(e.EmployeeId AS VARCHAR(99)) + '.' AS
[Path] ,
                        0 AS Level
                FROM    dbo.Employee AS e
                WHERE   e.ReportsTo IS NULL
                UNION ALL
-- Recursive member definition
                SELECT  e.ReportsTo ,
                        e.EmployeeId ,
                        e.FirstName ,
                        e.LastName ,
                        CAST(d.Path + '.'
                            + CAST(e.EmployeeId AS VARCHAR(100)) AS
VARCHAR(100))
                            + '.' AS [Path] ,
                        d.Level + 1
                FROM    dbo.Employee AS e
                        INNER JOIN EmployeeReportingPath AS d
                            ON e.ReportsTo = d.EmployeeId )
    SELECT  EmployeeReportingPath.ReportsTo ,
            EmployeeReportingPath.EmployeeId ,
            EmployeeReportingPath.FirstName ,
            EmployeeReportingPath.LastName ,
            EmployeeReportingPath.Level ,
            EmployeeReportingPath.Path
    FROM    EmployeeReportingPath
    ORDER BY EmployeeReportingPath.Level ,
            EmployeeReportingPath.LastName;
```

This will produce output like the following:

Reports To	Employee Id	First Name	Last Name	Level	Path
114	549	Donnie	Calderon	3	.1.12.114.549.
114	550	Louis	Collier	3	.1.12.114.550.
114	548	Nathaniel	Dickson	3	.1.12.114.548.
114	543	Jeannie	Franklin	3	.1.12.114.543.
114	552	Frankie	Glenn	3	.1.12.114.552.
114	551	Robert	Pacheco	3	.1.12.114.551.
114	544	Dorothy	Parrish	3	.1.12.114.544.

Reports To	Employee Id	First Name	Last Name	Level	Path
114	541	Micheal	Potts	3	.1.12.114.541.
114	547	Spencer	Rowe	3	.1.12.114.547.
114	545	Lillian	Shepherd	3	.1.12.114.545.

Result Set 5-3: Org chart hierarchy showing the full path to an employee

If a number is repeated in the **Path** column, then there is a cycle, and it will eventually lead to problems. We can add a **CASE** statement to the output to make these easier to spot.

Code Listing 5-14: Hierarchy query flagging cycles

```
DECLARE @root INT;
SET @root = 5095;
WITH    EmployeeReportingPath
          AS (
-- Anchor member definition
               SELECT   e.ReportsTo ,
                        e.EmployeeId ,
                        e.FirstName ,
                        e.LastName ,
                        '.' + CAST(e.EmployeeId AS VARCHAR(99)) + '.' AS
[Path] ,
                        0 AS Level ,
                        0 AS Cycle
               FROM     dbo.Employee AS e
               WHERE    -- e.EmployeeId = @root--
                        e.ReportsTo IS NULL
               UNION ALL
-- Recursive member definition
               SELECT   e.ReportsTo ,
                        e.EmployeeId ,
                        e.FirstName ,
                        e.LastName ,
                        CAST(d.Path + '.'
                            + CAST(e.EmployeeId AS VARCHAR(100)) AS
VARCHAR(100))
                            + '.' AS [Path] ,
                        d.Level + 1 ,
                        CASE WHEN d.Path LIKE '%.'
                                  + CAST(e.EmployeeId AS VARCHAR(10)) +
'.%'
                             THEN 1
                             ELSE 0
                        END
               FROM     dbo.Employee AS e
```

```
                        INNER JOIN EmployeeReportingPath AS d
                              ON e.ReportsTo = d.EmployeeId
                WHERE     d.Cycle = 0
                )
    SELECT  EmployeeReportingPath.ReportsTo ,
            EmployeeReportingPath.EmployeeId ,
            EmployeeReportingPath.FirstName ,
            EmployeeReportingPath.LastName ,
            EmployeeReportingPath.Level ,
            EmployeeReportingPath.Path ,
            EmployeeReportingPath.Cycle
    FROM    EmployeeReportingPath
    WHERE   EmployeeReportingPath.Cycle = 1
    ORDER BY
            EmployeeReportingPath.Level ,
            EmployeeReportingPath.LastName;
```

In this latest query, we have added a new column called **`Cycle`** that will have a value of 0 if there is not a cycle detected, or a value of 1 if there was a cycle detected. You can see that we instructed the recursive call not to continue going down a path when a cycle has been detected. Finally, in the **`WHERE`** clause for the **`CALLING`** query, we add a filter to show only the records that were found with a cycle. We can walk back through the **`Path`** for any cycles detected to track where the cycle forms and reroute the employees as appropriate.

Cleaning up this data can often lead to another common problem with hierarchical data: If there is not a path from an **`Employee`** record back to the root of the hierarchy, then the **`Employee`** record will not be included in these results. Missing data is harder to track down because we don't know what we don't see. Fortunately, we can identify the missing records using some set concepts:

Code Listing 5-15: Finding the Employee records missing from the hierarchy view

```
WITH    EmployeeReportingPath
          AS (
-- Anchor member definition
                SELECT   e.ReportsTo ,
                         e.EmployeeId ,
                         e.FirstName ,
                         e.LastName ,
                         CAST(e.EmployeeId AS VARCHAR(100)) AS [Path] ,
                         0 AS Level ,
                         0 AS Cycle
                FROM     dbo.Employee AS e
                WHERE    e.ReportsTo IS NULL
                UNION ALL
-- Recursive member definition
                SELECT   e.ReportsTo ,
                         e.EmployeeId ,
```

