STUDENT MANUAL

SQL Querying: Advanced

SQL Querying: Advanced

SQL Querying: Advanced

Part Number: 094006
Course Edition: 1.0

Acknowledgements

PROJECT TEAM

Author	*Media Designer*	*Content Editor*
Rozanne Whalen	Alex Tong	Tricia Murphy

Notices

SQL Querying: Advanced

About This Course

In the course *SQL Querying: Fundamentals,* you learned the basics of SQL querying. Now that you have a foundation in how to query databases, you will learn in this course how to identify and use advanced querying techniques. In addition, you might find that you need to perform tasks such as modifying the structure of a table; inserting, updating, or deleting data; and indexing tables to optimize query performance. In this course, you will learn how to complete these tasks and more.

In today's competitive environment, information is one of the most important factors in determining the success of an organization. If you are able to manage and retrieve information efficiently, you can streamline the organization's processes and give it a competitive edge. As the organization grows, you will need to handle large amounts of data. Under such circumstances, you might need to query multiple tables simultaneously and with increasing frequency. You must step up the speed of generating query output to cope with the increasing demands of data storage, management, and retrieval.

Course Description

Target Student

Students should have basic computer skills, SQL skills, and be familiar with concepts related to database structure and terminology.

Course Prerequisites

To ensure your success, we recommend you first take the following Logical Operations courses, or have equivalent skills and knowledge:

- *SQL Querying: Fundamentals*

Course Objectives

In this course, you will work with advanced queries to manipulate and index tables. You will also create transactions so that you can choose to save or cancel the data entry process.

You will:

- Use subqueries to generate query output.
- Manipulate table data by inserting and updating records in a table and deleting records from a table.
- Manipulate the table structure.
- Create views, manipulate data through views, modify the view structure, and drop views.
- Create indexes on table columns and drop inefficient indexes.
- Mark the beginning of a transaction, roll back a transaction, and commit a transaction.

The LogicalCHOICE Home Screen

The LogicalCHOICE Home screen is your entry point to the LogicalCHOICE learning experience, of which this course manual is only one part. Visit the LogicalCHOICE Course screen both during and after class to make use of the world of support and instructional resources that make up the LogicalCHOICE experience.

Log-on and access information for your LogicalCHOICE environment will be provided with your class experience. On the LogicalCHOICE Home screen, you can access the LogicalCHOICE Course screens for your specific courses.

Each LogicalCHOICE Course screen will give you access to the following resources:

- eBook: an interactive electronic version of the printed book for your course.
- LearnTOs: brief animated components that enhance and extend the classroom learning experience.

Depending on the nature of your course and the choices of your learning provider, the LogicalCHOICE Course screen may also include access to elements such as:

- The interactive eBook.
- Social media resources that enable you to collaborate with others in the learning community using professional communications sites such as LinkedIn or microblogging tools such as Twitter.
- Checklists with useful post-class reference information.
- Any course files you will download.
- The course assessment.
- Notices from the LogicalCHOICE administrator.
- Virtual labs, for remote access to the technical environment for your course.
- Your personal whiteboard for sketches and notes.
- Newsletters and other communications from your learning provider.
- Mentoring services.
- A link to the website of your training provider.
- The LogicalCHOICE store.

Visit your LogicalCHOICE Home screen often to connect, communicate, and extend your learning experience!

How to Use This Book

As You Learn

This book is divided into lessons and topics, covering a subject or a set of related subjects. In most cases, lessons are arranged in order of increasing proficiency.

The results-oriented topics include relevant and supporting information you need to master the content. Each topic has various types of activities designed to enable you to practice the guidelines and procedures as well as to solidify your understanding of the informational material presented in the course. Procedures and guidelines are presented in a concise fashion along with activities and discussions. Information is provided for reference and reflection in such a way as to facilitate understanding and practice.

Data files for various activities as well as other supporting files for the course are available by download from the LogicalCHOICE Course screen. In addition to sample data for the course exercises, the course files may contain media components to enhance your learning and additional reference materials for use both during and after the course.

At the back of the book, you will find a glossary of the definitions of the terms and concepts used throughout the course. You will also find an index to assist in locating information within the instructional components of the book.

As You Review

Any method of instruction is only as effective as the time and effort you, the student, are willing to invest in it. In addition, some of the information that you learn in class may not be important to you immediately, but it may become important later. For this reason, we encourage you to spend some time reviewing the content of the course after your time in the classroom.

As a Reference

The organization and layout of this book make it an easy-to-use resource for future reference. Taking advantage of the glossary, index, and table of contents, you can use this book as a first source of definitions, background information, and summaries.

Course Icons

Watch throughout the material for these visual cues:

Icon	*Description*
	A **Note** provides additional information, guidance, or hints about a topic or task.
	A **Caution** helps make you aware of places where you need to be particularly careful with your actions, settings, or decisions so that you can be sure to get the desired results of an activity or task.
	LearnTO notes show you where an associated LearnTO is particularly relevant to the content. Access LearnTOs from your LogicalCHOICE Course screen.
	Checklists provide job aids you can use after class as a reference to performing skills back on the job. Access checklists from your LogicalCHOICE Course screen.
	Social notes remind you to check your LogicalCHOICE Course screen for opportunities to interact with the LogicalCHOICE community using social media.
	Notes Pages are intentionally left blank for you to write on.

1 Using Subqueries to Perform Advanced Querying

Lesson Time: 1 hour, 25 minutes

Lesson Objectives

In this lesson, you will use subqueries to generate query output. You will:

- Search based on unknown values.
- Compare a value with unknown values.
- Search based on the existence of records.
- Generate output by using correlated subqueries.
- Filter grouped data within subqueries.
- Perform multiple-level subqueries.

Introduction

In this course, you will build on the fundamentals of SQL querying by mastering advanced techniques such as using subqueries, manipulating table data, and creating and modifying table structures. Advanced queries such as subqueries enable you to retrieve data when you aren't sure of the search value you want to use or you want to compare a search value with an unknown value. In addition, you might use nested queries in order to pinpoint the exact data you need. In this lesson, you will learn how to create advanced queries that use subqueries.

Imagine you are looking to buy a laptop. You aren't sure which model you want to buy. However, you do know how much RAM, hard disk space, and which processor you want in the laptop. A nested subquery enables you to search for all models that meet your requirements for RAM, hard disk size, and processor.

TOPIC A

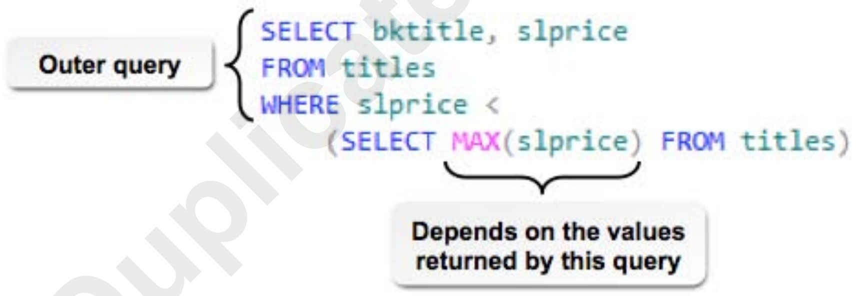
Search Based on Unknown Values

In the *SQL Querying: Fundamentals* course, you learned how to construct `WHERE` clauses to search for specific rows in tables. On occasion, you might need to search for rows when you don't have a specific search value. Instead, you want to search a table based on values you find in another table. To accomplish this type of search, you must use subqueries. In this topic, you will use subqueries to retrieve records based on unknown search values.

To serve as an incentive, the manager of a hotel decides to reward clients who have paid the highest bill amount for the month, and he assigns you the task of finding such clients. You need to first identify the highest bill amount, which is an unknown value. You then need to use this value to identify the client. In this case, you cannot use simple queries to perform the search because simple queries require concrete search values to search for specific records. To retrieve records based on an unknown value, you need to use subqueries.

Outer Queries

An *outer query* is a query that depends on the values returned by another query used to display records. An outer query can use multiple queries to display the required records. An outer query is also known as the main query. However, if the inner query returns a null value, then the outer query also returns a null value.



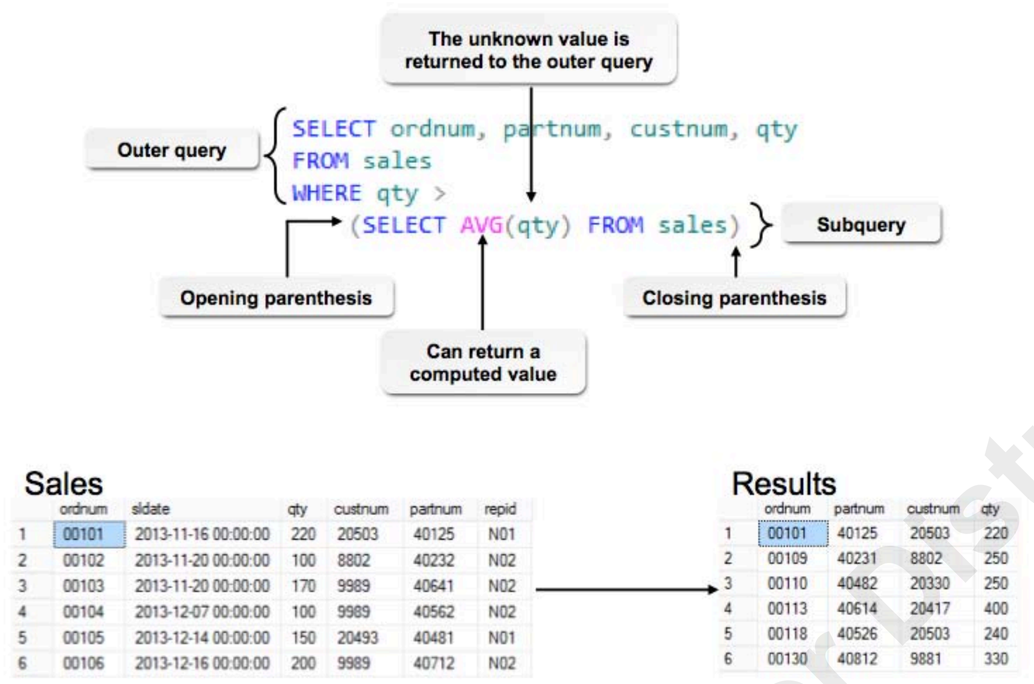
```
SELECT bktitle, slprice
FROM titles
WHERE slprice <
      (SELECT MAX(slprice) FROM titles)
```

Figure 1-1: An outer query example.

In this example, the query retrieves the title of the book (bktitle) and its sale price (slprice) from the Titles table. The `WHERE` clause, in conjunction with the inner query, retrieves only those books with a sale price that is lower than the highest-priced book in the Titles table.

Subqueries

A *subquery*, also known as an *inner query*, is a query that is contained within an outer query. A subquery is used when the value needed by the outer query's condition is unknown. The subquery retrieves the value and returns it to the outer query. Subqueries can also return computed values. They are always written within parentheses and can contain multiple inner queries, with each inner query enclosed within parentheses. You can sort and group records retrieved by the subquery.


The unknown value is returned to the outer query
Outer query
Subquery
Opening parenthesis
Closing parenthesis
Can return a computed value

```
SELECT ordnum, partnum, custnum, qty
FROM sales
WHERE qty >
(SELECT AVG(qty) FROM sales)
```

Sales

	ordnum	sldate	qty	custnum	partnum	repid
1	00101	2013-11-16 00:00:00	220	20503	40125	N01
2	00102	2013-11-20 00:00:00	100	8802	40232	N02
3	00103	2013-11-20 00:00:00	170	9989	40641	N02
4	00104	2013-12-07 00:00:00	100	9989	40562	N02
5	00105	2013-12-14 00:00:00	150	20493	40481	N01
6	00106	2013-12-16 00:00:00	200	9989	40712	N02

Results

	ordnum	partnum	custnum	qty
1	00101	40125	20503	220
2	00109	40231	8802	250
3	00110	40482	20330	250
4	00113	40614	20417	400
5	00118	40526	20503	240
6	00130	40812	9881	330

Figure 1-2: A subquery example.

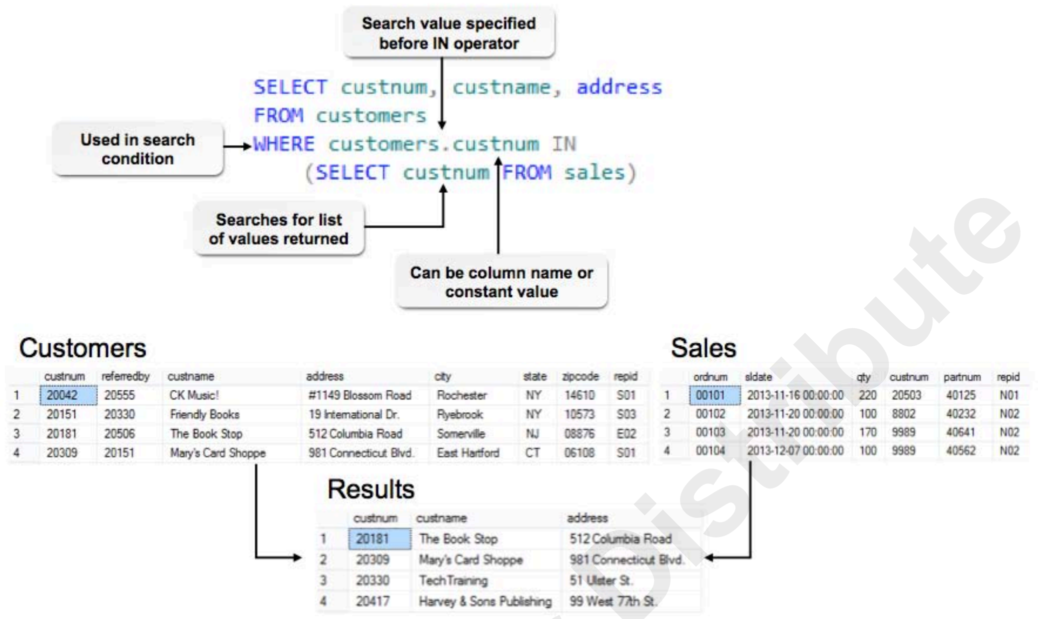
For this example, consider an instance where you need to display the order number, part number, customer number, and quantity of each sale in a bookstore where the quantity of books purchased is higher than the average quantity of books sold in the store. To obtain the necessary results, you use a subquery to return the average quantity value to the outer query. The outer query checks whether the quantity value of each sales record is greater than the average value. If the condition is true, it displays the required information.

```
SELECT ordnum, partnum, custnum, qty
FROM sales
WHERE qty >
   (SELECT AVG (qty) FROM sales)
```

The IN Operator

The *IN operator* is a logical operator that is used in the search condition of a query to search for a value based on a list of values. When used in the search condition of an outer query, the `IN` operator searches for a value in the list of values returned by the subquery.

The `IN` operator is always preceded by the value to be searched, and you can specify this value as a column name or as a constant value. The operator is followed by a subquery that returns a list of values. You can use the `IN` operator with the `NOT` operator to search for records that are not present in the list of values returned by the subquery.


Search value specified before IN operator
Used in search condition
Searches for list of values returned
Can be column name or constant value

```
SELECT custnum, custname, address
FROM customers
WHERE customers.custnum IN
      (SELECT custnum FROM sales)
```

Customers

	custnum	referredby	custname	address	city	state	zipcode	repid
1	20042	20555	CK Music!	#1149 Blossom Road	Rochester	NY	14610	S01
2	20151	20330	Friendly Books	19 International Dr.	Ryebrook	NY	10573	S03
3	20181	20506	The Book Stop	512 Columbia Road	Somerville	NJ	08876	E02
4	20309	20151	Mary's Card Shoppe	981 Connecticut Blvd.	East Hartford	CT	06108	S01

Sales

	ordnum	sldate	qty	custnum	partnum	repid
1	00101	2013-11-16 00:00:00	220	20503	40125	N01
2	00102	2013-11-20 00:00:00	100	8802	40232	N02
3	00103	2013-11-20 00:00:00	170	9989	40641	N02
4	00104	2013-12-07 00:00:00	100	9989	40562	N02

Results

	custnum	custname	address
1	20181	The Book Stop	512 Columbia Road
2	20309	Mary's Card Shoppe	981 Connecticut Blvd.
3	20330	TechTraining	51 Ulster St.
4	20417	Harvey & Sons Publishing	99 West 77th St.

Figure 1-3: An IN operator example.

In this example, the query displays the customer numbers, names, and addresses of customers who have bought books from a bookstore. The numbers, names, and addresses are stored in the Customers table, and all records of sales are stored in the Sales table. To retrieve this information, you must use a subquery to return a list of customer numbers present in the Sales table. You can then display the required records by using an `IN` operator in the outer query's condition to verify whether the customer number in the Customers table is present in the list returned by the subquery.

The IN Operator with NULL Values

Whenever a subquery returns NULL values and the outer query has the `IN` operator in the search condition, it will result in an UNKNOWN value, thus producing an unpredictable result.

Access the Checklist tile on your LogicalCHOICE course screen for reference information and job aids on How to Search Based on Unknown Values.

ACTIVITY 1-1

Searching Based on an Unknown Value

Scenario

You work for the Fuller & Ackerman Publishing bookstore. You maintain the SQL database named FullerAckerman. You've been asked to find the following information:

- To estimate the distribution of pay among the sales representatives in a team, the HR manager of a bookstore wants to identify sales representatives who are earning above-average salaries. The manager wants you to generate a list of sales representatives who have been earning a commission at rates above the average rate in the team.
- To focus on selling high-priced book titles, the sales manager wants you to identify the book titles that are priced above the average price of the books in the store. Use the Slspers and Titles tables to generate the required output.

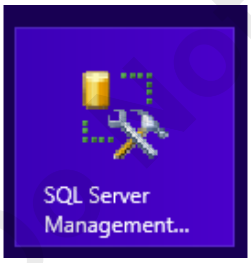
For information on the tables and column names in the FullerAckerman database, refer to the table structure in Appendix A.

Note: Activities may vary slightly if the software vendor has issued digital updates. Your instructor will notify you of any changes.

1. **Which task do you need to perform to identify commission rates that are above the average commission rate for the sales team?**
   - ○ Generate the average commission rate for the team.
   - ○ Generate the commission rate for each sales representative.
   - ○ Generate the maximum commission rate received by a sales representative.
   - ○ Compare the commission rate of each sales representative with the average commission rate for the team.

2. In Microsoft® SQL Server® Management Studio, create a subquery that lists the representative ID, first name, and last name of sales representatives who earn a commission rate that is above the average rate for the team.

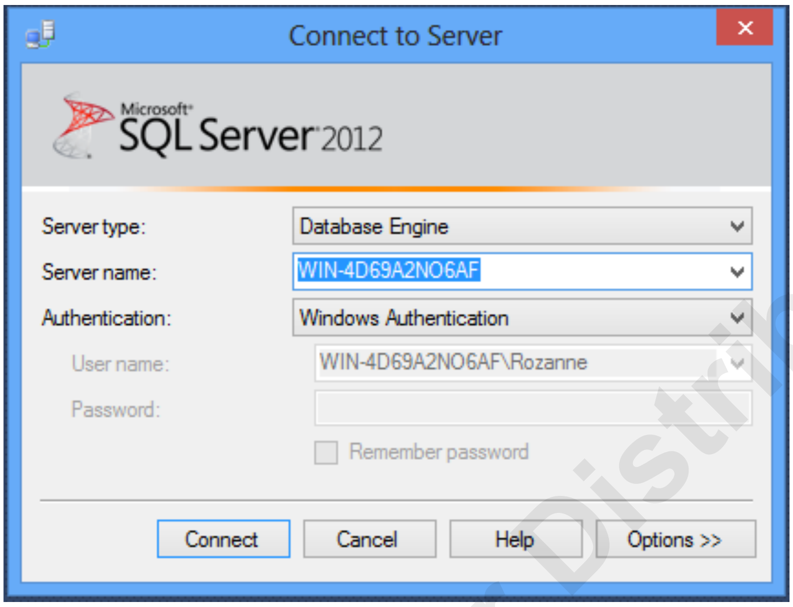
   a) On the **Start** screen, select **SQL Server Management Studio**.

   b) In the **Connect to Server** dialog box, in the **Server type** drop-down list, verify that **Database Engine** is selected.

   c) In the **Server name** drop-down list, verify that the name of the server is automatically selected. In this case, the server name is the same as your computer's name.

d) In the **Authentication** drop-down list, verify that **Windows Authentication** is selected.