```
                        e.FirstName ,
                        e.LastName ,
                        CAST(d.Path + '.' + CAST(e.EmployeeId AS
VARCHAR(100))
                        + '.' AS VARCHAR(100)) AS [Path] ,
                        d.Level + 1 ,
                        CASE WHEN d.Path LIKE '%.'
                                  + CAST(e.EmployeeId AS VARCHAR(10)) +
'.%'
                             THEN 1
                             ELSE 0
                        END
                FROM    dbo.Employee AS e
                        INNER JOIN EmployeeReportingPath AS d
                             ON e.ReportsTo = d.EmployeeId
                WHERE   d.Cycle = 0
                )
    SELECT  EmployeeId ,
            FirstName ,
            LastName ,
            ReportsTo
    FROM    dbo.Employee
    WHERE   EmployeeId NOT IN ( SELECT  EmployeeReportingPath.EmployeeId
                                FROM    EmployeeReportingPath )
    ORDER BY LastName ,
            FirstName;
```

This will show us employee details for any **`EmployeeId`** not included in the recursive query.

Employee Id	First Name	Last Name	Reports To
114	Brendan	Ramsey	114
541	Micheal	Potts	114
542	Deborah	Vincent	114
543	Jeannie	Franklin	114
544	Dorothy	Parrish	114
545	Lillian	Shepherd	114
546	Carla	Villanueva	114
547	Spencer	Rowe	114
548	Nathaniel	Dickson	114

Employee Id	First Name	Last Name	Reports To
549	Donnie	Calderon	114

Result Set 5-4: Employees missing from the hierarchy view

In this case, the problem is that Employee 114 is flagged as reporting to himself, so there is no path that leads to him. This also means that everyone reporting to this employee is also unreachable.

There's one final trick that's helpful when dealing with hierarchical data. Sometimes it may be useful to see the hierarchy not from the root, but from a particular node in the resulting tree. For example, we may want to see the hierarchy details for all the employees who report to this Employee 114 who was causing such a problem.

Code Listing 5-16: Starting the hierarchy below the root

```
DECLARE @root INT;
SET @root = 114;
WITH    EmployeeReportingPath
          AS (
-- Anchor member definition
               SELECT   e.ReportsTo ,
                        e.EmployeeId ,
                        e.FirstName ,
                        e.LastName ,
                        '.' + CAST(e.EmployeeId AS VARCHAR(99)) + '.' AS
[Path] ,
                        0 AS Level ,
                        0 AS Cycle
               FROM     dbo.Employee AS e
               WHERE     e.EmployeeId = @root
               UNION ALL
-- Recursive member definition
               SELECT   e.ReportsTo ,
                        e.EmployeeId ,
                        e.FirstName ,
                        e.LastName ,
                        CAST(d.Path
                        +  CAST(e.EmployeeId AS VARCHAR(100)) AS
VARCHAR(100))
                        + '.' AS [Path] ,
                        d.Level + 1 ,
                        CASE WHEN d.Path LIKE '%.'
                                  + CAST(e.EmployeeId AS VARCHAR(10)) +
'.%'
                             THEN 1
                             ELSE 0
                        END
               FROM     dbo.Employee AS e
```

```
                    INNER JOIN EmployeeReportingPath AS d
                               ON e.ReportsTo = d.EmployeeId
            WHERE   d.Cycle = 0
            )
SELECT  EmployeeReportingPath.ReportsTo ,
        EmployeeReportingPath.EmployeeId ,
        EmployeeReportingPath.FirstName ,
        EmployeeReportingPath.LastName ,
        EmployeeReportingPath.Level ,
        EmployeeReportingPath.Path ,
        EmployeeReportingPath.Cycle
FROM    EmployeeReportingPath
ORDER BY
        EmployeeReportingPath.Level ,
        EmployeeReportingPath.ReportsTo,
        EmployeeReportingPath.LastName
```

All we have to do is change the anchor in the recursive CTE to start with a specific **`EmployeeId`** rather than starting where the **`ReportsTo`** is NULL.

Summary

In this chapter, we have focused on what happens when a table joins against itself. We saw a practical application of **`CROSS APPLY`** and **`OUTER APPLY`**. We looked at an advanced optimization of a SQL query as we explored the problem of showing multiple phone numbers for each employee.

We also spent some time exploring hierarchical data and how to deal with some of the common problems that pop up when dealing with hierarchical data, such as orphaned records and cycles in data. We have seen recursive common table expressions in action and how to derive extra data from these queries, such as tracking the level and building the full path to a specific record.

Hierarchical data shows up in many places, and it is important to understand how to query and maintain this data.

Chapter 6 It's About Time

Much of our world revolves around time, not to mention that so much of our business logic often depends on dates and times. It seems strange that we have so little support for dates and times in our database.

Turns out we actually have a bit more support than most people realize, and with a bit of effort on our part, we can build even more.

Understanding the Date and Time data types

If you look at many databases, it's clear that most people are only aware of one data type for representing dates and times, the **`DateTime`** data type. In many situations, this is not the best data type to use.

DateTime

The **`DateTime`** data type is still the data type most commonly used to represent date and time values. It is far from the best, but often may be required for compatibility with other systems or languages. Columns of this type can represent dates in the range of January 1, 1753, through December 31, 9999, and can represent time in the range of 00:00:00 through 23:59:59.997. For most business applications, that's probably more accurate than you need; we generally don't care about three decimal places for the seconds. Often we don't care about time at all, but we have to allocate space to store it anyway, and worry about stripping it out when it's not needed.

Most importantly, this data type is not ANSI- or ISO-compliant.

DateTime2

`DateTime2` is an extension to **`DateTime`** to bring it up to ANSI standards. It has a larger date range as well as a larger time range with a larger default precision, and an option for a user-defined fractional precision. Columns of this type can represent dates in the range of January 1, 0001 through December 31, 9999. They can represent time in the range of 00:00:00 through 23:59:59.9999999. This makes them precise to within 100 nanoseconds.

Tip: Any new columns added should use DateTime2 over DateTime.

This data type was not introduced in SQL Server until 2008 R2, so if you're running an older database, you will still need to use **`DateTime`**.

Date

The **Date** data type was also introduced in SQL 2008 R2. It is equivalent to the **Date** portion of a **DateTime2** column, so it has a date range of January 1, 0001 through December 31, 9999. This is useful for cases where you don't care about the time, but only the date.

Time

The **Time** data type was also introduced in SQL 2008 R2. It is equivalent to the **Time** portion of a **DateTime2** column, so it has a time range of 00:00:00.0000000 through 23:59:59.9999999. In some cases, you may only care about the time outside of the context of an associated date.

DateTimeOffset

The **DateTimeOffset** data type is similar to the **DateTime2** data type, except it is also aware of time zones. It has all the same ranges as the **DateTime2** data type.

Common functions

Getting dates

We have several functions to get the current date and time.

- **SYSDATETIME(),**
- **SYSDATETIMEOFFSET(),**
- **SYSUTCDATETIME(),**
- **CURRENT_TIMESTAMP,**
- **GETDATE(),**
- **GETUTCDATE();**

You can see the subtle differences between these various functions by running the following SQL statement:

Code Listing 6-1: Date functions to get the current date

```
SELECT  SYSDATETIME() AS SYSDATETIME ,
        SYSDATETIMEOFFSET() AS SYSDATETIMEOFFSET ,
        SYSUTCDATETIME() AS SYSUTCDATETIME ,
        CURRENT_TIMESTAMP AS [CURRENT_TIMESTAMP] ,
        GETDATE() AS GETDATE ,
        GETUTCDATE() AS GETUTCDATE;
```

Getting parts of a date

We also have several functions that can be used to pull out part of the date:

- **DATENAME (datepart, date)**
- **DATEPART (datepart, date)**
- **DAY (date)**
- **MONTH (date)**
- **YEAR (date)**

Code Listing 6-2: Date functions to get parts of a date

```
DECLARE @TestDate DATETIME2;
SELECT  @TestDate = '7/4/2025';

SELECT  DATENAME(WEEK, @TestDate) [Week] ,
        DATEPART(DAYOFYEAR, @TestDate) [DayOfYear] ,
        DAY(@TestDate) [Day] ,
        MONTH(@TestDate) [Month] ,
        YEAR(@TestDate) [Year];
```

The output of this query should look like this:

Week	Day of Year	Day	Month	Year
27	185	4	7	2025

Result Set 6-1: The parts of a date

When you see **DATEPART**, it can have any of these values:

Table 6-1

Day	nanosecond
Dayofyear	quarter
Hour	second
ISO_WEEK	TZoffset
Microsecond	week
Millisecond	weekday
Minute	year
Month	

Code Listing 6-3: Naming the parts of the date

```
DECLARE @TestDate DATETIME2;
SELECT  @TestDate = '7/4/2025';

SELECT DATENAME(DAY, @TestDate) [Day],
       DATENAME(dayofyear, @TestDate)[DOY],
       DATENAME(hour, @TestDate) [H],
       DATENAME(microsecond, @TestDate) [Micro],
       DATENAME(millisecond, @TestDate) [MS],
       DATENAME(minute, @TestDate) [Min],
       DATENAME(month, @TestDate) [Month],
       DATENAME(nanosecond, @TestDate) [Ns],
       DATENAME(quarter, @TestDate) [Quart],
       DATENAME(second, @TestDate) [Sec],
       DATENAME(TZoffset, @TestDate) [Offset],
       DATENAME(week, @TestDate) [Week],
       DATENAME(weekday, @TestDate) [weekday],
       DATENAME(year, @TestDate) [Year]
```

Day	DOY	H	Micro	Ms	Min	Month	Ns	Quart	Sec	Offset	Week	Weekday	Year
4	185	0	0	0	0	July	0	3	0	+00:00	27	Friday	2025

Result Set 6-2: Naming the parts of a date

Difference between two dates

The **DATEDIFF** function takes a **DATEPART** and two dates and will return the difference between the two dates. So we can find the difference between two dates in any of the units we saw earlier for the **DATEPART**.

Code Listing 6-4: Differences between dates

```
DECLARE @EndDate DATETIME2;
SELECT  @EndDate = '7/4/2025';
DECLARE @StartDate DATETIME2;
SELECT  @StartDate = '1/1/2025';


SELECT  DATEDIFF(DAY, @StartDate, @EndDate) [Day] ,
        DATEDIFF(DAYOFYEAR, @StartDate, @EndDate) [Day of Year] ,
        DATEDIFF(HOUR, @StartDate, @EndDate) [hour] ,
        DATEDIFF(MINUTE, @StartDate, @EndDate) [minute] ,
        DATEDIFF(MONTH, @StartDate, @EndDate) [month] ,
        DATEDIFF(QUARTER, @StartDate, @EndDate) [quarter] ,
        DATEDIFF(SECOND, @StartDate, @EndDate) [second] ,
```

```
        DATEDIFF(WEEK, @StartDate, @EndDate) [week] ,
        DATEDIFF(WEEKDAY, @StartDate, @EndDate) [weekday] ,
        DATEDIFF(YEAR, @StartDate, @EndDate) [year];
```

Day	Day of Year	Hour	Minute	Month	Quarter	Second	Week
184	184	4416	264960	6	2	15897600	26

Result Set 6-3: Difference between dates

Note: Depending on how big the difference is, we can easily get an overflow error for some data parts, especially for the smaller date parts. In general, we probably won't care about the number of seconds, let alone nanoseconds between the OriginationDate and ClosingDate for a 30-year mortgage.

Changing the value of a date

We can also change the value of a date using the **DATEADD** function. Once again, this function uses the **DATEPART** logic to express which part of the date to add. We can combine the **DATEDIFF** logic we just looked at with **DATEADD** function to push the **@StartDate** to the **@EndDate**:

Code Listing 6-5: Effects of the DATEADD

```
DECLARE @EndDate DATETIME2;
SELECT  @EndDate = '7/4/2025';
DECLARE @StartDate DATETIME2;
SELECT  @StartDate = '1/1/2025';
SELECT  DATEADD(DAY, offset.Day, @StartDate) Day ,
        DATEADD(DAYOFYEAR, offset.[Day of Year], @StartDate) DOY ,
        DATEADD(HOUR, offset.hour, @StartDate) H ,
        DATEADD(MINUTE, offset.minute, @StartDate) M ,
        DATEADD(MONTH, offset.month, @StartDate) Month ,
        DATEADD(QUARTER, offset.quarter, @StartDate) Q ,
        DATEADD(SECOND, offset.second, @StartDate) Sec ,
        DATEADD(WEEK, offset.week, @StartDate) W ,
        DATEADD(WEEKDAY, offset.weekday, @StartDate) WD ,
        DATEADD(YEAR, offset.year, @StartDate) Y
FROM    ( SELECT    DATEDIFF(DAY, @StartDate, @EndDate) [Day] ,
                    DATEDIFF(DAYOFYEAR, @StartDate, @EndDate) [Day of
Year] ,
                    DATEDIFF(HOUR, @StartDate, @EndDate) [hour] ,
                    DATEDIFF(MINUTE, @StartDate, @EndDate) [minute] ,
                    DATEDIFF(MONTH, @StartDate, @EndDate) [month] ,
                    DATEDIFF(QUARTER, @StartDate, @EndDate) [quarter] ,
```