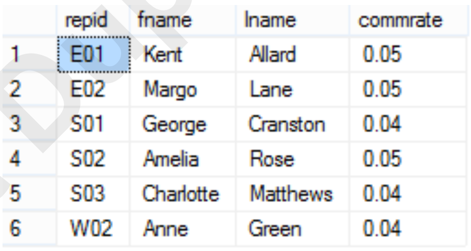


Server type: Database Engine
Server name: WIN-4D69A2NO6AF
Authentication: Windows Authentication
User name: WIN-4D69A2NO6AF\Rozanne
Password:
Remember password

e) Select **Connect** to connect to the server.
f) On the **Standard** toolbar, select **New Query** to open the **Query Editor** window.
g) On the **SQL Editor** toolbar, from the **Available Databases** drop-down list, select **FullerAckerman**.
h) Enter this query:

```
SELECT repid, fname, lname, commrate
FROM slspers
WHERE commrate >
    (SELECT AVG(commrate) FROM slspers)
```

i) On the **SQL Editor** toolbar, select the **Execute** button.
j) Observe that six records are displayed as the output of the query.



	repid	fname	lname	commrate
1	E01	Kent	Allard	0.05
2	E02	Margo	Lane	0.05
3	S01	George	Cranston	0.04
4	S02	Amelia	Rose	0.05
5	S03	Charlotte	Matthews	0.04
6	W02	Anne	Green	0.04

3. How would you identify books that are priced above the average book price? (Choose two.)
   - ☐ Generate the average sale price of books, which is an unknown value.
   - ☐ Add the total price of all books and divide the total price by two.
   - ☐ Compare the average sale price with the sale price of each book in the store.
   - ☐ Compare the average sale price with the minimum sale price of the books.

4. Create a subquery that lists the part number, book title, and sale price of titles that are priced above the average book price in the store.

   a) Select your previous query with the mouse and press **Del** to delete it.

b) Enter the query for the `SELECT` clause of the outer query to generate the part number using the partnum column, book title (bktitle column), and sale price (slprice column) from the Titles table:

```
SELECT partnum, bktitle, slprice
FROM titles
```

c) Specify the search condition for the outer query to search for a sale price that is greater than the value supplied by the inner query:

```
SELECT partnum, bktitle, slprice
FROM titles
WHERE slprice >
```

d) Write the inner query to generate the average sale price from the Titles table:

```
SELECT partnum, bktitle, slprice
FROM titles
WHERE slprice >
   (SELECT AVG(slprice) FROM titles)
```

e) Select the **Execute** button. SQL Server returns 42 rows.

f) Close the **Query Editor** window without saving the query.

ACTIVITY 1-2

Searching Based on Multiple Unknown Values

Before You Begin

- Ensure SQL Server Management Studio is open.
- On the **Standard** toolbar, select **New Query**.
- On the **SQL Editor** toolbar, in the **Database** drop-down list, ensure that the FullerAckerman database is selected.

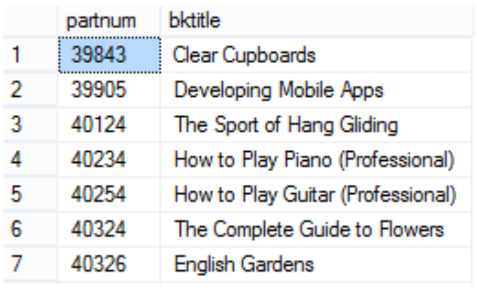
Scenario

A number of books are lying around unused in the Fuller & Ackerman Publishing bookstore and reports about sales representatives have been submitted to the sales manager. The manager wants you to identify book titles that have not been sold at all. You also need to identify sales representatives who have and have not sold books. You need to work with the Titles, Slspers, and Sales tables to generate the required output. For information on the tables and column names in the FullerAckerman database, refer to the table structure in Appendix A.

1. **True or False? To generate the list of books that have never been sold from the store, you need to verify whether the part number of each book is present in any of the sales recorded in the Sales table.**
   - ☐ True
   - ☐ False

2. Generate the list of books that have never been sold.
   a) In the **Query Editor** window, enter the outer query with a `SELECT` clause to generate the part number (partnum column), and book title (bktitle) from the Titles table:
   ```
   SELECT partnum, bktitle
   FROM titles
   ```
   b) Enter `WHERE partnum NOT IN` to specify the search condition with the `NOT IN` operator to search for part numbers in the Titles table that are not present in the inner query's output:
   ```
   SELECT partnum, bktitle
   FROM titles
   WHERE partnum NOT IN
   ```
   c) Add the inner query to retrieve a list of part numbers from the Sales table:
   ```
   SELECT partnum, bktitle
   FROM titles
   WHERE partnum NOT IN
      (SELECT partnum FROM sales)
   ```
   d) Execute the query.

e) Observe the list of books that have never been sold. There are 37 books that haven't sold.



	partnum	bktitle
1	39843	Clear Cupboards
2	39905	Developing Mobile Apps
3	40124	The Sport of Hang Gliding
4	40234	How to Play Piano (Professional)
5	40254	How to Play Guitar (Professional)
6	40324	The Complete Guide to Flowers
7	40326	English Gardens

3. **Which task do you need to perform to generate a list of sales representatives who have not made a single book sale?**

○ Verify whether the part number of each book is present in any of the sales recorded in the Sales table.

○ Verify whether the representative ID of each representative is present in any of the sales recorded in the Sales table.

○ Verify whether the customer ID of each customer is present in any of the sales recorded in the Sales table.

○ Verify whether the commission rate of each representative is present in any of the sales recorded in the Sales table.

4. Create a subquery to generate the list of sales representatives who have sold books for the store.

a) Delete the previous query.

b) Enter a `SELECT` statement that generates the representative IDs (repid column), and names of sales representatives (fname and lname columns) from the Slspers table:

```
SELECT repid, fname, lname
FROM slspers
```

c) Specify the search condition to search whether each representative ID of the Slspers table is present in the inner query output:

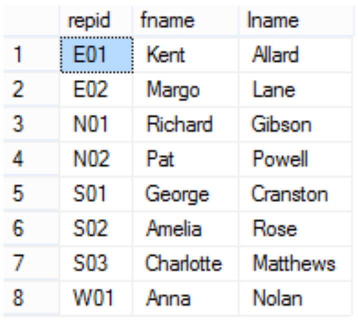
```
SELECT repid, fname, lname
FROM slspers
WHERE repid IN
```

d) Write an inner query that generates a list of representative IDs from the Sales table:

```
SELECT repid, fname, lname
FROM slspers
WHERE repid IN
   (SELECT repid FROM sales)
```

e) Execute the query you entered to display the list of sales representatives who have contributed to the sale of books.

f) Observe that eight sales representatives have contributed to the sale of books from the store.



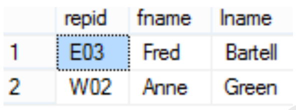
	repid	fname	lname
1	E01	Kent	Allard
2	E02	Margo	Lane
3	N01	Richard	Gibson
4	N02	Pat	Powell
5	S01	George	Cranston
6	S02	Amelia	Rose
7	S03	Charlotte	Matthews
8	W01	Anna	Nolan

5. Generate the list of sales representatives who have not sold books for the store.

a) Modify the search condition by adding the `NOT` keyword before the `IN` keyword to search for representative IDs. Your query should read as follows:

```
SELECT repid, fname, lname
FROM slspers
WHERE repid NOT IN
    (SELECT repid FROM sales)
```

b) Execute the query you entered and observe the list of two sales representatives who have not sold books for the store.



	repid	fname	lname
1	E03	Fred	Bartell
2	W02	Anne	Green

c) Close the **Query Editor** window without saving the query.

TOPIC B

Compare a Value with Unknown Values

You have retrieved records based on unknown values. Sometimes, you might want to write queries that compare a single known value with a list of unknown values in order to retrieve the exact rows you need. In this topic, you will display rows by comparing a known value with unknown values in order to obtain the required list of rows.

You are a trainee programmer and the human resources manager of your organization wants you to verify whether the salary of an employee in one department is comparatively lower than an employee's salary in another department in your company. To accomplish this task, you must compare the salary of the employee to the salaries of everyone in the company. You can accomplish this task by comparing a known value (the employee's salary) with a list of unknown values (the salaries of all other employees).

Modified Comparison Operators

A *modified comparison operator* is a comparison operator for which the function is modified when you combine it with the `ANY` or `ALL` logical operator. You can use the `ANY` and `ALL` operators only with comparison operators.

You use modified comparison operators in the `WHERE` clause of an outer query to compare a value with the maximum or minimum value in a list of values generated by a subquery. Specify the value to be compared before the operator. The choice of comparing a value with the maximum or minimum value in a list depends on the combination of the comparison operator and the `ANY` or `ALL` operators you use in the `WHERE` clause.

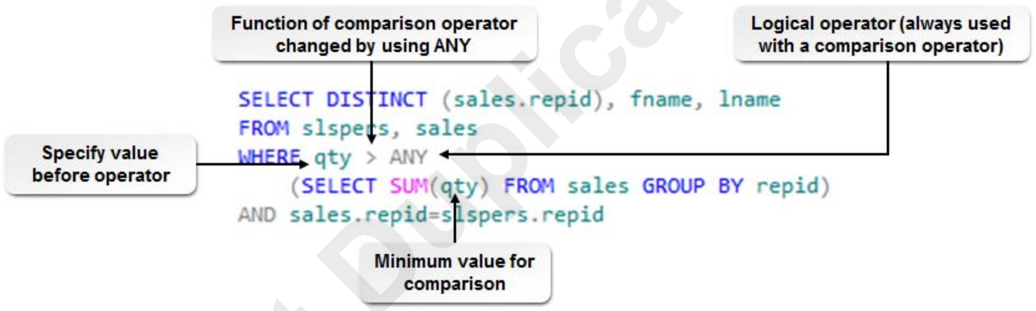
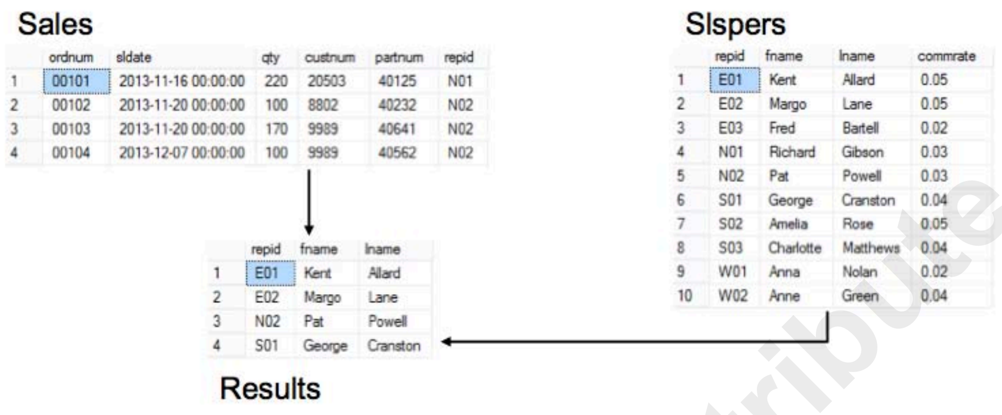

```
SELECT DISTINCT (sales.repid), fname, lname
FROM slspers, sales
WHERE qty > ANY
    (SELECT SUM(qty) FROM sales GROUP BY repid)
AND sales.repid=slspers.repid
```
Function of comparison operator changed by using ANY
Logical operator (always used with a comparison operator)
Specify value before operator
Minimum value for comparison

Figure 1-4: A modified comparison operator example.

In the query example, you want to display the representative IDs and names of sales representatives who have sold more books in a single sale than the total sales made by other sales representatives. You need to use the > comparison operator with the `ANY` logical operator in the search condition of the outer query to compare the quantity sold in each sale with the minimum value in the list generated by the subquery. The subquery lists the total quantities sold by each sales representative.


Sales

	ordnum	sldate	qty	custnum	partnum	repid
1	00101	2013-11-16 00:00:00	220	20503	40125	N01
2	00102	2013-11-20 00:00:00	100	8802	40232	N02
3	00103	2013-11-20 00:00:00	170	9989	40641	N02
4	00104	2013-12-07 00:00:00	100	9989	40562	N02

Slspers

	repid	fname	lname	commrate
1	E01	Kent	Allard	0.05
2	E02	Margo	Lane	0.05
3	E03	Fred	Bartell	0.02
4	N01	Richard	Gibson	0.03
5	N02	Pat	Powell	0.03
6	S01	George	Cranston	0.04
7	S02	Amelia	Rose	0.05
8	S03	Charlotte	Matthews	0.04
9	W01	Anna	Nolan	0.02
10	W02	Anne	Green	0.04

	repid	fname	lname
1	E01	Kent	Allard
2	E02	Margo	Lane
3	N02	Pat	Powell
4	S01	George	Cranston

Results

Figure 1-5: Results of a modified comparison operator.

Comparison Operators Used with ANY and ALL

The combination of the comparison operator and the `ANY` and `ALL` operators determines the comparison performed in the condition of an outer query. The following table explains various operator combinations.

Operator Combination	*Description*
`>ANY`	Greater than the minimum value in a list.
`>=ANY`	Greater than or equal to the minimum value in a list.
`!<ANY`	
`<ALL`	Lesser than the minimum value in a list.
`<=ALL`	Lesser than or equal to the minimum value in a list.
`!>ALL`	
`<ANY`	Lesser than the maximum value in a list.
`<=ANY`	Lesser than or equal to the maximum value in a list.
`!>ANY`	
`>ALL`	Greater than the maximum value in a list.
`>=ALL`	Greater than or equal to the maximum value in a list.
`!<ALL`	
`=ANY`	Equal to any value in a list.
`=ALL`	Equal to all values in a list.
`<>ANY`	Not equal to any of the values in a list.
`!=ANY`	

Operator Combination	*Description*

`<>ALL`
`!=ALL`
	Not equal to all the values in a list.

Access the Checklist tile on your LogicalCHOICE course screen for reference information and job aids on How to Compare a Value with Unknown Values.

ACTIVITY 1-3
Searching by Comparing with Unknown Values

Before You Begin

- Ensure SQL Server Management Studio is open.
- On the **Standard** toolbar, select **New Query**.
- On the **SQL Editor** toolbar, in the **Database** drop-down list, ensure that the FullerAckerman database is selected.

Scenario

At the close of the last top management meeting at the Fuller & Ackerman Publishing bookstore, it was decided that customers who make large purchases need to be identified and rewarded with discounts. The marketing manager at the store has come up with the Silver Card plan. All customers whose single purchase quantity is above the total purchases made by other customers are to be rewarded with a Silver Card. These cardholders can receive a 15 percent discount on all future purchases from the store. The marketing manager wants you to generate the list of customers who fall under the card plan. Use the Customers and Sales tables to generate the required output. For information on the tables and column names in the FullerAckerman database, refer to the table structure in Appendix A.

---

1. Which is the best possible logic you can use to identify customers who fall under the Silver Card plan?
   - ○ Use an inner query to generate a list of total quantities purchased by each customer and use the outer query's search condition to compare whether the quantity of each purchase is greater than the minimum value in the list of values generated by the inner query.
   - ○ Use a subquery to generate a list of total purchase quantities for each customer.
   - ○ Use a subquery to generate a list of the top five total purchase quantities for each customer.
   - ○ Generate the list of customers whose total purchase quantities are above the average purchase made by a customer in the store.

2. Retrieve the list of customers who can be rewarded with a Silver Card.
   a) Enter the `SELECT` statement to retrieve the Sales table's customer number (custnum), the name (custname), and complete address details (address, city, state, and zipcode columns) from the Customers and the Sales tables. Use the `DISTINCT` keyword when displaying the customer number to eliminate duplicate customer numbers in the outer query output:

```
SELECT DISTINCT (sales.custnum), custname, address, city, state, zipcode
FROM sales, customers
```

   b) Add to the query to compare a single purchase by a customer using the qty column with the minimum value in the list of values generated by the inner query output:

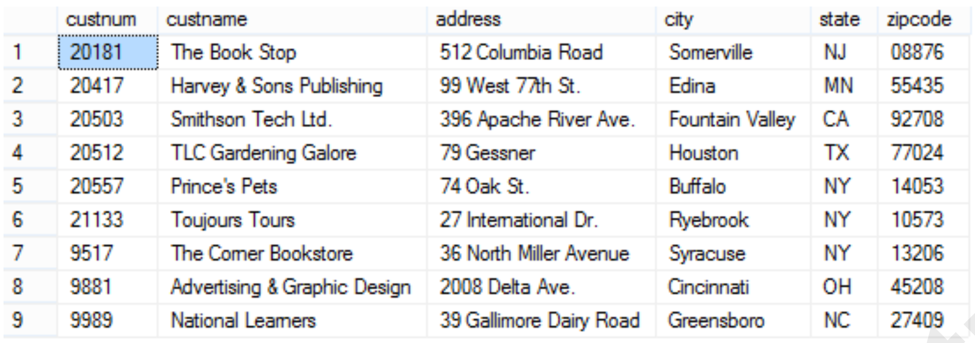
```
SELECT DISTINCT (sales.custnum), custname, address, city, state, zipcode
FROM sales, customers
WHERE qty > ANY
    (SELECT SUM(qty) FROM sales GROUP BY custnum)
```

   c) Specify another search condition to ensure that the records of only those customers who have matching customer numbers in the Sales and Customers tables are generated in the subquery output:

```
SELECT DISTINCT (sales.custnum), custname, address, city, state, zipcode
FROM sales, customers
WHERE qty > ANY
```

```
    (SELECT SUM(qty) FROM sales GROUP BY custnum)
AND sales.custnum=customers.custnum
```

d) Execute the query. There are nine customers who qualify for the Silver Card.



	custnum	custname	address	city	state	zipcode
1	20181	The Book Stop	512 Columbia Road	Somerville	NJ	08876
2	20417	Harvey & Sons Publishing	99 West 77th St.	Edina	MN	55435
3	20503	Smithson Tech Ltd.	396 Apache River Ave.	Fountain Valley	CA	92708
4	20512	TLC Gardening Galore	79 Gessner	Houston	TX	77024
5	20557	Prince's Pets	74 Oak St.	Buffalo	NY	14053
6	21133	Toujours Tours	27 International Dr.	Ryebrook	NY	10573
7	9517	The Corner Bookstore	36 North Miller Avenue	Syracuse	NY	13206
8	9881	Advertising & Graphic Design	2008 Delta Ave.	Cincinnati	OH	45208
9	9989	National Learners	39 Gallimore Dairy Road	Greensboro	NC	27409

e) Close the **Query Editor** window without saving the query.

TOPIC C

Search Based on the Existence of Records

In the previous topics, you queried tables by using subqueries to retrieve the values for your search conditions. On occasion, you might need to display rows from a table based on the existence of one or more rows in another table. In this topic, you will display records based on the existence of rows in another table.

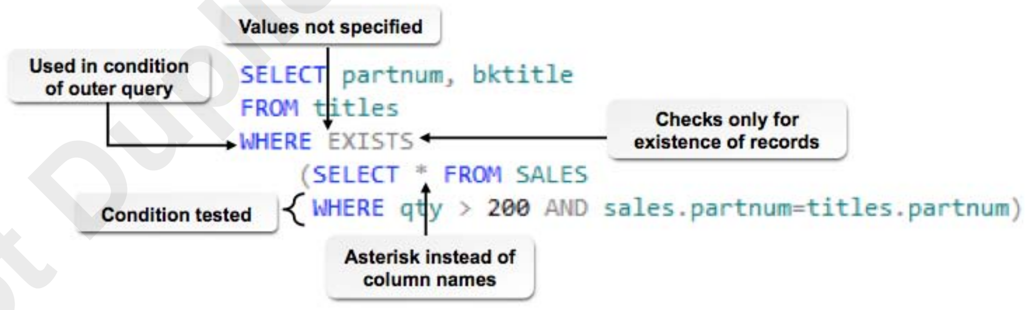
When you use a credit card to pay for a purchase, the cashier at a store must check whether the credit account really exists. Just as the cashier must verify the validity of your credit card account, so might you use outer queries and subqueries to check for the existence of records in another table before displaying the required records.

The EXISTS Operator

The *EXISTS operator* is a logical operator that you use in the `WHERE` clause of an outer query to check for the existence of records returned by a subquery. In the search condition, do not specify the column names or constant values before the `EXISTS` operator.

You can use the `EXISTS` operator to display records based on the success or failure of the subquery's condition. If this condition is true, the subquery generates records. The `EXISTS` operator then confirms the existence of these records by returning the value `TRUE`. Based on this confirmation, the outer query displays records corresponding to the records returned by the subquery. The `SELECT` clause of the subquery typically contains an asterisk instead of specific column names.

Note: You cannot use the `COMPUTE` and the `INTO` keywords in a query containing the `EXISTS` operator.