```
                    DATEDIFF(SECOND, @StartDate, @EndDate) [second] ,
                    DATEDIFF(WEEK, @StartDate, @EndDate) [week] ,
                    DATEDIFF(WEEKDAY, @StartDate, @EndDate) [weekday] ,
                    DATEDIFF(YEAR, @StartDate, @EndDate) [year] ) offset;
```

The end result from this query displays July 4, 2025, for each column.

Date-based business logic

Business logic is often based on dates. Sometimes it might be based on specific actions taking place during promotional windows, and sometimes it might revolve around honoring business logic that was in effect when the process was initiated. You might need to honor rates from when the loan was originated, for example. You might need to honor shipping costs based on when the order was placed or processed, or you might need to evaluate the turn time between key business transactions.

Sometimes date logic is based on calendar days, and sometimes it's based on business days. We need to clearly understand the difference. A calendar day is any day listed on a calendar. For such logic, the standard, built-in functions are all you need. An approval being good for 30 days means 30 calendar days, so you simply add 30 days to the approval date to get the expiration date. However, requiring that a specific document be generated in three business days is another matter.

A business day refers to days when business is conducted. Generally speaking, this excludes weekends and federal holidays. If you're in a different county, the list of holidays will be different. For many holidays, the actual date may change from year to year, or at least the dates the holidays fall on will change. Also, if the holiday falls on a weekend, we sometimes celebrate it on the following Monday or the preceding Friday, and you don't even want to think about the logic to track Easter.

Whenever we have this level of ambiguity, it's nice to have a lookup table. We need a calendar reference table that will allow us to easily track some useful attributes for dates. Also, because the logic for determining holidays is tricky, we want to be able to easily store the holidays in our **`Calendar`** table and know that they are not business days.

Our **`Calendar`** table might look like this:

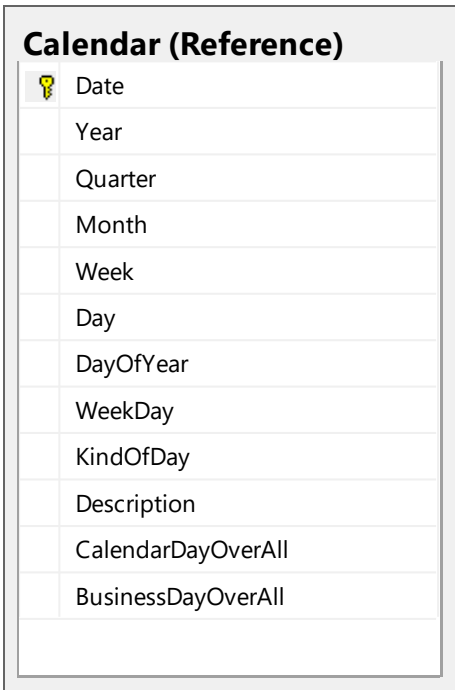


Figure 6-1: Calendar table

To populate this table, we will see some of the date functions in action, as well as some other tricks that we've learned along the way.

For our first trick, we will use a recursive CTE to get a list of every date between January 1, 2000, and December 31, 2050.

Code Listing 6-6: Using a recursive CTE to get a list of dates

```
WITH    Dates ( Date )
          AS ( SELECT   CAST('2000' AS DATETIME) Date
               UNION ALL
               SELECT   ( Date + 1 ) AS Date
               FROM     Dates
               WHERE    Date < CAST('2051' AS DATETIME) - 1
             )
    SELECT  *
    FROM    Dates
OPTION  ( MAXRECURSION 0 );
```

The **MAXRECURSION** option at the end tells our CTE not to worry about how many recursive calls this is. Normally, the database will keep up with this and complain if the recursive calls go past a set limit as a way of preventing cycles in the recursive calls. Here we know that the recursive calls will end about just under 19,000, and that's not a problem. Setting **MAXRECURSION** to 0 tells the database not to worry about it.

Next we want a CTE to track the date for the Thursday for each of these dates. We do this because we will count weeks by counting the Thursdays.

Code Listing 6-7: Finding the Thursday associated with each Date

```
WITH    Dates ( Date )
          AS ( SELECT   CAST('2000' AS DATETIME) Date
               UNION ALL
               SELECT   ( Date + 1 ) AS Date
               FROM     Dates
               WHERE    Date < CAST('2051' AS DATETIME) - 1
             ),
        DatesAndThursdayInWeek ( Date, Thursday )
          -- The weeks can be found by counting the Thursdays
          -- in a year so we find
          -- the Thursday in the week for a particular date.
          AS ( SELECT   Date ,
                        CASE DATEPART(WEEKDAY, Date)
                          WHEN 1 THEN Date + 3
                          WHEN 2 THEN Date + 2
                          WHEN 3 THEN Date + 1
                          WHEN 4 THEN Date
                          WHEN 5 THEN Date - 1
                          WHEN 6 THEN Date - 2
                          WHEN 7 THEN Date - 3
                        END AS Thursday
               FROM     Dates
             )
    SELECT  *
    FROM    DatesAndThursdayInWeek
```

Next we will partition this data by **`Year`** and track how many weeks into the year each **`Date`** is:

Code Listing 6-8: Calculate the week number for each Thursday

```
WITH    Dates ( Date )
          AS ( SELECT   CAST('2000' AS DATETIME) Date
               UNION ALL
               SELECT   ( Date + 1 ) AS Date
               FROM     Dates
               WHERE    Date < CAST('2051' AS DATETIME) - 1
             ),
        DatesAndThursdayInWeek ( Date, Thursday )
          -- The weeks can be found by counting the Thursdays
          -- in a year so we find
          -- the Thursday in the week for a particular date.
          AS ( SELECT   Date ,
                        CASE DATEPART(WEEKDAY, Date)
                          WHEN 1 THEN Date + 3
                          WHEN 2 THEN Date + 2
                          WHEN 3 THEN Date + 1
                          WHEN 4 THEN Date
                          WHEN 5 THEN Date - 1
```

```
                          WHEN 6 THEN Date - 2
                          WHEN 7 THEN Date - 3
                        END AS Thursday
              FROM      Dates
            ),
        Weeks ( Week, Thursday )
          AS ( SELECT   ROW_NUMBER() OVER
                  ( PARTITION BY YEAR(Date) ORDER BY Date ) Week ,
                        Thursday
              FROM      DatesAndThursdayInWeek
              WHERE     DATEPART(WEEKDAY, Date) = 4
            )
    SELECT  *
    FROM    Weeks
```

Now that we have all of our CTEs defined, we are ready to use them to gather all of the data for each of these dates:

Code Listing 6-9: Building out details for each Date

```
WITH    Dates ( Date )
          AS ( SELECT   CAST('1999' AS DATETIME) Date
               UNION ALL
               SELECT   ( Date + 1 ) AS Date
               FROM     Dates
               WHERE    Date < CAST('2026' AS DATETIME) - 1
             ),
        DatesAndThursdayInWeek ( Date, Thursday )
          -- The weeks can be found by counting the
          -- Thursdays in a year so we find
          -- the Thursday in the week for a particular date.
          AS ( SELECT   Date ,
                        CASE DATEPART(WEEKDAY, Date)
                          WHEN 1 THEN Date + 3
                          WHEN 2 THEN Date + 2
                          WHEN 3 THEN Date + 1
                          WHEN 4 THEN Date
                          WHEN 5 THEN Date - 1
                          WHEN 6 THEN Date - 2
                          WHEN 7 THEN Date - 3
                        END AS Thursday
               FROM     Dates
             ),
        Weeks ( Week, Thursday )
          AS ( SELECT   ROW_NUMBER() OVER
                  ( PARTITION BY YEAR(Date) ORDER BY Date ) Week ,
                        Thursday
               FROM     DatesAndThursdayInWeek
               WHERE    DATEPART(WEEKDAY, Date) = 4
             )
```

```
    SELECT  d.Date ,
            YEAR(d.Date) AS Year ,
            DATEPART(QUARTER, d.Date) AS Quarter ,
            MONTH(d.Date) AS Month ,
            w.Week as Week,
            DAY(d.Date) AS Day ,
            DATEPART(DAYOFYEAR, d.Date) AS DayOfYear ,
            DATEPART(WEEKDAY, d.Date) AS Weekday ,
            YEAR(d.Date) AS Fiscal_Year ,
            DATEPART(QUARTER, d.Date) AS Fiscal_Quarter ,
            MONTH(d.Date) AS Fiscal_Month ,
            CASE WHEN DATEPART(WEEKDAY, d.Date) = 6 THEN 'Saturday'
                 WHEN DATEPART(WEEKDAY, d.Date) = 7 THEN 'Sunday'
                 ELSE 'BusinessDay'
            END KindOfDay ,
            ''
    FROM    DatesAndThursdayInWeek d
            INNER JOIN Weeks w ON d.Thursday = w.Thursday
OPTION  ( MAXRECURSION 0 );
```

Wrap the **SELECT** with an **INSERT** into the **Calendar** table and we have it all populated. Once populated, you can go through and explicitly switch the **KindOfDay** from **BusinessDay** to **Holiday** for any holidays that you want to track.

Date	Quarter	Month	Week	Day	DayOfYear	Weekday	KindOfDay
2019-12-25	4	12	52	25	359	3	BusinessDay
2019-12-26	4	12	52	26	360	4	BusinessDay
2019-12-27	4	12	52	27	361	5	BusinessDay
2019-12-28	4	12	52	28	362	6	Saturday
2019-12-29	4	12	52	29	363	7	Sunday
2019-12-30	4	12	1	30	364	1	BusinessDay
2019-12-31	4	12	1	31	365	2	Holiday
2020-01-01	1	1	1	1	1	3	Holiday
2020-01-02	1	1	1	2	2	4	BusinessDay
2020-01-03	1	1	1	3	3	5	BusinessDay
2020-01-04	1	1	1	4	4	6	Saturday

Date	Quarter	Month	Week	Day	DayOfYear	Weekday	KindOfDay
2020-01-05	1	1	1	5	5	7	Sunday

Result Set 6-4: Dates with their details

Once we have entered the holidays that we want to track, we are ready to update the final two columns, **`CalendarDayOverall`** and **`BusinessDayOverall`**. Whenever we update a holiday, we need to update the **`BusinessDayOverall`** as well.

We can update the **`CalendarDayOverall`** with a query like this:

Code Listing 6-10: Initializing the CalendarDayOverAll

```
UPDATE  c
SET     c.CalendarDayOverAll = cdo.cdo
FROM    dbo.Calendar c
        INNER JOIN ( SELECT Date ,
                            ROW_NUMBER() OVER ( ORDER BY Date ) AS cdo
                     FROM   dbo.Calendar ) cdo ON cdo.Date = c.Date;
```

Updating the **`BusinessDayOverall`** is a bit more complex. For each record, we want to add up all of the business days that have come before it.

Code Listing 6-11: Initializing the BusinessDayOverAll

```
UPDATE c
SET c.BusinessDayOverAll = bdo.BusinessDayOverAll
FROM Reference.Calendar c
INNER JOIN (
SELECT  Date ,
        BusinessDayOverAll = ( SELECT  Sum(CASE c2.KindOfDay
                                 WHEN 'BuninessDay' THEN 1
                                 ELSE 0
                               END)
                FROM    reference.Calendar c2
                WHERE   c2.Date <= c1.Date)
FROM    reference.Calendar c1) bdo ON bdo.Date = c.Date
```

Now we can use this table to evaluate **`BusinessDay`** logic.

Evaluating right of rescission

When you refinance your mortgage, you get a small window to back out of the new mortgage. This is known as the right of rescission. Your right of rescission ends three business days after closing, so we often need to determine the date three business days after the closing date. With the **`Calendar`** table, we can now do this with a simple query.

Code Listing 6-12: Finding three business days into the future