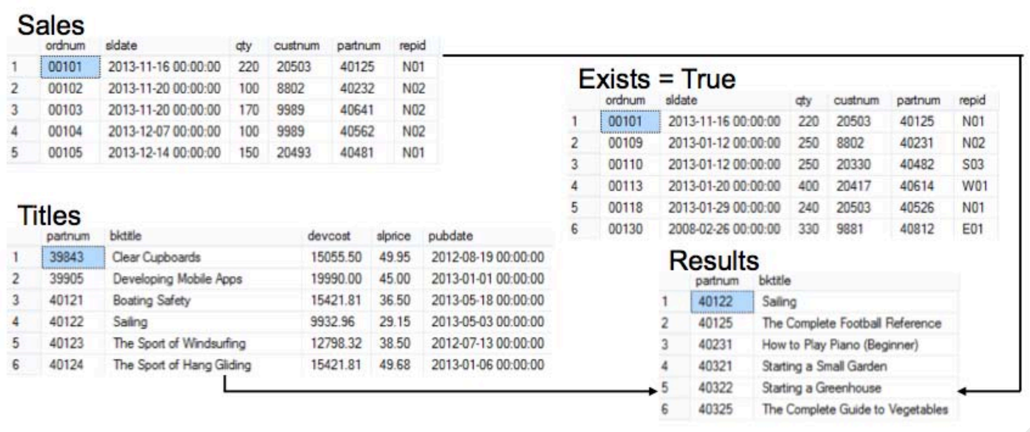


Values not specified
Used in condition of outer query
Checks only for existence of records
Condition tested
Asterisk instead of column names

```
SELECT partnum, bktitle
FROM titles
WHERE EXISTS
    (SELECT * FROM SALES
    WHERE qty > 200 AND sales.partnum=titles.partnum)
```

Figure 1-6: An EXISTS operator example.

In this example, the query tests whether 200 copies of a book have been sold in a single sale. If records of such a sale exist, the query displays the part number and the book title. The outer query displays the part number and title of the book. The query then uses the `EXISTS` operator in the search condition of the outer query to verify whether the subquery generates any records. The subquery tests each record in the Sales table to check whether the quantity value in the record is greater than 200. If the quantity is greater than 200, SQL Server returns the partnum and bktitle columns for the book.



Sales

	ordnum	sldate	qty	custnum	partnum	repid
1	00101	2013-11-16 00:00:00	220	20503	40125	N01
2	00102	2013-11-20 00:00:00	100	8802	40232	N02
3	00103	2013-11-20 00:00:00	170	9989	40641	N02
4	00104	2013-12-07 00:00:00	100	9989	40562	N02
5	00105	2013-12-14 00:00:00	150	20493	40481	N01

Titles

	partnum	bktitle	devcost	slprice	pubdate
1	39843	Clear Cupboards	15055.50	49.95	2012-08-19 00:00:00
2	39905	Developing Mobile Apps	19990.00	45.00	2013-01-01 00:00:00
3	40121	Boating Safety	15421.81	36.50	2013-05-18 00:00:00
4	40122	Sailing	9932.96	29.15	2013-05-03 00:00:00
5	40123	The Sport of Windsurfing	12798.32	38.50	2012-07-13 00:00:00
6	40124	The Sport of Hang Gliding	15421.81	49.68	2013-01-06 00:00:00

Exists = True

	ordnum	sldate	qty	custnum	partnum	repid
1	00101	2013-11-16 00:00:00	220	20503	40125	N01
2	00109	2013-01-12 00:00:00	250	8802	40231	N02
3	00110	2013-01-12 00:00:00	250	20330	40482	S03
4	00113	2013-01-20 00:00:00	400	20417	40614	W01
5	00118	2013-01-29 00:00:00	240	20503	40526	N01
6	00130	2008-02-26 00:00:00	330	9881	40812	E01

Results

	partnum	bktitle
1	40122	Sailing
2	40125	The Complete Football Reference
3	40231	How to Play Piano (Beginner)
4	40321	Starting a Small Garden
5	40322	Starting a Greenhouse
6	40325	The Complete Guide to Vegetables

Figure 1-7: Results of the EXISTS operator.

Access the Checklist tile on your LogicalCHOICE course screen for reference information and job aids on How to Search Based on the Existence of Records.

ACTIVITY 1-4
Searching Based on the Existence of a Record

Before You Begin

- Ensure SQL Server Management Studio is open.
- On the **Standard** toolbar, select **New Query**.
- On the **SQL Editor** toolbar, in the **Database** drop-down list, ensure that the FullerAckerman database is selected.

Scenario

The Fuller & Ackerman Publishing bookstore has instituted Gold Card and Silver Card schemes to award discounts to customers. Customers holding Gold Cards can receive a 25 percent discount on any purchase from the bookstore. Silver Card holders can receive a 15 percent discount on purchases. To make the plans successful, sales representatives have been asked to focus on selling books to customers who hold either of the cards. The sales manager wants to identify and reward sales representatives who have sold books to Gold Card customers. You need to generate a list of such sales representatives and submit the report to the sales manager. You know that a Gold Card customer typically buys more than 400 books in a single purchase. Use the Slspers and Sales tables to generate the required output. For information on the tables and column names in the FullerAckerman database, refer to the table structure in Appendix A.

1. Which is the appropriate logic to generate the list of sales representatives who have sold books to Gold Card customers? (Choose two.)

   ☐ Create an inner query that generates records from the Sales table where the quantity of each sale is higher than 400 and where the representative IDs of the Sales and the Slspers tables match.

   ☐ Create an inner query that generates records from the Sales table where the quantity of each sale is less than 400.

   ☐ Create an outer query that verifies whether the records generated by the inner query match with the records to be retrieved.

   ☐ Create an outer query that verifies whether the records generated by the inner query exist. Based on this check, the outer query must generate the records with information on each sales representative.

2. Frame the outer query to generate the representative IDs (repid), and names of sales representatives (fname and lname columns) from the Slspers table.
   a) Enter the code for the outer query to generate the representative IDs and names of sales representatives from the Slspers table:

      ```
      SELECT repid, fname, lname
      FROM slspers
      ```

   b) Enter the code to specify a search condition in the outer query with the `EXISTS` keyword to search for the existence of the records generated by the inner query:

      ```
      SELECT repid, fname, lname
      FROM slspers
      WHERE EXISTS
      ```

3. Frame the inner query to retrieve all records from the Sales table with a quantity sold greater than 400 and where the representative ID of the Sales table matches the representative ID in the Slspers table.
   a) Enter the code for the inner query to retrieve all records from the Sales table:

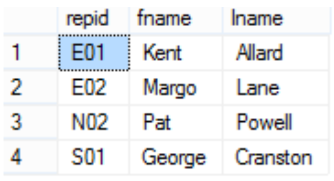
```
SELECT repid, fname, lname
FROM slspers
WHERE EXISTS
   (SELECT * FROM sales
```

b) Type the code to specify the search condition for the inner query to retrieve those records where the sale quantity (qty) is above 400 and the representative ID (repid) of the Sales table matches the representative ID in the Slspers table:

```
SELECT repid, fname, lname
FROM slspers
WHERE EXISTS
   (SELECT * FROM sales
    WHERE qty>400 AND sales.repid=slspers.repid)
```

c) Execute the query.

d) Observe the results. Four sales representatives have sold books to Gold Card customers.



	repid	fname	lname
1	E01	Kent	Allard
2	E02	Margo	Lane
3	N02	Pat	Powell
4	S01	George	Cranston

e) Close the **Query Editor** window without saving the query.

TOPIC D

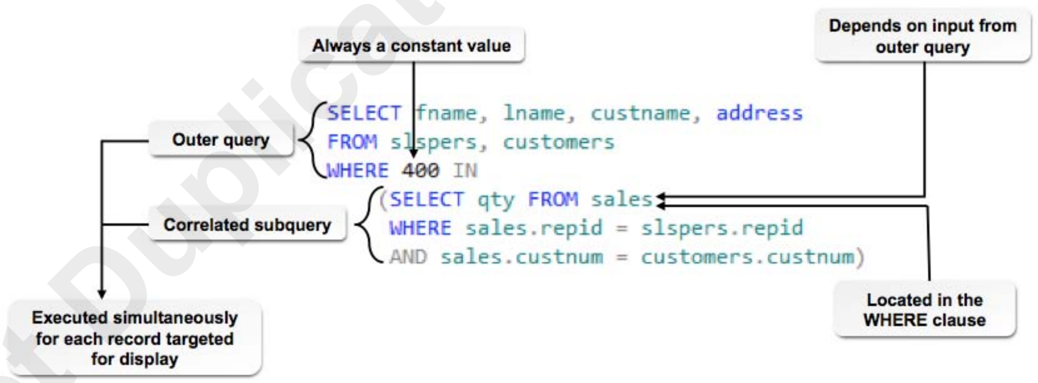
Generate Output Using Correlated Subqueries

You've used subqueries to display records based on unknown search values. Sometimes, based on business requirements, you might want SQL Server to execute the outer query first before executing the subquery. In this topic, you will use correlated subqueries so that SQL Server executes the outer query initially, followed by the subquery.

The human resources manager of an organization wants you to identify employees whose salaries are lower than the average salary in their departments. To find this information, the manager asks you to create a list containing all employees' names, salaries, and departments, and another list with the average salary for each department. If you use this strategy, you would have to manually compare the information in both lists to find the employees who are earning below-average salaries. Or, you could use a SQL-correlated subquery to generate the list of employees whose salaries are below average for you.

Correlated Subqueries

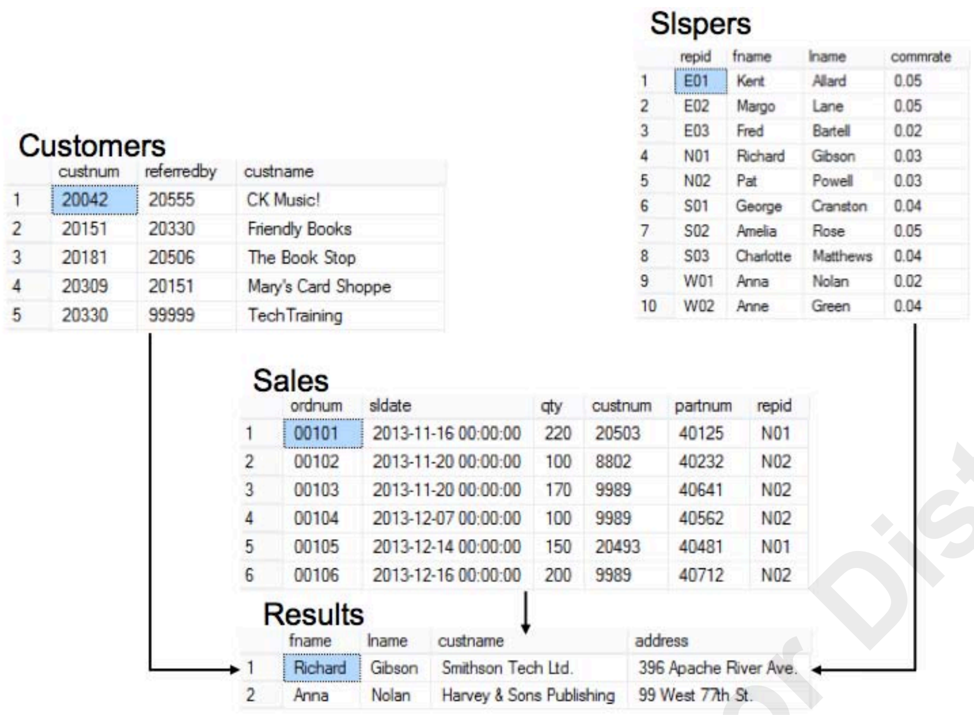
A *correlated subquery* is a subquery that SQL Server executes simultaneously with the outer query. Unlike other subqueries, the correlated subquery depends on the input from the outer query to return values. You always write a correlated subquery in the search condition of an outer query. The value to be searched, specified in the `WHERE` clause of the outer query, must always be a constant value and not a column name.


Always a constant value
Depends on input from outer query
Outer query
Correlated subquery
Executed simultaneously for each record targeted for display
Located in the WHERE clause

```
SELECT fname, lname, custname, address
FROM slspers, customers
WHERE 400 IN
  (SELECT qty FROM sales
   WHERE sales.repid = slspers.repid
   AND sales.custnum = customers.custnum)
```

Figure 1-8: A correlated subquery example.

In this example, the query displays the names of customers who bought 400 copies of a book from a bookstore in a single purchase along with the name of the sales representative who initiated the sale. You use the outer query to display the customer and sales representative names. The outer query selects a record in the Slspers and Customers tables and provides a part number as input to the subquery's condition. When the subquery's condition is satisfied, the subquery returns a list of quantity values from the Sales table. The outer query then searches for the value 400 in the list and displays the targeted record if the search value is found in the list.



Slspers

	repid	fname	lname	commrate
1	E01	Kent	Allard	0.05
2	E02	Margo	Lane	0.05
3	E03	Fred	Bartell	0.02
4	N01	Richard	Gibson	0.03
5	N02	Pat	Powell	0.03
6	S01	George	Cranston	0.04
7	S02	Amelia	Rose	0.05
8	S03	Charlotte	Matthews	0.04
9	W01	Anna	Nolan	0.02
10	W02	Anne	Green	0.04

Customers

	custnum	referredby	custname
1	20042	20555	CK Music!
2	20151	20330	Friendly Books
3	20181	20506	The Book Stop
4	20309	20151	Mary's Card Shoppe
5	20330	99999	TechTraining

Sales

	ordnum	sldate	qty	custnum	partnum	repid
1	00101	2013-11-16 00:00:00	220	20503	40125	N01
2	00102	2013-11-20 00:00:00	100	8802	40232	N02
3	00103	2013-11-20 00:00:00	170	9989	40641	N02
4	00104	2013-12-07 00:00:00	100	9989	40562	N02
5	00105	2013-12-14 00:00:00	150	20493	40481	N01
6	00106	2013-12-16 00:00:00	200	9989	40712	N02

Results

	fname	lname	custname	address
1	Richard	Gibson	Smithson Tech Ltd.	396 Apache River Ave.
2	Anna	Nolan	Harvey & Sons Publishing	99 West 77th St.

Figure 1-9: Results of a correlated subquery.

Process of Correlating Queries

The process of correlating an outer query with an inner query consists of six stages. These stages are detailed in the following table.

Stage	*Description*
Stage 1	The outer query selects a record for processing.
Stage 2	The outer query provides the information related to the selected record to the search condition of the subquery.
Stage 3	The correlated subquery uses this information to return values to the outer query.
Stage 4	The outer query compares the constant value in its search condition with the values returned by the subquery.
Stage 5	If the value being searched is found, the outer query displays the record it had selected. If the value is not found, the outer query does not display records.
Stage 6	The outer query selects the next record for processing.

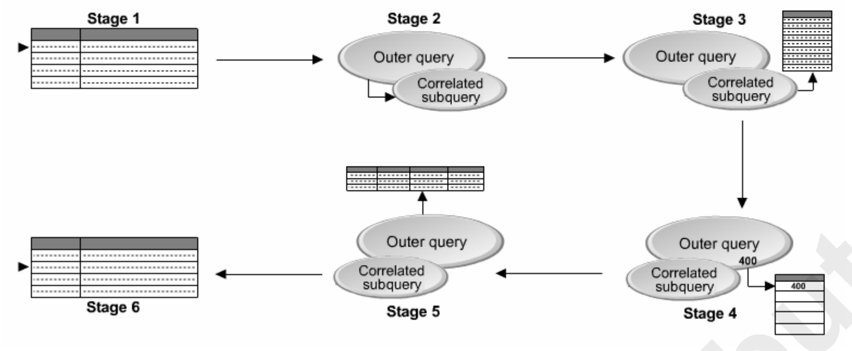
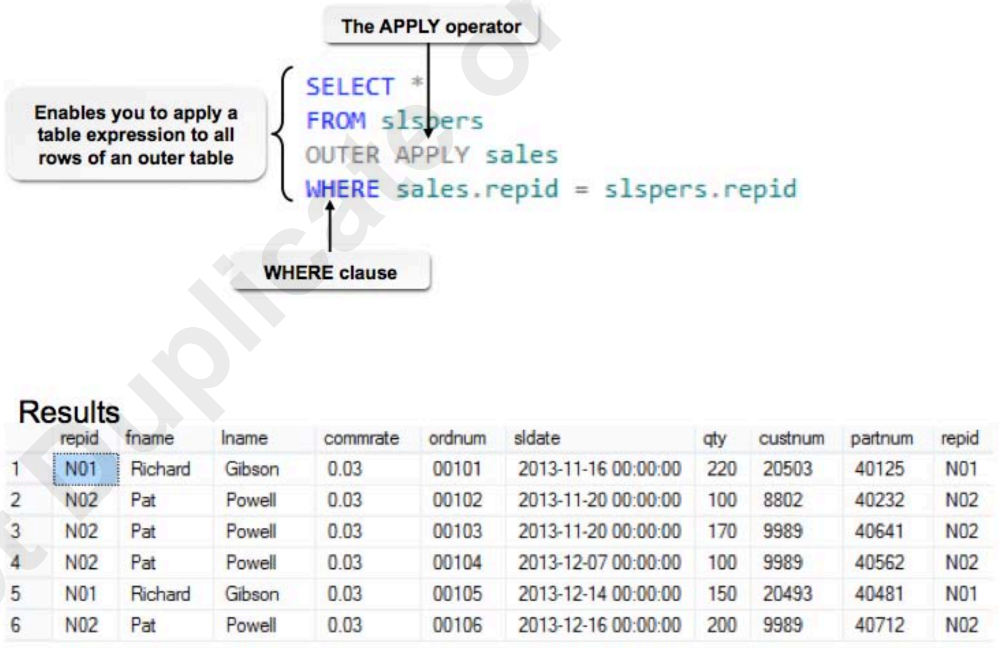


Figure 1-10: The correlating queries process.

The APPLY Operator

The *APPLY operator* is a relational operator that enables you to apply a table expression to all rows of an outer table. The `APPLY` operator can be used with multiple tables and can contain a `WHERE` condition. The output of a query using an `APPLY` operator is similar to a cross join, and the table expression could be a table, a view, a derived table, or even a function that returns a table.


The APPLY operator
Enables you to apply a table expression to all rows of an outer table
WHERE clause
Results

```
SELECT *
FROM slspers
OUTER APPLY sales
WHERE sales.repid = slspers.repid
```

	repid	fname	lname	commrate	ordnum	sldate	qty	custnum	partnum	repid
1	N01	Richard	Gibson	0.03	00101	2013-11-16 00:00:00	220	20503	40125	N01
2	N02	Pat	Powell	0.03	00102	2013-11-20 00:00:00	100	8802	40232	N02
3	N02	Pat	Powell	0.03	00103	2013-11-20 00:00:00	170	9989	40641	N02
4	N02	Pat	Powell	0.03	00104	2013-12-07 00:00:00	100	9989	40562	N02
5	N01	Richard	Gibson	0.03	00105	2013-12-14 00:00:00	150	20493	40481	N01
6	N02	Pat	Powell	0.03	00106	2013-12-16 00:00:00	200	9989	40712	N02

Figure 1-11: An APPLY operator example.

In this example, SQL Server generates a results set that consists of all columns from the Slspers table combined with all columns in the Sales table where the sales representative ID (repid) is the same in both tables.

APPLY Operator and Table-Valued Functions

The `APPLY` operator can be used with a table-valued function, where the function can accept one or more columns as parameters from a source table. The function uses these parameters to produce a table-valued result set.

Access the Checklist tile on your LogicalCHOICE course screen for reference information and job aids on How to Generate Output Using Correlated Subqueries.

ACTIVITY 1-5
Generating Output Using Correlated Subqueries

Before You Begin

- Ensure SQL Server Management Studio is open.
- On the **Standard** toolbar, select **New Query**.
- On the **SQL Editor** toolbar, in the **Database** drop-down list, ensure that the FullerAckerman database is selected.

Scenario

The sales manager at the Fuller & Ackerman Publishing bookstore has been instructed by top management to encourage high-quantity purchasers to achieve the sales targets for the year. From the data on the sales made in the past, the manager has identified that the maximum quantity purchased by a customer in a single purchase is 500. The manager wants you to list the contact information of customers that have bought 500 books in a single purchase. The manager also wants you to add the titles of the books sold. In addition, the manager would like a list of all salespeople and the titles they have sold. The data required to generate the output is available in the Customers, Sales, and Titles tables. For information on the tables and column names in the FullerAckerman database, refer to the table structure in Appendix A.

1. Which is the possible logic you can use to generate the list of top customers of the bookstore? (Choose two.)

   ☐ Create a subquery with the inner query generating the list of customer numbers and part numbers. Use the output from the inner query in the outer query to generate book titles and customer names.

   ☐ Create a correlated subquery where the outer query generates the book title, customer name, and address when the value 500 is found in a list of values supplied by the inner query.

   ☐ Create a correlated subquery where the inner query generates the list of quantities for those book titles where the customer numbers are matching in the Sales and Customers tables and the part numbers are matching in the Sales and Titles tables.

   ☐ Create a query that generates a list of customer names and book titles.

2. Frame the outer query to generate the part number, book title, name, and address of customers from the Titles and Customers tables.
   a) Enter the code for the outer query to generate the part number, book title, name, and address of customers from the Titles and Customers tables:
   ```
   SELECT partnum, bktitle, custname, address
   FROM titles, customers
   ```
   b) Enter the code to specify a search condition with the `IN` operator to search if the value 500 is present in the list of values generated by the inner query:
   ```
   SELECT partnum, bktitle, custname, address
   FROM titles, customers
   WHERE 500 IN
   ```

3. Create the inner query to generate a list of quantity values represented by “qty” from the Sales table and execute the query.
   a) Enter the code to create the `SELECT` clause for the inner query to generate a list of quantity values represented by the qty column from the Sales table:

```
SELECT partnum, bktitle, custname, address
FROM titles, customers
WHERE 500 IN
   (SELECT qty FROM sales
```

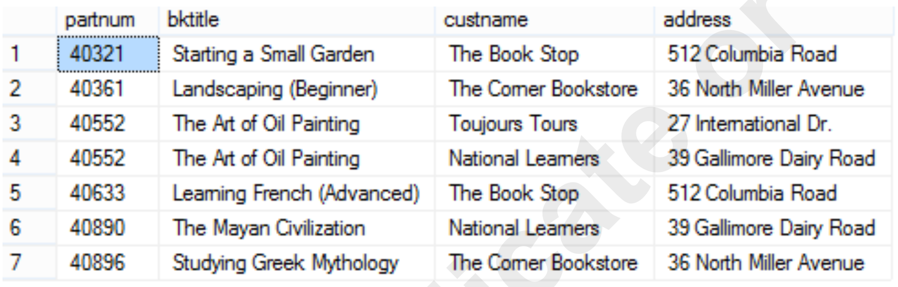
b) Enter the code to specify a search condition for the inner query to retrieve records that store part numbers that are also stored in the Titles table:

```
SELECT partnum, bktitle, custname, address
FROM titles, customers
WHERE 500 IN
   (SELECT qty FROM sales
    WHERE sales.partnum = titles.partnum
```

c) Specify another search condition for the inner query to retrieve records that store customer numbers that are stored in the Sales and Customers tables:

```
SELECT partnum, bktitle, custname, address
FROM titles, customers
WHERE 500 IN
   (SELECT qty FROM sales
    WHERE sales.partnum = titles.partnum
    AND sales.custnum=customers.custnum)
```

d) Execute the query. SQL Server returns seven books and the names of the customers who bought 500 of them.



	partnum	bktitle	custname	address
1	40321	Starting a Small Garden	The Book Stop	512 Columbia Road
2	40361	Landscaping (Beginner)	The Corner Bookstore	36 North Miller Avenue
3	40552	The Art of Oil Painting	Toujours Tours	27 International Dr.
4	40552	The Art of Oil Painting	National Learners	39 Gallimore Dairy Road
5	40633	Learning French (Advanced)	The Book Stop	512 Columbia Road
6	40890	The Mayan Civilization	National Learners	39 Gallimore Dairy Road
7	40896	Studying Greek Mythology	The Corner Bookstore	36 North Miller Avenue

4. Create an OUTER APPLY query to list all salespeople and the books they have sold.

a) Delete the previous query.

b) Type the code to select all rows from the Slspers table using the `OUTER APPLY` operator that have related records in the Sales table. Relate the Slspers table to the Sales table using the repid column:

```
SELECT *
FROM slspers
OUTER APPLY sales
WHERE slspers.repid = sales.repid
```

c) Execute the query. SQL Server returns 98 rows.

d) Close the **Query Editor** window without saving the query.

TOPIC E

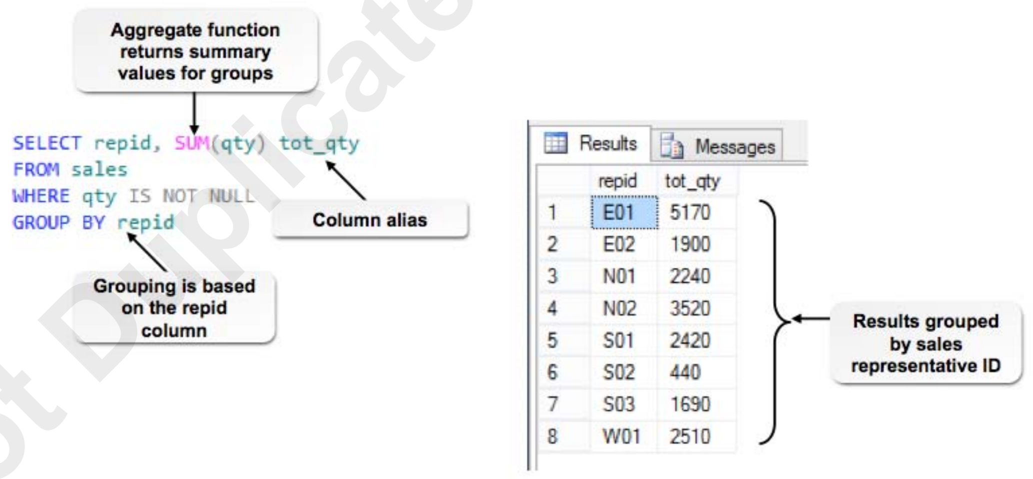
Filter Grouped Data Within Subqueries

On occasion, you might want to group the data returned by a subquery based on specific values. For example, you might want a list of all sales grouped by the salesperson responsible for the sales. In addition, you might want to use a search condition to filter the rows returned by a grouped subquery. For example, you might want a list of all sales, grouped by the salesperson responsible for the sales, and filtered so that you see only those rows where the quantity sold exceeds the average quantity sold.

The sales department manager wants you to identify employees who have worked on teams that met a specific sales target. You already know how to create a list of employees, grouped as teams, together with each team's sales revenue. You could then go through the list and cross out those teams that did not meet the target, or highlight those that did meet the target. But it would save you some extra work if you used a SQL query to filter the list so that the list displays only the teams that exceeded the sales target. Instead of manually filtering through the list of data, you can use SQL groups and filters to do the work for you.

The GROUP BY Clause

GROUP BY is a clause you can use to group rows in the output based on the content of one or more columns. The column(s) you specify in the `GROUP BY` clause must be included in the `SELECT` statement. Add an `ORDER BY` clause to sort the output so that SQL Server groups the rows properly.


Aggregate function returns summary values for groups
Column alias
Grouping is based on the repid column
Results
Messages
Results grouped by sales representative ID

```
SELECT repid, SUM(qty) tot_qty
FROM sales
WHERE qty IS NOT NULL
GROUP BY repid
```

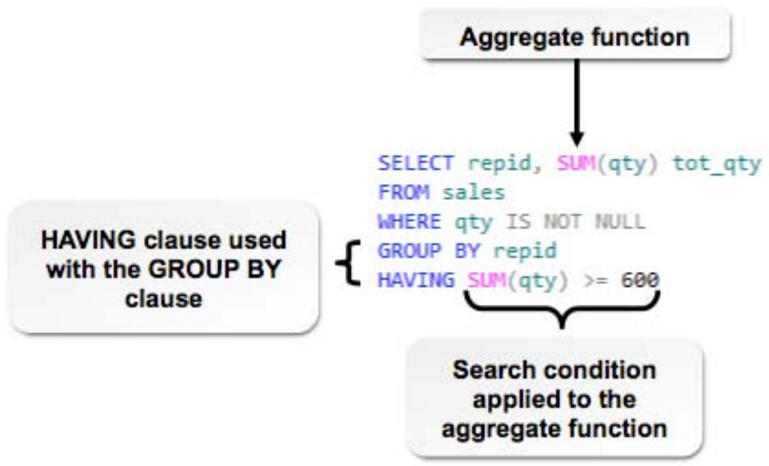
```
   repid  tot_qty
1  E01    5170
2  E02    1900
3  N01    2240
4  N02    3520
5  S01    2420
6  S02    440
7  S03    1690
8  W01    2510
```

Figure 1-12: A GROUP BY clause example.

In this example, SQL Server groups the rows in the Sales table by sales representative ID. It then calculates the total quantity of the sales for the group. In other words, this query enables you to determine the total sales quantity for each sales representative.

The HAVING Clause

You can use the *HAVING clause* to specify a search condition based on an aggregate value. (You should use a `WHERE` clause if you want to search based on one of the columns in the `SELECT` statement.) You use the `HAVING` clause with the `GROUP BY` clause. After SQL Server groups and aggregates the data, it applies the conditions in the `HAVING` clause. If you don't use the `GROUP BY` clause in a query, the `HAVING` clause behaves like a `WHERE` clause.



```sql
SELECT repid, SUM(qty) tot_qty
FROM sales
WHERE qty IS NOT NULL
GROUP BY repid
HAVING SUM(qty) >= 600
```

Aggregate function
HAVING clause used with the GROUP BY clause
Search condition applied to the aggregate function

Figure 1-13: A HAVING clause example.

In this example, the `GROUP BY repid` clause groups the output rows based on the sales representative IDs. The `SUM(qty)` aggregate function in the `SELECT` statement calculates the total sales for each sales representative. The `HAVING` clause then restricts the output to only the sales representatives who have sold a total quantity of 600 or more.

Access the Checklist tile on your LogicalCHOICE course screen for reference information and job aids on How to Filter Grouped Data Within Subqueries.

ACTIVITY 1-6
Filtering Grouped Data Within a Subquery

Before You Begin

- Ensure SQL Server Management Studio is open.
- On the **Standard** toolbar, select **New Query**.
- On the **SQL Editor** toolbar, in the **Database** drop-down list, ensure that the FullerAckerman database is selected.

Scenario

The top management of the Fuller & Ackerman Publishing bookstore chain has asked for a 25 percent increase in the sales targets for this quarter. The sales manager of a branch in the store chain had estimated at the beginning of the quarter that each sales representative needs to sell a minimum of 2,375 books to achieve the targets set by the top management. At the end of the quarter, the manager wants you to list the sales representatives who have met or exceeded the target for the quarter. The Slspers and Sales tables contain the data needed to generate the required output. For information on the tables and column names in the FullerAckerman database, refer to the table structure in Appendix A.

1. **Which options represent the possible logic needed to generate the list of sales representatives who have equaled or exceeded the target? (Choose two.)**

   ☐ Create an outer query that searches for the representative ID returned by the inner query.

   ☐ Create an inner query that returns a list of representative IDs. The representative IDs must be listed after grouping them based on the repid column and after filtering the grouped IDs based on the condition that the total quantity values sold by a representative is greater than or equal to 2,375.

   ☐ Create an outer query that searches for the total quantity values for each representative returned by the inner query.

   ☐ Create an inner query that generates a list of representative IDs after grouping the records based on the qty column.

2. Frame the outer query to generate the representative IDs and names of the sales representatives from the Slspers table.

   a) Enter the code to generate the representative IDs and names of the sales representatives from the Slspers table:

   ```
   SELECT repid, fname, lname
   FROM slspers
   ```

   b) Enter the code to specify the search condition for the outer query to search for the representative ID in a list of values:

   ```
   SELECT repid, fname, lname
   FROM slspers
   WHERE repid IN
   ```

3. Create the inner query to generate a list of representative IDs from the Sales table and execute the query.

   a) Enter the code for the inner query to generate a list of representative IDs from the Sales table:

   ```
   SELECT repid, fname, lname
   FROM slspers
   ```

```
WHERE repid IN
   (SELECT repid FROM sales
```

b) Write the inner query with the `GROUP BY` and `HAVING` clauses to group the output of the inner query based on representative IDs and to filter the grouped output based on the condition that the total quantity value in each record is greater than or equal to 2,375:

```
SELECT repid, fname, lname
FROM slspers
WHERE repid IN
   (SELECT repid FROM sales
    GROUP BY repid
    HAVING SUM(qty) >= 2375)
```

c) Execute the query.

d) Observe the list of four sales representatives who have met or exceeded sales targets.

e) Close the **Query Editor** window without saving the query.

TOPIC F

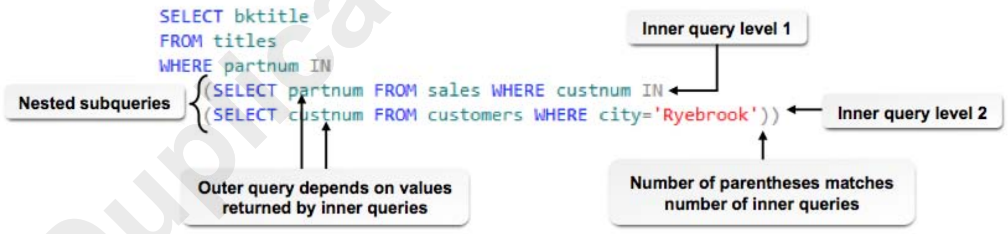
Perform Multiple-Level Subqueries

You have created a subquery that is written within an outer query. When a report based on a complicated analysis of data is required, you may need to use multiple levels of inner queries within a subquery. In this topic, you will create subqueries that contain multiple levels of inner queries.

Let's say you have ordered a burger at a fast food restaurant. A team works together to assemble your burger. One person grills the burger, another adds it to the bun, another adds lettuce and tomato, and the next one wraps it up. The work done by each person becomes the input for the next person, and to get you the final product, the tasks must be performed in the correct sequence. Some queries contain many levels of inner queries, with each inner query depending on the results from another inner query, similar to the line of people working on a burger. The result provided by each level of inner query to its higher level helps the outer query to display the required records.

Nested Subqueries

A *nested subquery* is a subquery that contains multiple levels of inner queries. Each inner query in a nested subquery is enclosed within parentheses. The number of closing parentheses at the end of the nested subquery must match the number of inner queries contained within the subquery. An unlimited number of inner queries can be contained within a nested subquery. Nested subqueries are used in situations where an outer query requires the values returned by many levels of subqueries to display the required records.

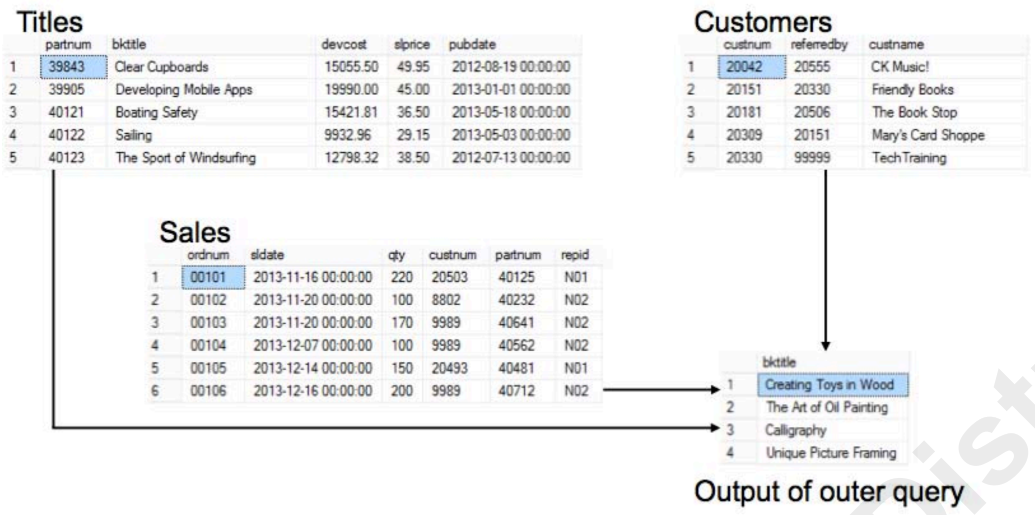


```
SELECT bktitle
FROM titles
WHERE partnum IN
(SELECT partnum FROM sales WHERE custnum IN
(SELECT custnum FROM customers WHERE city='Ryebrook'))
```

Inner query level 1
Nested subqueries
Inner query level 2
Outer query depends on values returned by inner queries
Number of parentheses matches number of inner queries

Figure 1-14: A nested subqueries example.

In this example, you want to display the names of books bought by customers living in the city of Ryebrook. The outer query displays the book title and checks for the part number in the output provided by the subquery. This subquery generates a list of part numbers based on the customer numbers present in the list returned by its inner query. The second inner query returns a list of customers living in the city of Ryebrook.



	partnum	bktitle	devcost	slprice	pubdate
1	39843	Clear Cupboards	15055.50	49.95	2012-08-19 00:00:00
2	39905	Developing Mobile Apps	19990.00	45.00	2013-01-01 00:00:00
3	40121	Boating Safety	15421.81	36.50	2013-05-18 00:00:00
4	40122	Sailing	9932.96	29.15	2013-05-03 00:00:00
5	40123	The Sport of Windsurfing	12798.32	38.50	2012-07-13 00:00:00

	custnum	referredby	custname
1	20042	20555	CK Music!
2	20151	20330	Friendly Books
3	20181	20506	The Book Stop
4	20309	20151	Mary's Card Shoppe
5	20330	99999	TechTraining

	ordnum	sldate	qty	custnum	partnum	repid
1	00101	2013-11-16 00:00:00	220	20503	40125	N01
2	00102	2013-11-20 00:00:00	100	8802	40232	N02
3	00103	2013-11-20 00:00:00	170	9989	40641	N02
4	00104	2013-12-07 00:00:00	100	9989	40562	N02
5	00105	2013-12-14 00:00:00	150	20493	40481	N01
6	00106	2013-12-16 00:00:00	200	9989	40712	N02

	bktitle
1	Creating Toys in Wood
2	The Art of Oil Painting
3	Calligraphy
4	Unique Picture Framing

Figure 1-15: The results of a nested subquery.

Access the Checklist tile on your LogicalCHOICE course screen for reference information and job aids on How to Perform Multiple-Level Subqueries.

ACTIVITY 1-7
Generating Output Using Nested Subqueries

Before You Begin

- Ensure SQL Server Management Studio is open.
- On the **Standard** toolbar, select **New Query**.
- On the **SQL Editor** toolbar, in the **Database** drop-down list, ensure that the FullerAckerman database is selected.

Scenario

With the holiday season fast approaching, the sales manager at the Fuller & Ackerman Publishing bookstore wants to send gifts or discount coupons as incentives to customers. The manager wants to focus on those customers who have purchased high-priced books, and has determined from the price list in the store that any book that is priced above $49 can be classified under the high-price category. To mail the incentives, the manager wants you to generate the name and address details of customers who have made high-price purchases. The Customers, Sales, and Titles tables contain the data needed to generate the required output. For information on the tables and column names in the FullerAckerman database, refer to the table structure in Appendix A.

---

1. **True or False? The most practical method of generating the required customer details is to create a query to identify books that are priced above $49, create another query to identify the customer of each of the books generated by the first query, and use the result of both queries to identify the customer details manually.**

   ☐ True

   ☐ False

2. **True or False? To generate the required customer details, create a nested subquery with the innermost level of the subquery generating the list of part numbers of books that are priced above $49, a second level of the subquery that uses the part numbers to list the customer numbers of customers who bought the books, and an outer query that uses the customer numbers.**

   ☐ True

   ☐ False

3. Frame the outer query to generate the customer number, customer name, address, and city from the Customers table of the nested subquery.
   a) Enter the code for the outer query to generate the customer number, customer name, address, and city from the Customers table:
   ```
   SELECT custnum, custname, address, city
   FROM customers
   ```
   b) Enter the code to specify the search condition for the outer query to search for customer numbers from a list of values:
   ```
   SELECT custnum, custname, address, city
   FROM customers
   WHERE custnum IN
   ```

4. Frame the inner levels of the subquery to generate the customer numbers from the Sales table for the part numbers of books that are priced above $49.

a) Enter the code for the first level of the subquery to generate the customer numbers from the Sales table. The search condition for this inner query must search for the part numbers from a list of values:

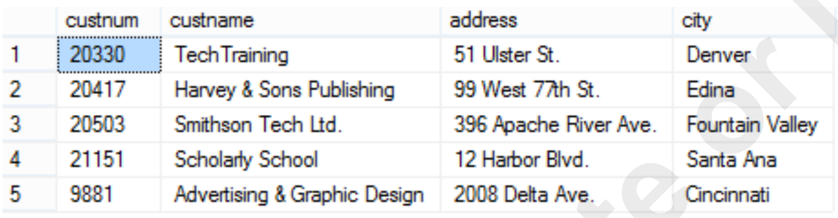
```
SELECT custnum, custname, address, city
FROM customers
WHERE custnum IN
    (SELECT custnum FROM sales WHERE partnum IN
```

b) Write the second level of the subquery to generate the part numbers of books that are priced above $49:

```
SELECT custnum, custname, address, city
FROM customers
WHERE custnum IN
    (SELECT custnum FROM sales WHERE partnum IN
    (SELECT partnum FROM titles WHERE slprice>49))
```

Note: Ensure that you provide two closing parentheses at the end of the subquery.

c) Execute the query and observe the customer numbers and the part numbers of five books that are priced above $49.



	custnum	custname	address	city
1	20330	Tech Training	51 Ulster St.	Denver
2	20417	Harvey & Sons Publishing	99 West 77th St.	Edina
3	20503	Smithson Tech Ltd.	396 Apache River Ave.	Fountain Valley
4	21151	Scholarly School	12 Harbor Blvd.	Santa Ana
5	9881	Advertising & Graphic Design	2008 Delta Ave.	Cincinnati

d) Close the **Query Editor** window without saving the query.