```
SELECT  Closing.Date ClosingDate ,
        recission.Date RecissionDate ,
        Closing.BusinessDayOverAll ,
        recission.BusinessDayOverAll
FROM    Reference.Calendar Closing
        INNER JOIN ( SELECT MIN(Date) Date ,
                            BusinessDayOverAll
                     FROM   Reference.Calendar
                     GROUP BY BusinessDayOverAll ) recission
       ON recission.BusinessDayOverAll = Closing.BusinessDayOverAll + 3;
```

ClosingDate	RecissionDate	BusinessDayOverAll	BusinessDayOverAll
2019-09-29	2019-10-02	5411	5414
2019-09-30	2019-10-03	5412	5415
2019-10-01	2019-10-06	5413	5416
2019-10-02	2019-10-07	5414	5417
2019-10-03	2019-10-08	5415	5418
2019-10-04	2019-10-08	5415	5418
2019-10-05	2019-10-08	5415	5418
2019-10-06	2019-10-09	5416	5419
2019-10-07	2019-10-10	5417	5420
2019-10-08	2019-10-13	5418	5421

Result Set 6-5: Three business days into the future

Tip: This technique can be used to evaluate any cancellation window.

Determining turn time

We often need to track turn time. This is how long it takes to go from one stage in a workflow to the next. We might need to track this to verify service-level agreements, as a metric for capacity planning, or as a metric for promotions or performance reviews.

For many processes, you may need a **Time** table similar to the **Calendar** table we have been working on that would track **BusinessHours** in case you care about turn time in increments smaller than a business day. For our purposes here, we will assume that we only care about business days, so **TurnTime** will be tracking the number of business days between two dates.

Imagine that we have a table called **LoanMilestone** that tracks when a milestone happens on a loan.

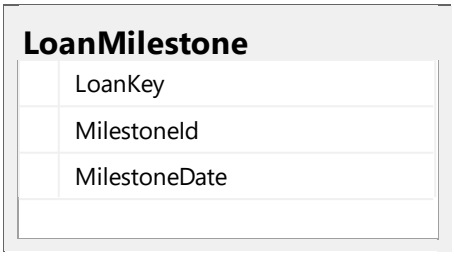


Figure 6-2: LoanMilestone table

In this system, **MilestoneId 25** refers to the **InitialProcessor** being assigned, and **MilestoneId 30** refers to the **FinalUnderwriter** being assigned. We have a workflow with the following steps:

1. Assign initial processor
2. Assign initial underwriter
3. Assign final processor
4. Assign final underwriter

For tracking purposes, we want to track how long it takes to go from initial processing to final underwriting. To accommodate all business scenarios, we need to accommodate that any of these steps could potentially be repeated. Events may happen to the loan during processing that require it to be reassigned to initial processor, or to repeat any step in the pipeline. To track the turn time, we care about the first time a loan was assigned to initial processing and the last time it was assigned to final underwriting.

To calculate the turn time, we need to find the number of business days between these two dates. To find the first time a loan was assigned to an initial processor, we will use the **MIN** function. To find the last time it was assigned to a final underwriter, we will use the **MAX** function. To find the number of business days between these two dates, we will join to the **Calendar** table twice and subtract the two **BusinessDayOverall** values.

Code Listing 6-13: Tracking business days between assigning processor and assigning underwriter

```
SELECT  InitialProcessor.InitialProcessorAssigned ,
        InitialProcessor.LoanKey ,
        InitialUnderwriter.InitialUnderwriterAssigned ,
        ProcessorCalendar.BusinessDayOverAll ,
        UnderwriterCalendar.BusinessDayOverAll ,
        UnderwriterCalendar.BusinessDayOverAll
        - ProcessorCalendar.BusinessDayOverAll AS TurnTime
FROM    ( SELECT    MIN(MilestoneDate) AS InitialProcessorAssigned ,
```