Summary

Using subqueries enables you to search for rows when you don't know the value or group of values on which you want to search. Subqueries help you retrieve complex information from a database based on your business needs.

Which type of query do you think would have a longer execution time? Why?

What kind of query do you think would be best to use when you need to retrieve the most intricate details from large records of data?

	Note: Check your LogicalCHOICE Course screen for opportunities to interact with your classmates, peers, and the larger LogicalCHOICE online community about the topics covered in this course or other topics you are interested in. From the Course screen you can also access available resources for a more continuous learning experience.

2 Manipulating Table Data

Lesson Time: 50 minutes

Lesson Objectives

In this lesson, you will manipulate table data by inserting and updating records in a table and deleting records from a table. You will:

- Insert data into a table.
- Modify data and delete the records in a table.

Introduction

A database system is only as good as the information it contains. As your organization goes about its daily business, the information you store in its database must change. For example, new customers purchase items from inventory, so you must add the new customers, decrease the inventory, and prepare a shipping list to ship the customer's purchase. The daily business at your organization requires that you know how to insert, update, and delete rows from SQL tables. For this reason, in this lesson, you will learn how to insert, update, and delete rows.

TOPIC A

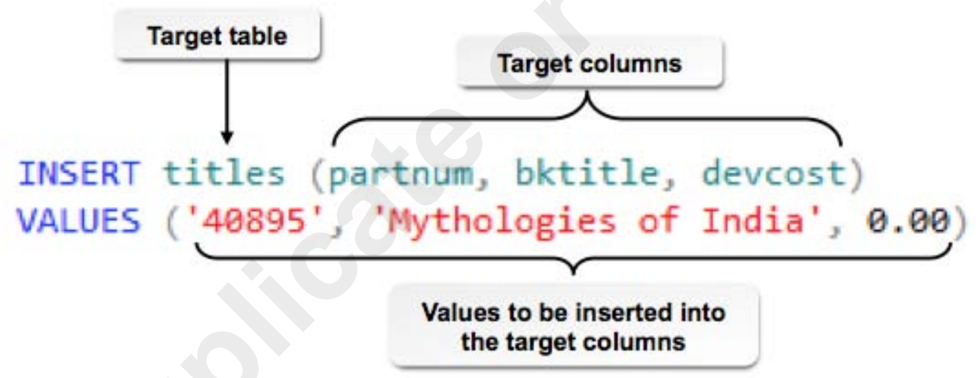
Insert Data

The information in a SQL database constantly changes as your organization goes about doing business. Your organization gains new customers and adds stock to inventory. Salespeople create new sales invoices. As someone who works with the database, you need to know how to add the necessary information to the tables in your organization's database. In this topic, you will learn how to insert rows into tables.

You have drawn up a blank table to plan your work schedule for the week. This table becomes useful to you only when you start filling in each row with information specific to your schedule. Database tables, too, become functional when you populate them with information by inserting rows into them.

The INSERT Statement

You use the *INSERT statement* to insert a record into a table. By executing the `INSERT` statement alone, you can insert only one record at a time.



```
INSERT titles (partnum, bktitle, devcost)
VALUES ('40895', 'Mythologies of India', 0.00)
```

Figure 2-1: Sample code for the INSERT statement.

In this example, the `INSERT` statement adds a record to the Titles table with a part number of 40895, the title "Mythologies of India," and 0.00 for the development cost.

If you want to insert multiple rows into a table, you can use the `INSERT` statement along with a `SELECT` clause. Microsoft® SQL Server® populates the table you specify in the `SELECT` clause of the `INSERT` statement. The order of the columns specified in the `SELECT` clause must match the column order in the `INSERT` statement or in the table from which records are retrieved. You can also filter the records inserted into a table by adding a `WHERE` clause to the `SELECT` clause in the `INSERT` statement.

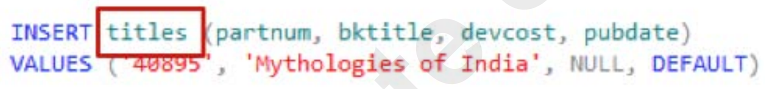
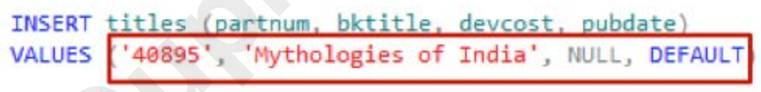
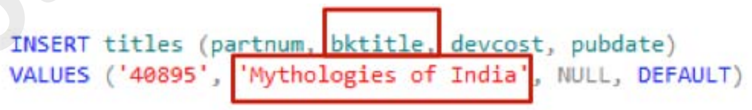
```
INSERT tablename
SELECT columnname1, columnname2, columnname3
FROM tablename
WHERE search condition
```

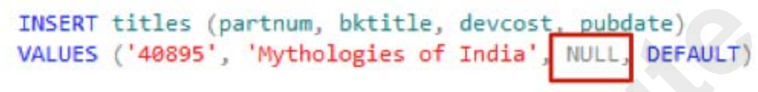
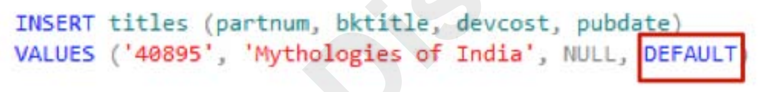
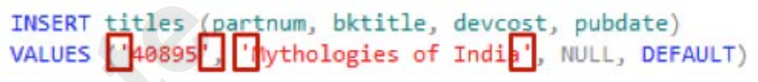
Record Insertion Methods

You can insert records into tables two different ways using the `INSERT` statement with the `VALUES` statement. The methods of record insertion are detailed in the following table.

Method of Record Insertion	*Description*
Inserting all column values	You don't have to specify the column names with the `INSERT` clause. However, if you don't specify the column names, the order of values you put in the `VALUES` clause must match the order in which the columns are organized in the table.
Inserting specific column values	You must specify the required column names after the table name in the `INSERT` clause. In addition, the order in which you write the values in the `VALUES` clause must match the order of column names specified with the `INSERT` clause.

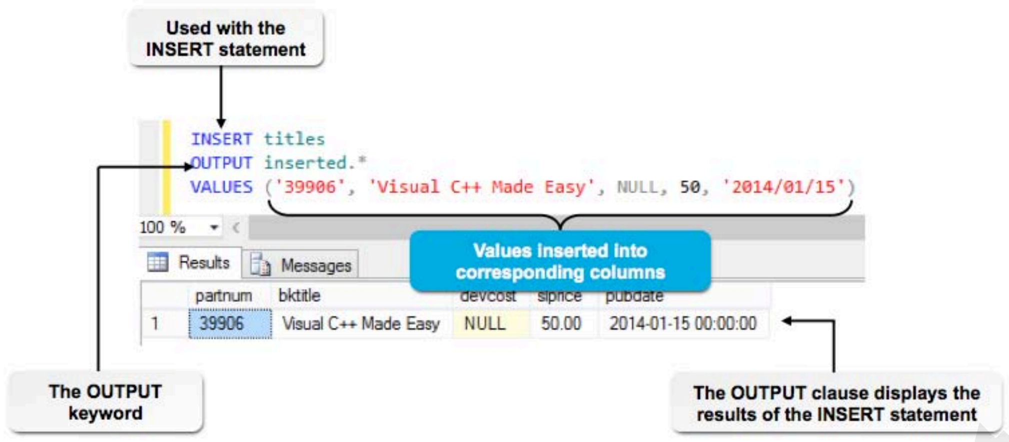
You will have to follow some rules while using the `INSERT` statement. These rules are detailed in the following table.

Rule	*Description*
Table name	You must specify the name of the table targeted for data entry after the `INSERT` statement.  `INSERT titles (partnum, bktitle, devcost, pubdate)` `VALUES ('40895', 'Mythologies of India', NULL, DEFAULT)`
Values to be stored	Specify the values you want to store in the columns with the `VALUES` clause enclosed within parentheses and separated by commas.  `INSERT titles (partnum, bktitle, devcost, pubdate)` `VALUES ('40895', 'Mythologies of India', NULL, DEFAULT)`
Value-data type match	The value you specify for a column must match the column's data type and size.  `INSERT titles (partnum, bktitle, devcost, pubdate)` `VALUES ('40895', 'Mythologies of India', NULL, DEFAULT)`

Rule	*Description*
Null values	If you do not specify a value for a column, SQL Server inserts a null value in the column automatically. If you want to explicitly enter a null value in a column, you need to specify the `NULL` keyword when providing the column values in the `VALUES` clause.  `INSERT titles (partnum, bktitle, devcost, pubdate)` `VALUES ('40895', 'Mythologies of India', NULL, DEFAULT)`
Default values	Some columns can accept default values if you do not explicitly specify a value in the `VALUES` clause. To populate such a column, you need to use the `DEFAULT` keyword in the `VALUES` clause in the place of a constant value.  `INSERT titles (partnum, bktitle, devcost, pubdate)` `VALUES ('40895', 'Mythologies of India', NULL, DEFAULT)`
Single quotes	The values you specify for columns of certain data types, such as `char`, `varchar`, and `datetime`, must be enclosed within single quotes.  `INSERT titles (partnum, bktitle, devcost, pubdate)` `VALUES ('40895', 'Mythologies of India', NULL, DEFAULT)`

The OUTPUT Clause

The *OUTPUT clause* is a SQL clause that enables you to obtain information from rows that are affected by the `INSERT`, `UPDATE`, `DELETE`, and `MERGE` statements. You can use the `OUTPUT` clause only in combination with an `INSERT`, `DELETE`, `UPDATE`, or `MERGE` statement. The `OUTPUT` clause returns data modified using these statements in the form of a table. You might use the results generated by the `OUTPUT` clause to send information to an application so that it can display a message for confirmation or for backing up data.



```
INSERT titles
OUTPUT inserted.*
VALUES ('39906', 'Visual C++ Made Easy', NULL, 50, '2014/01/15')
```

	partnum	bktitle	devcost	slprice	pubdate
1	39906	Visual C++ Made Easy	NULL	50.00	2014-01-15 00:00:00

Used with the INSERT statement
The OUTPUT keyword
100 %
Results
Messages
Values inserted into corresponding columns
The OUTPUT clause displays the results of the INSERT statement

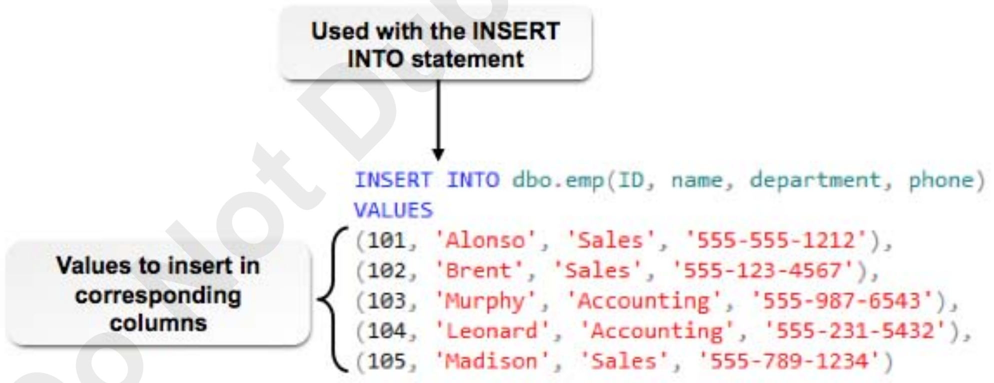
Figure 2-2: Sample code for the OUTPUT statement.

In the example, the `INSERT` statement inserts a row into the Titles table with the values for each column defined in the `VALUES` clause. In addition, the `OUTPUT` clause results in SQL Server displaying the row it inserted in the **Results** pane.

You can also use the `OUTPUT` clause along with a user-defined function to store the output data in a new table. You will have to declare table variables to store these records in the table.

Table Value Constructors

You use table value constructors to specify a set of row value expressions you want to insert into a table. In other words, you can use table value constructors to insert multiple rows into a SQL table without having to type multiple `INSERT` statements. Instead, you use a single `VALUES` clause followed by the values you want to insert into each row enclosed in parentheses and separated by commas.



```
INSERT INTO dbo.emp(ID, name, department, phone)
VALUES
(101, 'Alonso', 'Sales', '555-555-1212'),
(102, 'Brent', 'Sales', '555-123-4567'),
(103, 'Murphy', 'Accounting', '555-987-6543'),
(104, 'Leonard', 'Accounting', '555-231-5432'),
(105, 'Madison', 'Sales', '555-789-1234')
```

Used with the INSERT INTO statement
Values to insert in corresponding columns

Figure 2-3: Sample code for using table value constructors.

In this example, the `INSERT INTO` clause specifies the table in which you want to create the new rows and the columns for which you are going to specify values. The `VALUES` clause then contains values for five new rows, with each row's values enclosed in its own set of parentheses.

Access the Checklist tile on your LogicalCHOICE course screen for reference information and job aids on How to Insert Data.

ACTIVITY 2-1
Inserting Data into a Table

Before You Begin

- Ensure SQL Server Management Studio is open.
- On the **Standard** toolbar, select **New Query**.
- On the **SQL Editor** toolbar, in the **Database** drop-down list, ensure that the FullerAckerman database is selected.

Scenario

When conducting a review of the book titles sold by the Fuller & Ackerman Publishing bookstore, the store manager found only one book under the information technology section. To take advantage of the surge in demand for IT-related book titles, the manager has purchased six new titles for the store. You now have to enter the information related to the new books into the Titles table. You have not yet received the development costs for three of the six titles from the publisher. Refer to the record details provided in the list. For information on the tables and column names in the FullerAckerman database, refer to the table structure in Appendix A.

Book details to be inserted include the following.

Part Number	*Book Title*	*Development Cost*	*Sale Price*	*Date of Publishing*
39906	VC++ Made Easy	Null	50	2013-06-01
39907	History of the Internet	Null	25	2013-07-01
39908	Flash Tips and Tricks	Null	35	2013-08-01
39909	eCommerce: The Future of the Internet	19000	30	2013-09-01
39910	Programming Made Easy	22000	40	2013-10-01
39911	The Essence of Java Programming	20000	29	2013-10-01

1. **True or False? For books where the development cost is unavailable, you need to specify the column names and the corresponding values. For the other books, you need to specify only the values that you want to insert.**
   - ☐ True
   - ☐ False

2. Insert records for the books with part numbers 39906, 39907, and 39908. These books do not have information on their development costs.

   Note: When specifying the values for the sale price, do not use single quotes.

   a) Navigate to the **C:\094006Data\Manipulating Table Data** folder and open the **Insert_Record.txt** file in Notepad. Copy the contents of the Notepad file and paste them in the **Query** window. This `INSERT` statement inserts three rows into the Titles table.
   b) Execute the statements to insert the records.
   c) Close **Notepad**.

3. Insert records for books with part numbers 39909, 39910, and 39911. These books have information on their development costs.
   a) Select the previous query with the mouse and press **Delete** to delete it.
   b) Enter the following query to insert a new record for the book titled “eCommerce: The Future of the Internet”:

```
INSERT titles
VALUES ('39909', 'eCommerce: The Future of the Internet', 19000, 30,
'2013-10-01')
```

   c) Execute the query to insert the record. Verify that the **Results** pane displays the message "(1 row(s) affected)."
   d) Delete the previous query and enter the following query:

```
INSERT INTO titles
VALUES ('39910', 'Programming Made Easy',22000, 40, '2013-10-01'),
('39911', 'The Essence of Java Programming', 20000, 29, '2013-10-01');
```

   e) Execute the query to insert the records. Verify that the **Results** pane displays the message "(2 row(s) affected)."

4. **True or False? To verify whether you have successfully entered the records into the Titles table, you need to query the table to retrieve all records that have part numbers between 39906 and 39911.**

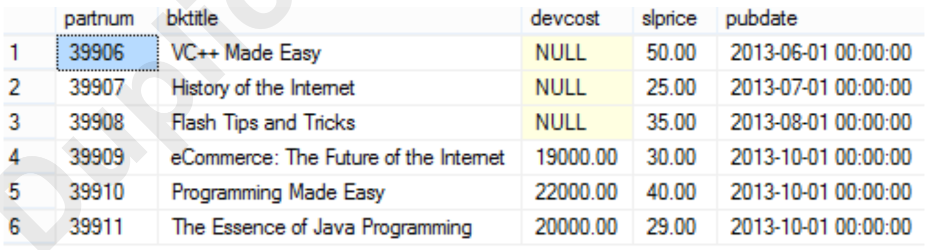
   ☐ True

   ☐ False

5. Verify that you have inserted the records into the Titles table.
   a) Delete the previous query.
   b) Enter the following query:

```
SELECT *
FROM titles
WHERE partnum BETWEEN 39906 AND 39911
```

   c) Execute the query. You should see the six rows you just inserted.



	partnum	bktitle	devcost	slprice	pubdate
1	39906	VC++ Made Easy	NULL	50.00	2013-06-01 00:00:00
2	39907	History of the Internet	NULL	25.00	2013-07-01 00:00:00
3	39908	Flash Tips and Tricks	NULL	35.00	2013-08-01 00:00:00
4	39909	eCommerce: The Future of the Internet	19000.00	30.00	2013-10-01 00:00:00
5	39910	Programming Made Easy	22000.00	40.00	2013-10-01 00:00:00
6	39911	The Essence of Java Programming	20000.00	29.00	2013-10-01 00:00:00

   d) Close the **Query Editor** window without saving the query.

TOPIC B

Modify and Delete Data

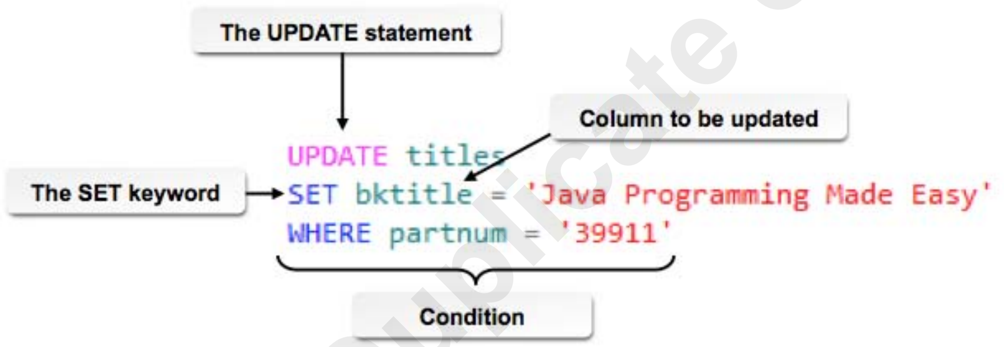
To support the needs of your organization, not only must you know how to insert data into tables, but you also must know how to modify and delete data from tables. In this topic, you will learn how to update rows and delete rows from tables.

As your organization goes about its normal operations, its data needs change. For example, customers move to new locations, inventory changes as the warehouse ships and restocks items, and customers sometimes cancel orders. To properly maintain your organization's database, you must know how to insert, modify, and delete data.

The UPDATE Statement

You use the *UPDATE statement* to modify the data in a column. The statement consists of two parts: an `UPDATE` clause with the name of the table with the column values you want to update, and a `SET` clause that specifies the column name and the new value you want to assign to the column.

Note: You define the default value for a column when you create a table. You can use a default value in any SQL statement by specifying the `DEFAULT` keyword.



```
UPDATE titles
SET bktitle = 'Java Programming Made Easy'
WHERE partnum = '39911'
```

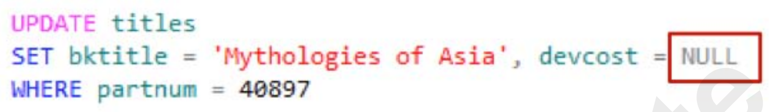
Figure 2-4: Sample code for the UPDATE statement.

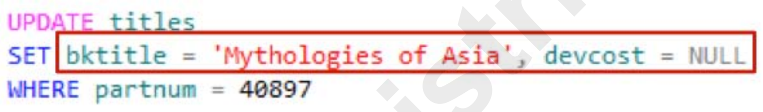
In the example, the `UPDATE` statement modifies the title of the book in the Titles table with the part number of 39911.

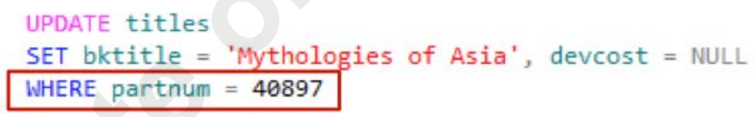
Data Updating Rules

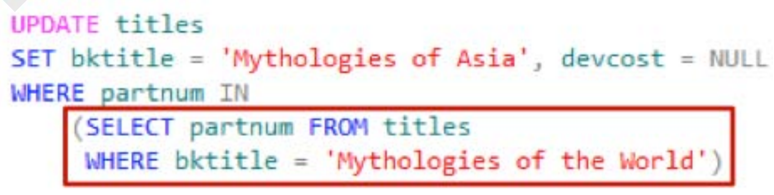
You need to follow specific rules when using an `UPDATE` statement. These rules are detailed in the following table.

Data Updating Rules	Description

Null or default values	You can assign a null value or default value to a column by specifying the `NULL` or `DEFAULT` keyword in the `SET` clause.
	
	`UPDATE titles`
	`SET bktitle = 'Mythologies of Asia', devcost = NULL`
	`WHERE partnum = 40897`

Modifying multiple column values	You can modify multiple column values by specifying the column names and the modified values in the `SET` clause with each column name-value pair separated by a comma.
	
	`UPDATE titles`
	`SET bktitle = 'Mythologies of Asia', devcost = NULL`
	`WHERE partnum = 40897`

Modifying values in specific records	You can modify the column values of a specific record by using a search condition to retrieve the record.
	
	`UPDATE titles`
	`SET bktitle = 'Mythologies of Asia', devcost = NULL`
	`WHERE partnum = 40897`

Subqueries in the `UPDATE` statement	You can use a subquery in the search condition to help in retrieving a targeted record. If you don't use a search condition to retrieve a row, the `UPDATE` statement modifies ALL the column values in the table.
	
	`UPDATE titles`
	`SET bktitle = 'Mythologies of Asia', devcost = NULL`
	`WHERE partnum IN`
	`(SELECT partnum FROM titles`
	`WHERE bktitle = 'Mythologies of the World')`

The DELETE Statement

You use the *DELETE statement* to delete records from a table. Columns cannot be deleted using the `DELETE` statement. You must specify the name of the target table after the `DELETE` statement. By default, the `DELETE` statement deletes all records from a table. However, you can delete specific records by retrieving the targeted records using a `WHERE` clause. You can also use a subquery in the search condition to help retrieve the records that you want to delete.

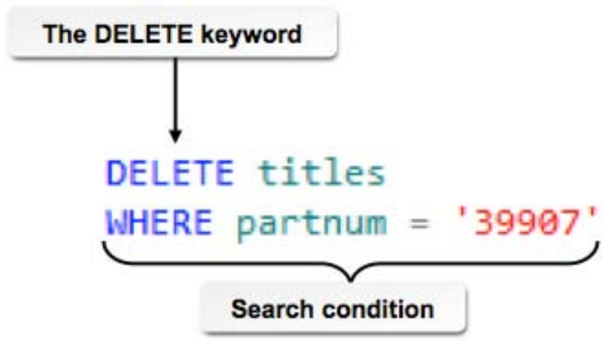


Figure 2-5: Sample code for the DELETE statement.

In this example, the `DELETE` statement deletes the record in the Titles table with the part number of 39907.

The TRUNCATE TABLE Statement

The *TRUNCATE TABLE statement* is a SQL statement that deletes all records from a table without disturbing the table structure. You must specify the name of the table you want to truncate using the `TRUNCATE TABLE` statement. Although the `DELETE` statement can also perform this function, when deleting records from a large table, the `DELETE` statement requires more processing time than the `TRUNCATE TABLE` statement to complete the task. Unlike the `DELETE` statement, the `TRUNCATE TABLE` statement cannot delete specific records.

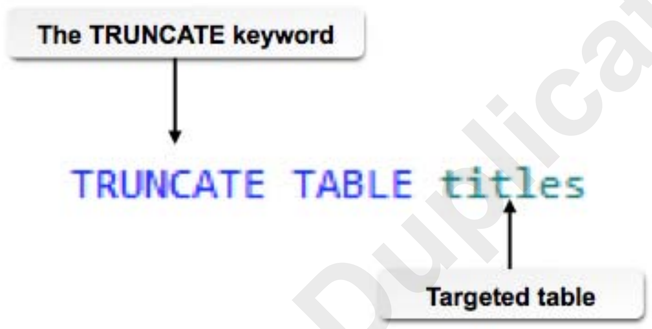


Figure 2-6: Sample code for the TRUNCATE TABLE statement.

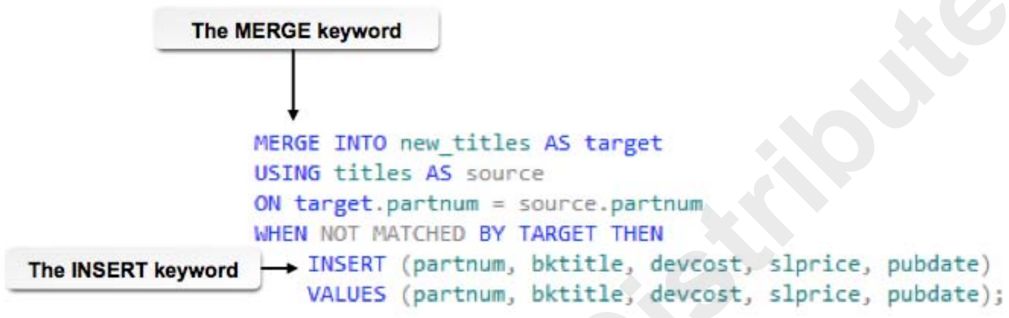
In this example, the `TRUNCATE TABLE` statement deletes all rows in the Titles table.

The MERGE Statement

You use the *MERGE statement* to merge data present in two tables or columns. In SQL Server, you can use a single `MERGE` statement to perform the equivalent actions of the `INSERT`, `UPDATE`, and `DELETE` statements. SQL Server processes a single `MERGE` statement more efficiently than executing individual `INSERT`, `UPDATE`, and `DELETE` statements. When using the `MERGE` statement, you need to specify a source table in the `USING` clause and a target table in the `MERGE INTO` clause. You must terminate a `MERGE` statement with a semicolon. The basic syntax for the `MERGE` statement is:

```
MERGE table_name AS target
USING table_name AS source
ON (condition)
```

```
WHEN MATCHED THEN
  UPDATE SET column1 = value1 [, column2 = value2 ...]
  WHEN NOT MATCHED BY TARGET THEN
     INSERT (column1 [, column2 ...]) VALUES (value1 [, value2 ...])
  WHEN NOT MATCHED BY SOURCE THEN
     DELETE (column1 [, column2 ...]) VALUES (value1 [, value2 ...]);
```



```
MERGE INTO new_titles AS target
USING titles AS source
ON target.partnum = source.partnum
WHEN NOT MATCHED BY TARGET THEN
    INSERT (partnum, bktitle, devcost, slprice, pubdate)
    VALUES (partnum, bktitle, devcost, slprice, pubdate);
```

Figure 2-7: Sample code for the MERGE statement.

The example uses the `MERGE` statement to insert rows into the New_titles table from the Titles table when a matching row does not exist in the New_titles table.

Access the Checklist tile on your LogicalCHOICE course screen for reference information and job aids on How to Modify and Delete Data.

ACTIVITY 2-2
Modifying Data

Before You Begin

- Ensure SQL Server Management Studio is open.
- On the **Standard** toolbar, select **New Query**.
- On the **SQL Editor** toolbar, in the **Database** drop-down list, ensure that the FullerAckerman database is selected.

Scenario

You have added six new book titles to the catalog of the Fuller & Ackerman Publishing bookstore. You recorded the information related to these titles in the Titles table. However, the development cost information was not provided by the publisher for titles with part numbers 39906, 39907, and 39908. The publisher has now requested that you update your Titles table with the development cost of $21,000 for each of those titles.

You have also been informed by the sales representative in the store that the information on one of the new titles with part number 39911 is completely incorrect in the table. He has provided you with the correct information. The book title should be "Java Programming Made Easy." The book's development cost was $25,000, and its sale price is $32. This book was published on 11-01-2013.

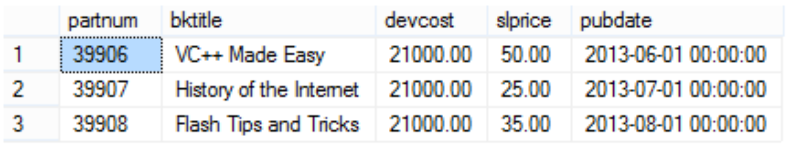
For more information on the tables and column names in the FullerAckerman database, refer to the table structure in Appendix A.

---

1. **Identify the correct logic you should use to set the development cost to $21,000 for titles with part numbers 39906, 39907, and 39908.**
   - ○ Update the value of the devcost column for all records in the Titles table.
   - ○ Update the value of the devcost column for six books that were recently purchased.
   - ○ Create a query to generate a list of all part numbers. Update the devcost column value for part numbers that start with the value 399.
   - ○ Update the value of the devcost column for the records of books with part numbers 39906, 39907, and 39908.

2. Update the records of books that do not have the development cost information and verify whether the update is successful.
   a) Enter the following query to update the development cost:
   ```
   UPDATE titles
   SET devcost=21000
   ```
   b) Add the `WHERE` clause to identify the books for which you want to update the devcost:
   ```
   UPDATE titles
   SET devcost=21000
   WHERE partnum BETWEEN 39906 AND 39908
   ```
   c) Execute the `UPDATE` statement to update the required records.
   d) To verify that your changes took effect, delete the previous query and execute the following query:
   ```
   SELECT *
   FROM titles
   WHERE partnum BETWEEN 39906 AND 39908
   ```

e) Observe that the development costs for the books with part numbers from 39906 to 39908 have been updated to $21000.



	partnum	bktitle	devcost	slprice	pubdate
1	39906	VC++ Made Easy	21000.00	50.00	2013-06-01 00:00:00
2	39907	History of the Internet	21000.00	25.00	2013-07-01 00:00:00
3	39908	Flash Tips and Tricks	21000.00	35.00	2013-08-01 00:00:00

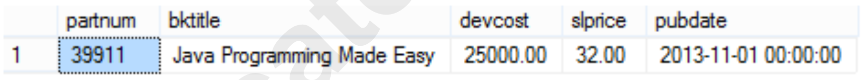
3. Correct the information in the record for the book with the part number 39911 and verify whether you made the changes successfully.
   a) Delete the previous query.
   b) Write a query to update the information in the record for the book with part number 39911. Update the **Book Title**, **Development Cost**, **Sale Price**, and **Date of Publishing** with **'Java Programming Made Easy'**, **25000**, **32.000**, and **'2013–11–01'**, respectively:

```
UPDATE titles
SET bktitle='Java Programming Made Easy', devcost=25000, slprice=32.000,
pubdate='2013/11/01'
WHERE partnum='39911'
```

   c) Execute the query.
   d) Verify that you updated the information for the book with part number 39911. Edit the query to read:

```
SELECT *
FROM titles
WHERE partnum='39911'
```

   e) Execute the `SELECT` statement and view the results.



	partnum	bktitle	devcost	slprice	pubdate
1	39911	Java Programming Made Easy	25000.00	32.00	2013-11-01 00:00:00

   f) Close the **Query Editor** window without saving the query.

ACTIVITY 2-3

Deleting Data from a Table

Before You Begin

- Ensure SQL Server Management Studio is open.
- On the **Standard** toolbar, select **New Query**.
- On the **SQL Editor** toolbar, in the **Database** drop-down list, ensure that the FullerAckerman database is selected.

Scenario

After reviewing the sales reports of various branches, the sales manager at the Fuller & Ackerman Publishing bookstore is concerned that some of the titles were selling much below the targeted numbers. One of the books named "History of the Internet" with the part number 39907 has not sold at all. To comply with an organizational policy, the manager ordered the removal of titles that sold less than 100 copies from the store shelves. In addition, the SQL administrator wants you to empty the Titles1 table but retain the structure of the table. These changes have to be reflected in the Titles and Titles1 tables. For information on the tables and column names in the FullerAckerman database, refer to the table structure in Appendix A.

1. **True or False? To delete the record for the title "History of the Internet", you need to specify the part number of the book as a value for the search condition of the `DELETE` statement.**

   ☐ True

   ☐ False

2. Delete the record for the book with the part number 39907 from the Titles table.
   a) Enter this query:
   ```
   DELETE titles
   WHERE partnum='39907'
   ```
   b) Execute the statement to delete the record.

3. Verify whether you deleted the targeted record from the Titles table.
   a) Edit the query to retrieve the record of the book with the part number 39907 from the Titles table:
   ```
   SELECT *
   FROM titles
   WHERE partnum='39907'
   ```
   b) Execute the `SELECT` statement.
   c) Observe that there is no record with the part number 39907.

4. Truncate the Titles1 table.
   a) Edit the query to read as follows:
   ```
   SELECT *
   FROM titles1
   ```
   b) Press **Enter** twice.
   c) Type a second query:
   ```
   TRUNCATE TABLE titles1
   ```
   d) Select the `SELECT` statement above the `TRUNCATE TABLE` statement with your mouse and then select **Execute** to query the contents of the Titles1 table. This table contains 92 records.

e) Select and execute the `TRUNCATE TABLE` statement. SQL Server deletes all records from the table.
f) Select and execute the `SELECT` statement again. The Titles1 table no longer contains any records.
g) Close the **Query Editor** window without saving the query.

Summary

A SQL database is only as good as the information it contains. During the normal operations of a business, the contents of a business' database must change to reflect new customers, reductions in inventory, and sales. For these reasons, to maintain an accurate database, you must know how to insert, update, and delete data from tables.

What are the benefits of manipulating table data?

Why should you delete unused records or tables in a database at regular intervals?

Note: Check your LogicalCHOICE Course screen for opportunities to interact with your classmates, peers, and the larger LogicalCHOICE online community about the topics covered in this course or other topics you are interested in. From the Course screen you can also access available resources for a more continuous learning experience.

3 Manipulating the Table Structure

Lesson Time: 1 hour, 40 minutes

Lesson Objectives

In this lesson, you will manipulate the table structure. You will:

- Create a table.
- Create a table with constraints.
- Modify a table's structure.
- Back up tables.
- Delete tables.

Introduction

You now know how to create advanced queries and manipulate the contents of tables. But what if the information your organization wants to track changes? For example, what if your company decides to track categories of customers (commercial businesses, individuals, non-profit organizations, and so on)? In order to track this information, you must know how to modify the customer table's structure. Because business needs change, in this lesson, you will learn how to create a new table, modify an existing table, back up a table, and delete a table.

As an organization changes over time, you might need to modify the design of the tables in its database. For example, you might need to add new columns or change the definition of columns. In some instances, you might need to delete columns from a table or entire tables because they are not relevant anymore. In place of obsolete tables, you might need to create new tables that meet the latest data storage requirements. You use SQL to manipulate tables in a database to reflect any of these changes in business requirements.

TOPIC A

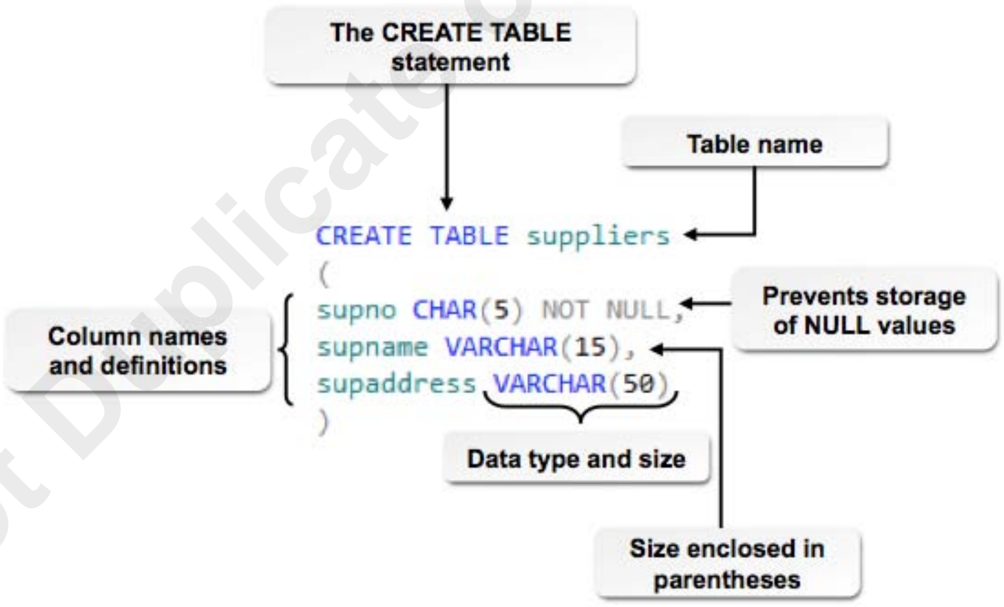
Create a Table

As your organization grows and responds to changes in the marketplace, you might find that it needs to store new information. In order to store this information, you must know how to create a new table. In this topic, you will learn how to create a new table.

If you consider a database to be a community of people, then you can consider tables as individual homes. To establish an active community, you need to effectively design and build the homes in the community. Similarly, to work with the data in a database, you first need to define the structure of a table and then create the table. Defining the table structure involves deciding on the columns and their data types. After determining the table structure, you can create the table by using SQL statements.

The CREATE TABLE Statement

You use the *CREATE TABLE statement* to create the structure of a table. It, however, does not populate the table with data. For example, the following query creates a table named Suppliers with three columns: supplier number (supno), supplier name (supname), and supplier address (supaddress).



```
CREATE TABLE suppliers
(
supno CHAR(5) NOT NULL,
supname VARCHAR(15),
supaddress VARCHAR(50)
)
```

The CREATE TABLE statement
Table name
Prevents storage of NULL values
Column names and definitions
Data type and size
Size enclosed in parentheses

Figure 3-1: A CREATE TABLE statement example.

You need to follow some rules when writing a `CREATE TABLE` statement. These rules are detailed in the following table.

Rule	*Description*
Table name	You must specify the name of the table you want to create after the `CREATE TABLE` statement.

Rule	*Description*
Column definitions	After the table name, specify the names of the columns and their definitions, enclosed within parentheses. Each column definition consists of the data type, size, and any other information you need to specify for the column. You can also specify computed columns while defining a column.
Column size	Depending on the column's data type, you might need to specify the size of the column, also enclosed within parentheses.
Null values	By default, a column can store null values. However, you can restrict the storage of null values in a column by specifying the `NOT NULL` keyword in the column definition.

The INSERT INTO Statement

After you create a table, you can use the `INSERT INTO` statement to insert data into the table. For example, the following two statements insert two rows into the Suppliers table:

```
INSERT INTO suppliers
VALUES('S001','Allendales','1149 Blossom Road');
INSERT INTO suppliers
VALUES('S002','Caldwell','981 Connecticut Blvd');
```

Displaying the Table Structure

Use the `SP_HELP` stored procedure to display a table's structure. The syntax is:

```
SP_HELP tablename1
```

SQL Data Types

Microsoft® SQL Server® 2012 supports a number of data types that you can use when defining the columns of a table. SQL Server includes support for the following categories of data types:

- Exact numeric.
- Approximate numeric.
- Date and time.
- Character strings.
- Binary strings.
- Other data types.

The exact numeric data types enable you to store integers as well as numbers with decimal places. The following table describes the exact numeric data types.

Data Type	*Range of Values Supported*	*Storage Size*
bigint	-2^{63} to $2^{63}-1$	8 bytes
int	-2^{31} to $2^{31}-1$	4 bytes
smallint	-2^{15} to $2^{15}-1$	2 bytes
tinyint	0 to 255	1 byte

Data Type	*Range of Values Supported*	*Storage Size*
bit	1, 0, or NULL	If there are 8 or less bit columns in a table, Microsoft SQL Server stores the columns in 1 byte. If there are 9 - 16 bit columns in a table, SQL Server uses 2 bytes to store them; and so on.
money	-922,337,203,685,477.5808 to 922,337,203,685,477.5807	8 bytes
smallmoney	- 214,748.3648 to 214,748.3647	4 bytes
decimal (*precision*, *scale*)	1 - 9 digits of precision	5 bytes
You must specify the precision (the total number of digits both before and after the decimal point), and the scale (the number of digits after the decimal point). Precision determines the amount of storage the column requires. The *numeric* data type is functionally equivalent to the decimal data type.	10 - 19 digits of precision 20 - 28 digits of precision 29 - 38 digits of precision	9 bytes 13 bytes 17 bytes

The approximate data types enable you to store numbers with a varying number of digits after the decimal point. When you use the approximate data types, you can specify a value that identifies the number of digits you need to store the mantissa (the significant digits) of a number in scientific notation. The following table describes the approximate numeric data types.

Data Type	*Range of Values Supported*	*Storage Size*
float (*n*), where *n* is a number from 1 to 53	- 1.79^{308} to -2.23^{-308}, 0, and 2.23^{-308} to 1.79^{308}	If *n* is 1 - 24, 4 bytes; if *n* is 25 - 53, 8 bytes.
real, which is equivalent to float (24)	-3.40^{38} to -1.18^{-38}, 0, and 1.18^{-38} to 3.40^{38}	4 bytes

You use the date and time data types in SQL Server to store date information, time information, or both. The following table describes the date and time data types you can assign.