```
                    LoanKey
          FROM      dbo.LoanMilestone
          WHERE     MilestoneId = 25
          GROUP BY  LoanKey ) AS InitialProcessor
        INNER JOIN ( SELECT MAX(MilestoneDate) AS
InitialUnderwriterAssigned ,
                           LoanKey
                    FROM   dbo.LoanMilestone
                    WHERE  MilestoneId = 30
                    GROUP BY LoanKey ) InitialUnderwriter
              ON InitialUnderwriter.LoanKey = InitialProcessor.LoanKey
        INNER JOIN Reference.Calendar ProcessorCalendar
              ON InitialProcessor.InitialProcessorAssigned =
ProcessorCalendar.Date
        INNER JOIN Reference.Calendar UnderwriterCalendar
               ON InitialUnderwriter.InitialUnderwriterAssigned =
                  UnderwriterCalendar.Date
ORDER BY TurnTime;
```

Loan Key	Initial Underwriter Assigned	Initial Processor Assigned	Business Day Over All	Business Day Over All	Turn Time
2602	2018-03-28	2018-01-31	4979	5019	40
2606	2018-03-26	2018-01-29	4977	5017	40
2798	2018-03-26	2018-01-25	4975	5017	42
2958	2018-03-21	2018-01-22	4972	5014	42
2772	2018-03-31	2018-01-30	4978	5020	42
2821	2018-03-31	2018-01-30	4978	5020	42
2498	2018-03-26	2018-01-27	4975	5017	42
2455	2018-03-22	2018-01-23	4973	5015	42
2468	2018-03-30	2018-01-29	4977	5020	43
2960	2018-03-27	2018-01-25	4975	5018	43

Result Set 6-6: Showing the turn time data

Summary

In this chapter, we explored the importance of time and date information and reviewed the various data types for tracking them. We saw the functions for getting the various date parts from a date and how to find the difference between two dates in any increment, available as a **`DatePart`**.

Finally, we explored the difference between calendar days and business days, and built out the **`Calendar`** table to track a count of these two types of days so that we could evaluate business logic based on dates, and used business dates to evaluate the recession dates and produce turn time reports.

Date and time information is very important, and helps drive much of our business logic.

Chapter 7 Importance of the Data Dictionary

Learning the data dictionary is one of the most important things that you can do to make it easier to learn a new data model. Learning a new data model is often the hardest part of writing SQL queries. Quite often there is little to no documentation for the database, and sometimes there are remnants from old requirements and defunct systems. If the database has been in use for a while, there is a good chance that many of the relationships have been dropped to accommodate dirty data or changing data requirements.

All of this makes it harder to learn the data model.

The data dictionary can help. No matter how frequently the database is changed, the data dictionary is still up-to-date. No matter how out-of-date all other documentation may be, the data dictionary will be up-to-date because it's kept current by the database itself.

The data dictionary is a collection of tables and views that can track literally millions of pieces of information about the database. Internally, the database uses this information for performance tuning, tracking maintenance, building execution plans, etc. We can use it to learn where the data is stored and ferret out key information and relationships.

Learning the data model

When I talk about the data dictionary here, I am specifically talking about a set of system views stored in each database that make up the **`Information_Schema`** schema. There are other views that could be used. Each database vendor has system views that accommodate proprietary details for their implementation. They may also change from one version to the next to accommodate new features introduced in each version.

Tip: While you could get to the information you want by referencing system tables directly, stick with the Information_Schema views instead. Their structure is an ANSI standard and will be consistent across most database systems and versions. The same cannot be said about the raw system tables.

If you need to track vendor and version-specific metadata, you will need to use the system tables directly, but as you will see, we can get a lot of useful details while limiting ourselves to the **`Information_Schema`**, and this will help future-proof our processes. The queries we are about to go over will be portable across vendors and across database versions with the same vendor.

Note: While these views are an ANSI standard, not all database vendors follow them. The biggest missing player is Oracle, but there are open source projects devoted to filling this gap, such as this one.

Our discussion will focus on the following views:

- **Information_Schema.Tables**
- **Information_Schema.Columns**
- **Information_Schema.Views**
- **Information_Schema.View_Table_Usage**
- **Information_Schema.View_Column_Usage**
- **Information_Schema.Routines**
- **Information_Schema.Parameters**

Information_Schema.Tables

The **Tables** view has data for every table in the database. This is where we would go to get a list of tables in the database or, more often, a list of tables that satisfy a nested question, such as having a particular column. Once we have explored all of these views, we will explore some creative ways to use this information.

Tables has the following columns:

- **Table_Catalog**: The name of the database.
- **Table_Schema**: The name of the table's owner.
- **Table_Name**: The name of the table.
- **Table_Type**: An indicator for whether it's a base table of a view.

Information_Schema.Columns

The **Columns** view has data for every column in the database. There are quite a few columns in this view. In some circumstances, you may care about each of these columns, but for our purposes, only a handful of these columns are relevant.

- **Table_Catalog**: The name of the database.
- **Table_Schema**: The name of the table's owner.
- **Table_Name**: The name of the table.
- **Column_Name**: The name of the column.
- **Ordinal_Position**: The sequence for where the column is defined in the table.
- **Is_Nullable**: An indicator for whether or not this column can have NULL values.
- **Data_Type**: A string for the data type for the column.

The rest of the columns in this view are used to gather more details about the data type, such as the length, precision scale, etc.

Information_Schema.Views

The **Views** view has data for every view defined in the database. This is very similar to the **Tables** view, except for one notable addition. The **Views** view includes a view definition column, which contains the SQL used to create the view. This can be useful if we want to get the definition of a view.

Information_Schema.View_Table_Usage

The **View_Table_Usage** view has a record for every table used in any view definition. This can make it easier to query and get a list of tables involved in any view to track down data mapping.

Information_Schema.View_Column_Usage

The view **Information_Schema.View_Column_Usage** has a record for every column that is exposed in a view. This includes the **Table_Name** and **Column_Name** where the column comes from home. This makes it easy to see where the data from views comes from.

Information_Schema.Routines

The **Information_Schema.Routines** view has a record for every stored procedure and stored function defined in the database. There are lots of columns here to give more information about the programmability objects defined. For the purposes of learning the data model, we don't really care about most of these columns. The columns that we do care about include:

- **Specific_Name**: The name of the routine being defined.
- **Routine_Type**: An indicator to identify the type of routine being defined.
- **Routine_Definition**: This column holds the SQL used to define the routine.

Often all we really care about is identifying the stored procedures that are in the database.

Information_Schema.Parameters

The **Information_Schema.Parameters** view has a record for every parameter used in any routine.

- **Specific_Name** will have the name of the routine using the parameter.
- **Parameter_Name** will have the name of the parameter being defined.
- **Ordinal_Position** will have the order that the parameter is expected.
- **Parameter_Name** will be the name of the parameter being defined.

We also have a slew of columns to define the data type for the parameter.

We might use this to identify all of the stored procedures that include a specific parameter or list all of the parameters used by a specific stored procedure.

Common queries

There are some queries that you might use over and over to get a feel for what you have when you meet a new database for the first time.

Find out who the owners are

Code Listing 7-1: Who are the owners?

```
SELECT  TABLE_SCHEMA ,
        COUNT(1)
FROM    INFORMATION_SCHEMA.TABLES
GROUP BY TABLE_SCHEMA;
```

This will tell you who the owners are and how many tables they each own.

Tip: In many databases, you may find that everything is owned by one dbo (database owner), but sometimes tables are grouped together into functional areas by ownership.

Which tables are owned by a specific owner?

Code Listing 7-2: What does Finance own?

```
SELECT  TABLE_NAME
FROM    INFORMATION_SCHEMA.TABLES
WHERE   TABLE_SCHEMA = 'Finance'
ORDER BY TABLE_NAME;
```

This will identify all of the tables owned by **`Finance`**. If you have table owners for each functional area, this will give us a list of the tables that would be used in the **`Finance`** module of your system.

Which tables include a specific column?

Sometimes you might have a column name used throughout the data model. For example, in an insurance policy management system, you might find that every policy-level table will include a **`PolicyKey`** column. In a mortgage loan origination system, every loan level table may include a **`LoanKey`** column. This can help identify or filter out tables that are at that level.

Code Listing 7-3: Where are the PolicyKey columns?

```
SELECT  t.TABLE_SCHEMA ,
        t.TABLE_NAME
FROM    INFORMATION_SCHEMA.TABLES t
        INNER JOIN INFORMATION_SCHEMA.COLUMNS ON COLUMNS.TABLE_NAME =
t.TABLE_NAME
                                   AND COLUMNS.TABLE_SCHEMA =
t.TABLE_SCHEMA
WHERE   COLUMN_NAME = 'PolicyKey'
```

```
ORDER BY t.TABLE_NAME;
```

This query should identify all of the policy-level tables in your database.

Sometimes you want to do the opposite. In this case, for example, we want to find all of the tables that do not have a column named **`PolicyKey`**:

Code Listing 7-4: Which tables do not have a PolicyKey column?

```
SELECT  t.TABLE_SCHEMA ,
        t.TABLE_NAME
FROM    INFORMATION_SCHEMA.TABLES t
        LEFT  OUTER JOIN INFORMATION_SCHEMA.COLUMNS
                 ON COLUMNS.TABLE_NAME = t.TABLE_NAME
                     AND COLUMNS.TABLE_SCHEMA = t.TABLE_SCHEMA
                     AND COLUMN_NAME = 'PolicyKey'
WHERE COLUMN_NAME IS null
ORDER BY t.TABLE_NAME;
```

This will give us a list of all of the tables that are not at the policy level. Often this might mean that the tables are for configuration, administration, or something else not at the policy level.

Now we can extend this further to find the tables that have a combination of columns that we are interested in:

Code Listing 7-5: Do any tables have a LoanKey and Term column combination?

```
SELECT  t.TABLE_SCHEMA ,
        t.TABLE_NAME
FROM    INFORMATION_SCHEMA.TABLES t
        INNER JOIN INFORMATION_SCHEMA.COLUMNS P
                         ON P.TABLE_NAME = t.TABLE_NAME
                        AND P.TABLE_SCHEMA = t.TABLE_SCHEMA
        INNER JOIN INFORMATION_SCHEMA.COLUMNS Term
                         ON Term.TABLE_NAME = t.TABLE_NAME
                        AND Term.TABLE_SCHEMA = t.TABLE_SCHEMA
WHERE   P.COLUMN_NAME = 'LoanKey' AND term.COLUMN_NAME = 'Term'
ORDER BY t.TABLE_NAME;
```

Here we get a list of the tables that have both a **`LoanKey`** column and a **`Term`** column. This could be used to filter out any tables used to configure term options or term restrictions, and instead only report the references to **`Term`** that are at the loan level.

Search for common patterns that might point to a problem

I worked on an application once where the city, state, and zip fields for an address were combined in **CSZ** fields. Various tables would have all three values in one column. This will ultimately be problematic for various reasons, so I started explicitly looking for tables that have any columns that end with **CSZ**.

Code Listing 7-6: Did anyone put city, state, and zip in a single column?

```
SELECT  TABLE_NAME ,
        COLUMN_NAME
FROM    INFORMATION_SCHEMA.COLUMNS
WHERE   COLUMN_NAME LIKE '%CSZ';
```

Another problem may be dealing with common or inconsistent abbreviations. I see this a lot with date fields. Often, people get creative when naming date columns. For example, I have seen **ClosingDt**, **FirstPaymentDueDte**, and **RecissionDate** all in the same table. My preference is to always spell out the components of the name, but not everyone agrees with me. We can easily find all the variations used in your database:

Code Listing 7-7: How did we name the date columns?

```
SELECT  TABLE_NAME ,
        COLUMN_NAME
FROM    INFORMATION_SCHEMA.COLUMNS
WHERE   (DATA_TYPE = 'DateTime' or DATA_TYPE = 'Date')
        AND COLUMN_NAME NOT LIKE '%Date';
```

Tip: Never include a NOT LIKE clause in a condition that you plan to run in production, because it will generally involve a full table scan to retrieve the data. Fortunately, you can easily get the data you want in from the development or training environments.

Another common problem you are likely to run into is inconsistent data types for columns that should be consistent. This can complicate integrating with other systems or cause unexpected results when reporting within a single application.

Code Listing 7-8: Did we standardize the length of a City column?

```
SELECT  TABLE_NAME ,
        COLUMN_NAME, CHARACTER_MAXIMUM_LENGTH
FROM    INFORMATION_SCHEMA.COLUMNS
WHERE   COLUMN_NAME LIKE '%City';
```

You probably want to run a similar query to confirm consistent lengths for all state columns, zip code columns, phone number columns, etc. Anytime you see a common component for multiple field names, review these columns and ensure that they have a consistent data type, or that you understand why they are not consistent.

Note: Sometimes they may need to be different to accommodate a third-party interface or similar integration concerns.

Are we using any obsolete data types?

There are several data types that have been identified as obsolete, and we have been warned that they may be dropped in the future. It is a good idea to track these columns and make sure that new columns don't show up in your list. You can potentially even update them to a more functional standard, depending on your change control.

Code Listing 7-9: Finding obsolete types

```
SELECT  TABLE_NAME ,
        COLUMN_NAME
FROM    INFORMATION_SCHEMA.COLUMNS
WHERE   DATA_TYPE IN ( 'text', 'ntext', 'image', 'datetime' );
```

You may identify other data types that you want to avoid, so you may want to track their usage.

Code Listing 7-10: Finding types to avoid

```
SELECT  TABLE_NAME ,
        COLUMN_NAME
FROM    INFORMATION_SCHEMA.COLUMNS
WHERE   DATA_TYPE IN ( 'uniqueidentifier', 'ntext');
```

What data types are being used?

It's good to know what data types are being used across the system. The first time you run such a query, you may find a lot of inconsistencies.

Code Listing 7-11: What data types are being used?

```
SELECT  DATA_TYPE ,
        COUNT(1)
FROM    INFORMATION_SCHEMA.COLUMNS
GROUP BY DATA_TYPE
ORDER BY DATA_TYPE;
```

You may find results like this:

Data Type	Count
Bit	17
char	2
datetime	34
datetime2	9
decimal	124
float	1
Int	25
Nchar	2
nvarchar	5
Smallint	2
Varbinary	1
varchar	270

Result Set 7-1: Data types being used

This is good information to have. You can quickly tell that you have not taken advantage of the new **`Date`** data type. You can also quickly tell that someone has probably gotten confused about the various numeric data types, such as **`NChar`** and **`NVarChar`**.

Tip: Unless you need support for internationalization, stick with Char and VarChar.

This will help you track the inconsistencies and progress towards standardization.

Summary

We have seen in this chapter that the data dictionary provides a good source of documentation for the database, even when all other documentation has long since been outdated, because the database itself is keeping this documentation current. We have seen how to use this information to quickly get an overview of what's in the database by finding out who owns the structures in the database, and getting a list of every table in the database and the columns associated with each of these tables. We can easily identify which tables are reference data and which have transaction data by searching for key columns being included in the table.

We have also seen some examples of how to detect patterns in the database structure that may cause problems. We looked at finding a pattern where city, state, and zip code data was combined in a single field, where naming conventions were inconsistent for **`Date`** columns, where the lengths for common columns such as **`City`** were not consistent, or we had not standardized on data types for numeric values.

There is a wealth of information available that can help you quickly get up to speed on a data model when you face one and wonder where to begin.