Data Type	*Range of Values Supported*	*Storage Size*
date	0001-01-01 through 9999-12-31	3 bytes
datetime	January 1, 1753 through December 31, 9999 00:00:00 through 23:59:59.997	8 bytes
datetime2	0001-01-01 through 9999-12-31 00:00:00 through 23:59:59.9999999	6 bytes for precisions less than 3; 7 bytes for precisions 3 and 4. All other precisions require 8 bytes.
smalldatetime	1900-01-01 through 2079-06-06 00:00:00 through 23:59:59	4 bytes

Data Type	Range of Values Supported	Storage Size
time	00:00:00.0000000 through 23:59:59.9999999	5 bytes
datetimeoffset	0001-01-01 through 9999-12-31 00:00:00 through 23:59:59.9999999	10 bytes

SQL Server includes several data types for storing alphanumeric characters. The `char` and `varchar` data types enable you to store any one of the 255 characters in the American Standard Code for Information Interchange (ASCII) character set. In contrast, the `nchar` and `nvarchar` data types enable you to store any of 65,535 characters in the Unicode or Universal Character Set (UCS)-2 character set. The `nchar` and `nvarchar` data types use two bytes to store each character in the column.

The `char` and `nchar` data types are considered fixed because they consume a fixed amount of storage space regardless of how many characters you store in columns with these data types. For example, if you assign the data type `char(15)` to a column, it consumes 15 bytes of storage space even if it's empty. In contrast, `varchar` and `nvarchar` consume storage space based on the number of characters they contain. For example, a column that has the data type `varchar(20)` will use only 6 bytes to store the value "Fuller" instead of 20 bytes.

Another category of data types is binary. You use the binary data types to store binary data (ones and zeroes). When you assign a binary data type to a column, you must specify the number of bytes you want it to use; you can specify 1 to 8,000 bytes. The following table describes the binary data types you can define.

Data Type	Storage Size
binary(*n*), where *n* is the number of bytes you want to reserve for the column.	1 - 8,000 bytes
varbinary(*n*), where *n* is the maximum number of bytes the column can use. Varbinary is a variable-length data type, so it consumes storage space based on the values stored in the column.	Actual length of the data stored in the column plus two bytes.
image	Variable-length binary data (images) from 0 to 2^{31}-1 bytes.

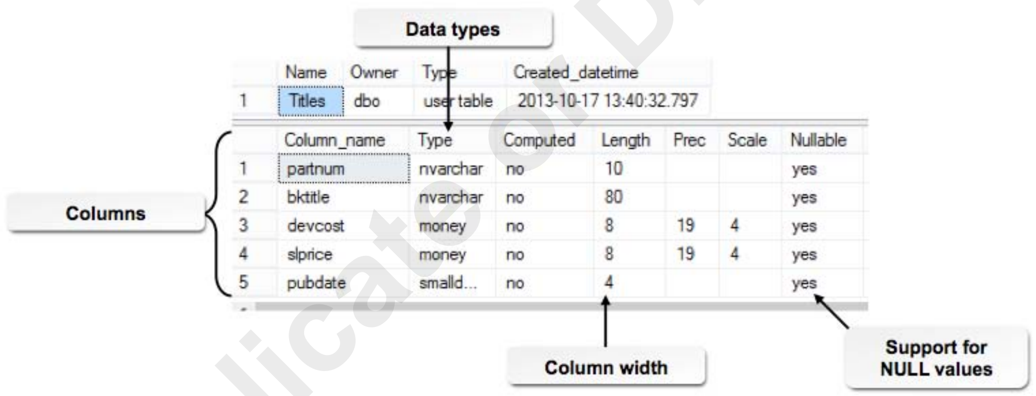
Finally, SQL Server includes other data types for storing a variety of data. The following table describes the other data types.

Data Type	Purpose	Storage Size
timestamp (also known as rowversion)	Use to assign a version identifier in binary to table rows. SQL Server changes this value for each `INSERT` or `UPDATE` statement you execute.	8 bytes
xml	Use to store XML data.	Up to 2 GB
uniqueidentifier	Use to assign a unique ID to each row in a table.	16 bytes

Table Design

Table design refers to how you structure a table with the required columns and rows. Creating a well-defined table design helps in the effective organization and storage of data in a table. To design an effective table, follow these guidelines:

- Identify the business requirement that you need to fulfill by creating the table.
- To determine the columns of a table, decide on the information related to the business requirement that you want to store in the table, and convert these points of information into columns.
- Identify the appropriate data type you should assign to each column. To choose the most appropriate data type for a column, base your decision on the characteristics of the data that you plan to store in the column.
- Decide on the column width based on the likely size of values that you will store in the column.
- Determine whether you need to impose any limitations on the data that is entered into a column. If required, decide on the restrictions that you need to apply to the column.
- Determine the conditions that you must fulfill to meet the business requirements for creating the column. Based on these conditions, create restrictions on the column.



	Name	Owner	Type	Created_datetime
1	Titles	dbo	user table	2013-10-17 13:40:32.797

	Column_name	Type	Computed	Length	Prec	Scale	Nullable
1	partnum	nvarchar	no	10			yes
2	bktitle	nvarchar	no	80			yes
3	devcost	money	no	8	19	4	yes
4	slprice	money	no	8	19	4	yes
5	pubdate	smalld...	no	4			yes

Figure 3-2: A table design example.

For example, at the Fuller & Ackerman Publishing bookstore, you have a business requirement to store information on books so you decide to create a Titles table. You determine that you want to store information about each book's part number, book title, development cost, the selling price, and the publication date in the table. Because you plan to store part numbers of up to 10 characters in the partnum column, you assign it the `nvarchar` data type.

Because you expect to store titles of varying lengths in the bktitle column, you decide to assign the `nvarchar` data type to this column. You specify 80 as the column size for the bktitle column because you expect the column to store values up to a maximum of 80 characters. You also need to assign data types to the slprice column and the devcost column for the book. Because both of these columns store money information, you assign them the `money` data type. Finally, you plan to store dates in the pubdate column so you assign it the `smalldatetime` data type.

Access the Checklist tile on your LogicalCHOICE course screen for reference information and job aids on How to Create a Table.

ACTIVITY 3-1
Creating Tables

Before You Begin

- Ensure SQL Server Management Studio is open.
- On the **Standard** toolbar, select **New Query**.
- On the **SQL Editor** toolbar, in the **Database** drop-down list, ensure that the FullerAckerman database is selected.

Scenario

One of the important points at a monthly meeting of store managers of the Fuller & Ackerman Publishing bookstore chain is the increase in demand for titles that were taken off the shelves from the store. The demand for these titles, classified as obsolete titles, seems to increase when a movie is filmed based on these titles. Using this indicator, store managers feel that they can predict the demand for obsolete titles and prefer to have a mechanism by which such titles can be tracked and procured quickly. The sales manager of the bookstore chain wants you to create a table to record the part number, book title, sale price, and publishing date information related to titles that are taken off the shelves. You now need to create a table called Obsolete_titles1. For information on the tables and column names in the FullerAckerman database, refer to the table structure in Appendix A.

1. Create the Obsolete_titles1 table.
   a) Enter the `CREATE TABLE` portion of the query:
   ```
   CREATE TABLE obsolete_titles1
   (
   )
   ```
   b) In the `CREATE TABLE` statement, click inside the parentheses, and enter the column definition to store part numbers:
   ```
   CREATE TABLE obsolete_titles1
   (
   partnum varchar(10) NOT NULL,
   )
   ```
   c) Enter the code to specify the column definition to store the book titles:
   ```
   CREATE TABLE obsolete_titles1
   (
   partnum varchar(10) NOT NULL,
   bktitle varchar(80),
   )
   ```
   d) Enter the code to specify the column definition to store the development cost:
   ```
   CREATE TABLE obsolete_titles1
   (
   partnum varchar(10) NOT NULL,
   bktitle varchar(80),
   devcost money,
   )
   ```
   e) Enter the code to specify the column definition to store the sale price of obsolete books:
   ```
   CREATE TABLE obsolete_titles1
   (
   partnum varchar(10) NOT NULL,
   ```

```
bktitle varchar(80),
devcost money,
slprice money,
)
```

f) Specify the column definition to store the date of publishing for a book:

```
CREATE TABLE obsolete_titles1
(
partnum varchar(10) NOT NULL,
bktitle varchar(80),
devcost money,
slprice money,
pubdate smalldatetime
)
```

g) Execute the `CREATE TABLE` statement to create the Obsolete_titles1 table.

2. View the table structure.

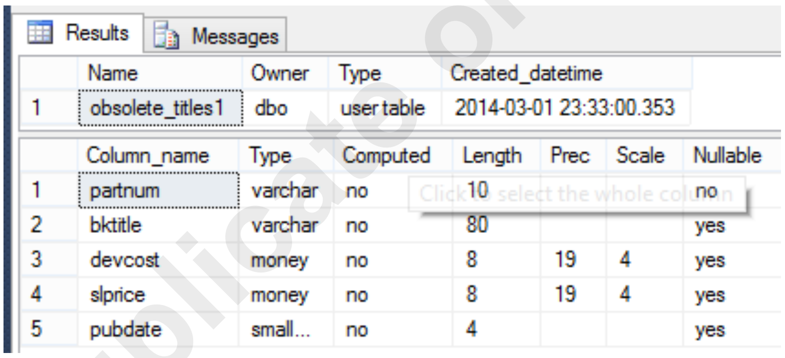
a) Delete the previous query.

b) Enter the command to display the table structure of the Obsolete_titles1 table:

```
SP_HELP obsolete_titles1
```

c) Execute the `SP_HELP` query.

d) In the **Results** pane, view the column definitions for the Obsolete_titles1 table.



	Name	Owner	Type	Created_datetime
1	obsolete_titles1	dbo	user table	2014-03-01 23:33:00.353

	Column_name	Type	Computed	Length	Prec	Scale	Nullable
1	partnum	varchar	no	10			no
2	bktitle	varchar	no	80			yes
3	devcost	money	no	8	19	4	yes
4	slprice	money	no	8	19	4	yes
5	pubdate	small...	no	4			yes

e) Close the **Query Editor** window without saving the query.

TOPIC B

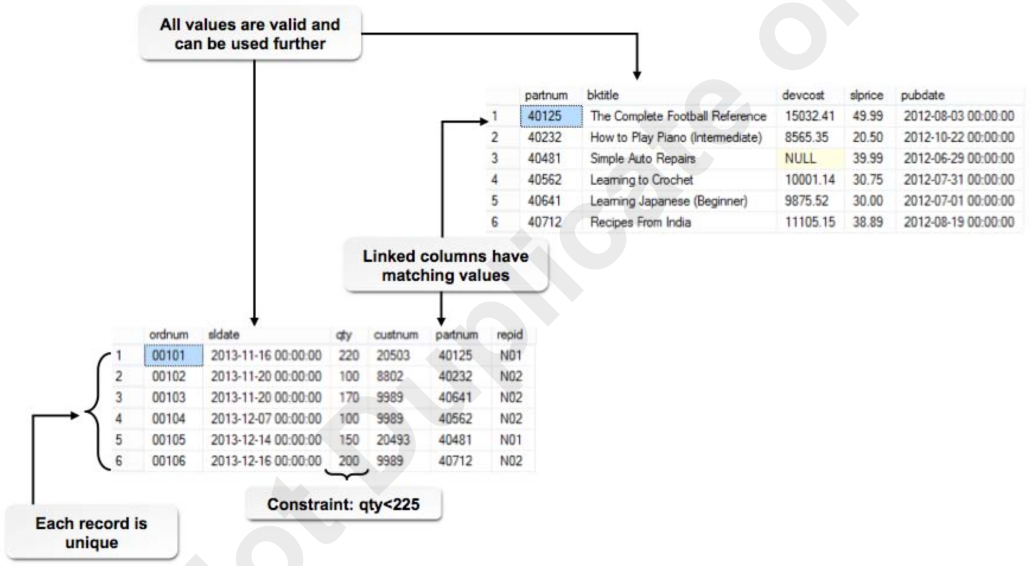
Create a Table with Constraints

A table with a simple structure is sufficient to store data but is not effective in an actual business environment. Typically, you need to add checks and conditions to the table structure to ensure the validity of the data stored in a table. You implement data validation in SQL by adding constraints to columns. In this topic, you will create a table with constraints.

A table without constraints is similar to a home without locks. If you don't implement constraints, it's possible for users to add invalid data to tables. By using constraints, you can prevent incorrect or redundant entries in a column. You can also use constraints to ensure that only specific values are stored in a column.

Data Integrity

Data integrity is the state of information in a database where all values stored in the database are correct. The integrity of data in a database indicates whether the values are valid and whether they can be used further. Data integrity can be classified into entity integrity, domain integrity, referential integrity, and user-defined integrity.


All values are valid and can be used further
Linked columns have matching values
Constraint: qty<225
Each record is unique

	partnum	bktitle	devcost	slprice	pubdate
1	40125	The Complete Football Reference	15032.41	49.99	2012-08-03 00:00:00
2	40232	How to Play Piano (Intermediate)	8565.35	20.50	2012-10-22 00:00:00
3	40481	Simple Auto Repairs	NULL	39.99	2012-06-29 00:00:00
4	40562	Learning to Crochet	10001.14	30.75	2012-07-31 00:00:00
5	40641	Learning Japanese (Beginner)	9875.52	30.00	2012-07-01 00:00:00
6	40712	Recipes From India	11105.15	38.89	2012-08-19 00:00:00

	ordnum	sldate	qty	custnum	partnum	repid
1	00101	2013-11-16 00:00:00	220	20503	40125	N01
2	00102	2013-11-20 00:00:00	100	8802	40232	N02
3	00103	2013-11-20 00:00:00	170	9989	40641	N02
4	00104	2013-12-07 00:00:00	100	9989	40562	N02
5	00105	2013-12-14 00:00:00	150	20493	40481	N01
6	00106	2013-12-16 00:00:00	200	9989	40712	N02

Figure 3-3: An example of data integrity.

There are various data integrity types. They are detailed in the following table.

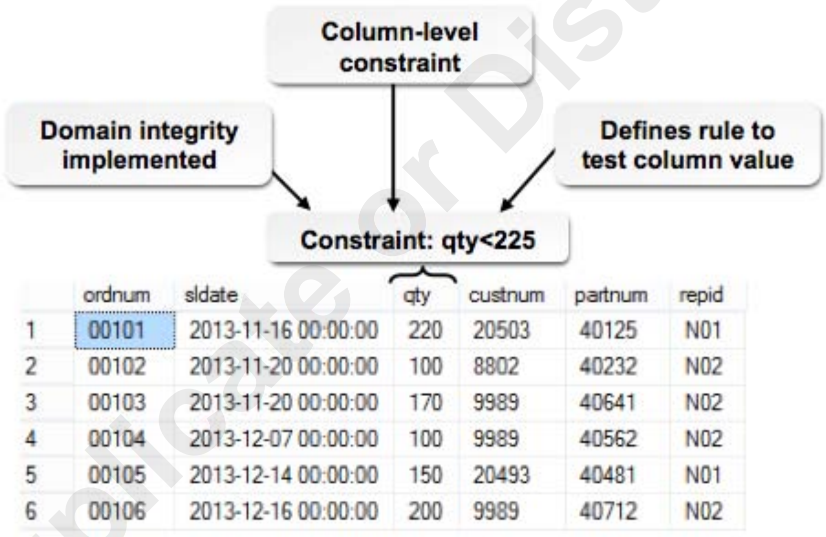
Type of Data Integrity	*Description*
Entity integrity	Indicates the state of the records in a table where each record is unique.
Domain integrity	Indicates that the columns of a table contain valid data.
Referential integrity	Indicates the state of data in linked tables where the columns that link the tables have matching values.

Type of Data Integrity	*Description*
User-defined integrity	Indicates the state of data in a database where the validity of the data is maintained based on business rules defined by a user.

Note: To further explore data integrity, you can access the LearnTO **Design a Database** presentation from the **LearnTO** tile on the LogicalCHOICE Course screen.

Constraints

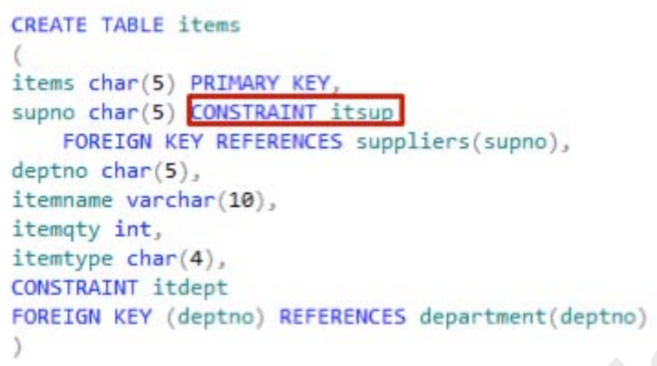
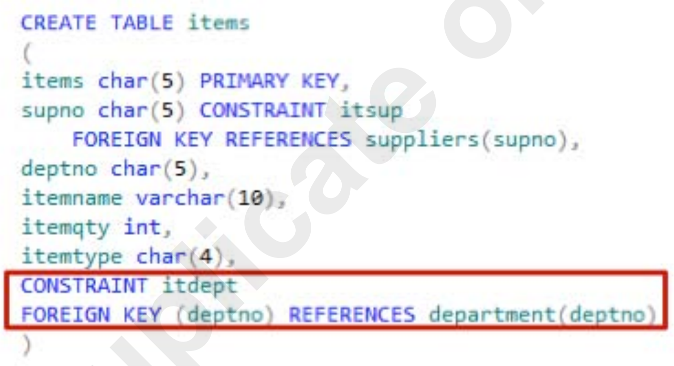
A *constraint* is a validation mechanism you implement in a column of a table to ensure data integrity. Constraints define rules that test the values specified for a column when inserting, modifying, or deleting a record. If the values do not comply with the rules you defined by the constraint, then SQL Server cannot insert, update, or delete the records.


Column-level constraint
Domain integrity implemented
Defines rule to test column value
Constraint: qty<225

	ordnum	sldate	qty	custnum	partnum	repid
1	00101	2013-11-16 00:00:00	220	20503	40125	N01
2	00102	2013-11-20 00:00:00	100	8802	40232	N02
3	00103	2013-11-20 00:00:00	170	9989	40641	N02
4	00104	2013-12-07 00:00:00	100	9989	40562	N02
5	00105	2013-12-14 00:00:00	150	20493	40481	N01
6	00106	2013-12-16 00:00:00	200	9989	40712	N02

Figure 3-4: A constraint example.

You can add constraints at the column level or table level. These constraints are detailed in the following table.

Type of Constraint	*Description*
Column-level constraint	Defined as part of the column definition in a `CREATE TABLE` statement and applied to that column alone. 
Table-level constraint	Defined after the column definition in a `CREATE TABLE` statement. You use this type of constraint when you need to specify multiple columns as part of the constraint. 

```
CREATE TABLE items
(
items char(5) PRIMARY KEY,
supno char(5) CONSTRAINT itsup
    FOREIGN KEY REFERENCES suppliers(supno),
deptno char(5),
itemname varchar(10),
itemqty int,
itemtype char(4),
CONSTRAINT itdept
FOREIGN KEY (deptno) REFERENCES department(deptno)
)
```

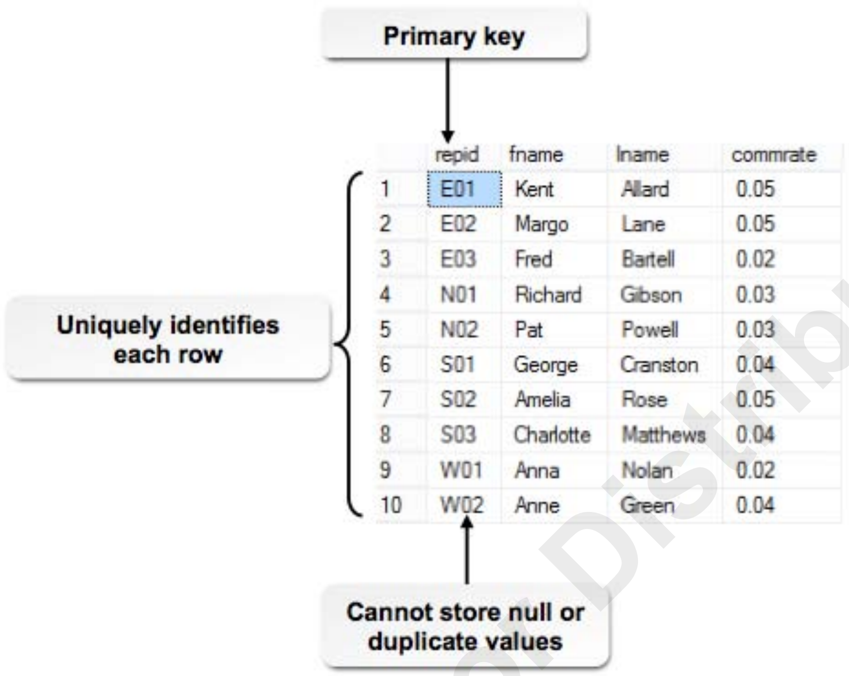
```
CREATE TABLE items
(
items char(5) PRIMARY KEY,
supno char(5) CONSTRAINT itsup
    FOREIGN KEY REFERENCES suppliers(supno),
deptno char(5),
itemname varchar(10),
itemqty int,
itemtype char(4),
CONSTRAINT itdept
FOREIGN KEY (deptno) REFERENCES department(deptno)
)
```

Naming a Constraint

If you don't name a constraint, SQL Server will assign a default name to it. Later, if you want to alter the table constraint, you would need to find the system-generated name before writing the query to alter the table constraint. However, if you assign a name to a constraint, then you can refer to the constraint name whenever you need to modify or delete it. This is much easier than finding out the system-generated constraint names and using them in a query.

Primary Keys

A *primary key* is a column or a combination of columns that stores values to uniquely identify each record in a table. A value entered in the primary key column for any record must be unique from the values in all other records. A table can contain only one primary key, but a primary key can contain multiple columns. Such primary keys are called *composite keys*. A primary key column cannot store null values.


Primary key
Uniquely identifies each row
Cannot store null or duplicate values

	repid	fname	lname	commrate
1	E01	Kent	Allard	0.05
2	E02	Margo	Lane	0.05
3	E03	Fred	Bartell	0.02
4	N01	Richard	Gibson	0.03
5	N02	Pat	Powell	0.03
6	S01	George	Cranston	0.04
7	S02	Amelia	Rose	0.05
8	S03	Charlotte	Matthews	0.04
9	W01	Anna	Nolan	0.02
10	W02	Anne	Green	0.04

Figure 3-5: A primary key example.

PRIMARY KEY Constraints

You use a *PRIMARY KEY constraint* to enforce the uniqueness in a column or a combination of columns in a table. By creating a primary key, you ensure that the values stored in the relevant column(s) are not duplicated across the records of the table. You can define a `PRIMARY KEY` constraint as a column-level constraint or as a table-level constraint when writing the `CREATE TABLE` statement.

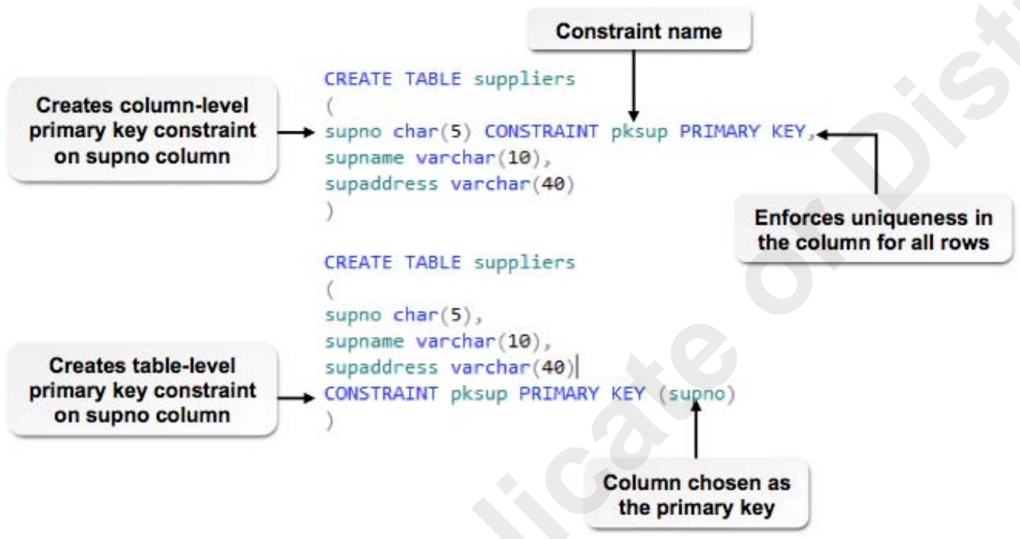
There are two types of `PRIMARY KEY` constraints. They are detailed in the following table.

Constraint Type	*Description*

Column-level `PRIMARY KEY` constraints	You write column-level `PRIMARY KEY` constraints after the definition of the column you've chosen as the primary key. For such constraints, you write the `CONSTRAINT` keyword first, followed by the name of the constraint, and then the `PRIMARY KEY` keywords. The `CONSTRAINT` keyword and the constraint name are optional for column-level `PRIMARY KEY` constraints. If you don't specify a constraint name, SQL Server generates a system-defined name for the constraint.
	`CREATE TABLE tablename`
	`(`
	`columnname1`
	`   datatype (sizevalue) [CONSTRAINT constraintname] PRIMARY KEY,`
	`columnname2 datatype (sizevalue),`
	`columnname3 datatype (sizevalue)`
	`)`

Constraint Type	Description

Table-level `PRIMARY KEY` constraints	For table-level `PRIMARY KEY` constraints, you write the `CONSTRAINT` keyword first, followed by the constraint name, the `PRIMARY KEY` keywords, and then the name of the primary key column. Enclose the name within parentheses.
	`CREATE TABLE tablename`
	`(`
	`columnname1 datatype (sizevalue),`
	`columnname2 datatype (sizevalue),`
	`columnname3 datatype (sizevalue)`
	`[CONSTRAINT constraintname] PRIMARY KEY(columnname1)`
	`)`


Constraint name
Creates column-level primary key constraint on supno column
Enforces uniqueness in the column for all rows
Creates table-level primary key constraint on supno column
Column chosen as the primary key

```
CREATE TABLE suppliers
(
supno char(5) CONSTRAINT pksup PRIMARY KEY,
supname varchar(10),
supaddress varchar(40)
)

CREATE TABLE suppliers
(
supno char(5),
supname varchar(10),
supaddress varchar(40)
CONSTRAINT pksup PRIMARY KEY (supno)
)
```

Figure 3-6: A PRIMARY KEY constraint example.

These two examples demonstrate how to define a primary key at the column and table levels. In the first example, you create a table named Suppliers and use the supplier number (supno) column as the primary key for the table. In the second example, you create the table named Suppliers but define the `PRIMARY KEY` constraint at the table level, after you define the columns in the table. Regardless of which strategy you use to create the `PRIMARY KEY` constraint, the primary key functions the same: it ensures that each row in the table has a unique value for the supno column.

UNIQUE Constraints

You use a *UNIQUE constraint* to enforce uniqueness in a column of a table. Although `PRIMARY KEY` constraints implement the same feature, you can define only one primary key for a table. In contrast, you can create multiple `UNIQUE` constraints for a table. In addition, columns that are constrained by the `UNIQUE` constraint can accept null values. You can define a `UNIQUE` constraint as a table-level constraint or as a column-level constraint. You can also reference a `UNIQUE` constraint by a `FOREIGN KEY` constraint.

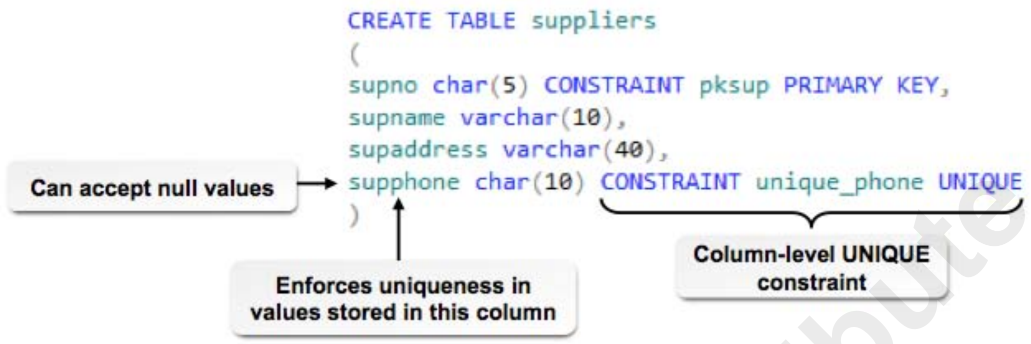

```
CREATE TABLE suppliers
(
supno char(5) CONSTRAINT pksup PRIMARY KEY,
supname varchar(10),
supaddress varchar(40),
supphone char(10) CONSTRAINT unique_phone UNIQUE
)
```
Can accept null values
Enforces uniqueness in values stored in this column
Column-level UNIQUE constraint

Figure 3-7: A UNIQUE constraint example.

There are two types of `UNIQUE` constraints: column-level and table-level. The syntax for defining table-level and column-level `UNIQUE` constraints differs, as shown in the following table.

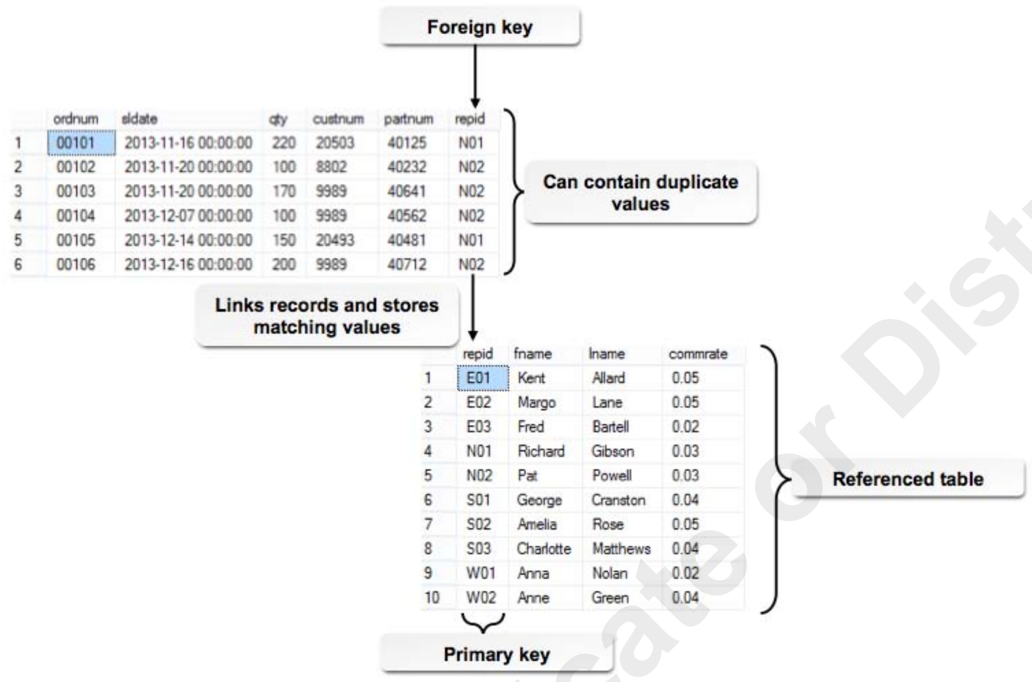
Constraint Type	*Description*

Column-level `UNIQUE` constraint	You write the `CONSTRAINT` keyword first, followed by the constraint name, and then the `UNIQUE` keyword.
	`CREATE TABLE tablename`
	`(`
	`columnname1`
	`   datatype(sizevalue) [CONSTRAINT constraintname] PRIMARY KEY,`
	`columnname2`
	`   datatype(sizevalue),`
	`columnname3`
	`   datatype(sizevalue),`
	`columnname4`
	`   datatype(sizevalue) [CONSTRAINT constraintname] UNIQUE`
	`)`

Table-level `UNIQUE` constraint	You write the `CONSTRAINT` keyword first, followed by the constraint name, the `UNIQUE` keyword, and then write the name of the targeted column enclosed within parentheses.
	`CREATE TABLE tablename`
	`(`
	`columnname1`
	`   datatype(sizevalue) [CONSTRAINT constraintname] PRIMARY KEY,`
	`columnname2`
	`   datatype(sizevalue),`
	`columnname3`
	`   datatype(sizevalue),`
	`columnname4`
	`   datatype(sizevalue)`
	`[CONSTRAINT constraintname] UNIQUE (columnname3)`
	`)`

Foreign Keys

You use a *foreign key* to link a column in one table with a column in another table. The table with the column referenced by the foreign key of another table is called the referenced table. To maintain referential integrity, the values stored in the foreign key column must match the values stored in the primary key or unique key columns of the referenced table. The records in a foreign key can store values that exist in other records of the key.


Foreign key
Can contain duplicate values
Links records and stores matching values
Referenced table
Primary key

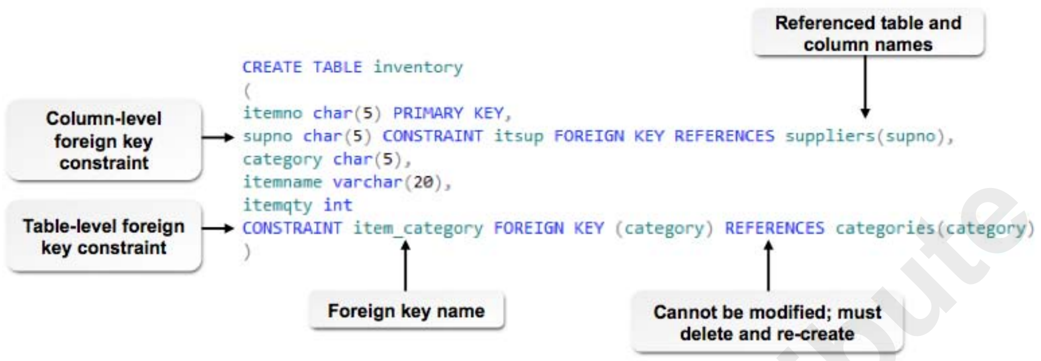
	ordnum	sldate	qty	custnum	partnum	repid
1	00101	2013-11-16 00:00:00	220	20503	40125	N01
2	00102	2013-11-20 00:00:00	100	8802	40232	N02
3	00103	2013-11-20 00:00:00	170	9989	40641	N02
4	00104	2013-12-07 00:00:00	100	9989	40562	N02
5	00105	2013-12-14 00:00:00	150	20493	40481	N01
6	00106	2013-12-16 00:00:00	200	9989	40712	N02

	repid	fname	lname	commrate
1	E01	Kent	Allard	0.05
2	E02	Margo	Lane	0.05
3	E03	Fred	Bartell	0.02
4	N01	Richard	Gibson	0.03
5	N02	Pat	Powell	0.03
6	S01	George	Cranston	0.04
7	S02	Amelia	Rose	0.05
8	S03	Charlotte	Matthews	0.04
9	W01	Anna	Nolan	0.02
10	W02	Anne	Green	0.04

Figure 3-8: A foreign key example.

FOREIGN KEY Constraints

You use a *FOREIGN KEY constraint* to define the foreign key for a table. You create the `FOREIGN KEY` constraint when you create a table with the `CREATE TABLE` statement. You can define the `FOREIGN KEY` constraint as a column-level or as a table-level constraint. You can't modify an existing `FOREIGN KEY` constraint without deleting the constraint and creating it with a new definition.



```
CREATE TABLE inventory
(
itemno char(5) PRIMARY KEY,
supno char(5) CONSTRAINT itsup FOREIGN KEY REFERENCES suppliers(supno),
category char(5),
itemname varchar(20),
itemqty int
CONSTRAINT item_category FOREIGN KEY (category) REFERENCES categories(category)
)
```

Referenced table and column names
Column-level foreign key constraint
Table-level foreign key constraint
Foreign key name
Cannot be modified; must delete and re-create

Figure 3-9: A FOREIGN KEY constraint example.

The syntax for column-level and table-level `FOREIGN KEY` constraints differs, as described in the following table.

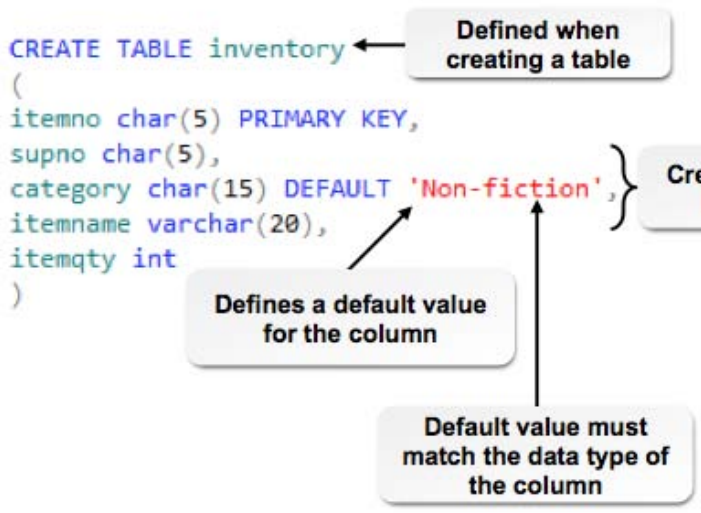
Constraint Type	*Description*

Column-level `FOREIGN KEY` constraint	You specify the `CONSTRAINT` keyword first, followed by the constraint name, the `FOREIGN KEY` keywords, the `REFERENCES` keyword, the name of the referenced table, and the name of the column on the referenced table to which you want to link the foreign key. You must enclose this column name within parentheses.
	`CREATE TABLE tablename`
	`(`
	`columnname1  datatype(sizevalue),`
	`columnname2  datatype(sizevalue),`
	`columnname3  datatype(sizevalue)`
	`   [CONSTRAINT constraintname] FOREIGN KEY REFERENCES`
	`referencingtablename(columnname),`
	`columnname4  datatype(sizevalue),`
	`columnname5  datatype(sizevalue)`
	`)`

Table-level `FOREIGN KEY` constraint	The syntax for a table-level `FOREIGN KEY` constraint is similar to that of the column-level constraint, except that you specify the foreign key name immediately after the `FOREIGN KEY` keywords and enclose the referenced column name within parentheses.
	`CREATE TABLE tablename(`
	`columnname1  datatype(sizevalue),`
	`columnname2  datatype(sizevalue),`
	`columnname3  datatype(sizevalue),`
	`columnname4  datatype(sizevalue),`
	`columnname5  datatype(sizevalue),`
	`CONSTRAINT constraintname FOREIGN KEY (foreignkeycolumnname1,`
	`foreignkeycolumnname2)`
	`   REFERENCES referencingtablename(columnname1, columnname2)`
	`)`

DEFAULT Constraints

A *DEFAULT constraint* is a database constraint you use to specify a default value for a column. You can define the constraint when creating a table with the `CREATE TABLE` statement. The default value specified must correspond to the data type of the constrained column. You can create a `DEFAULT` constraint only as a column-level constraint. You define the constraint by using the `DEFAULT` keyword after the column definition, followed by the constant value you want to use as the default value for the column.



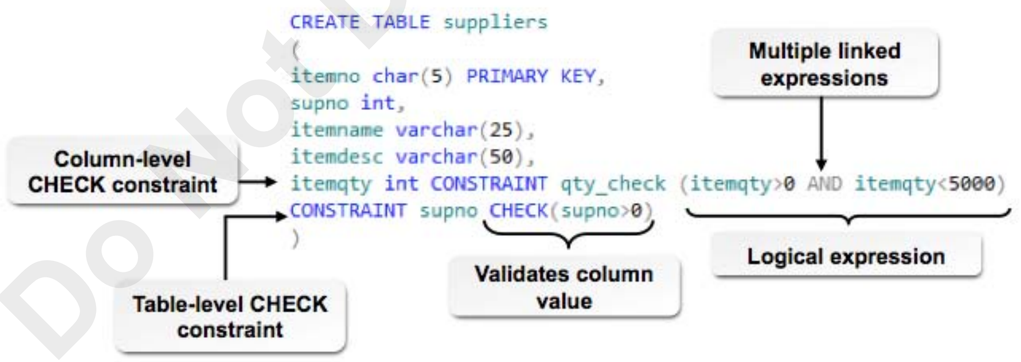
```
CREATE TABLE inventory
(
itemno char(5) PRIMARY KEY,
supno char(5),
category char(15) DEFAULT 'Non-fiction',
itemname varchar(20),
itemqty int
)
```

Defined when creating a table
Created as a column-level constraint
Defines a default value for the column
Default value must match the data type of the column

Figure 3-10: A DEFAULT constraint example.

CHECK Constraints

You use a *CHECK constraint* to validate the values stored in a column based on a condition you specify with the constraint. The condition is a logical operator that returns a value of `TRUE` or `FALSE`, depending on whether the value you insert into the column meets the `CHECK` constraint's requirements. If the column value you insert or modify does not comply with the `CHECK` constraint, SQL Server prevents the insert or update. You can use multiple logical expressions linked with logical operators such as `AND` and `OR` in the `CHECK` constraint. You can also define multiple `CHECK` constraints for a column and SQL Server implements each constraint on the column's value in the order in which you defined them.



```
CREATE TABLE suppliers
(
itemno char(5) PRIMARY KEY,
supno int,
itemname varchar(25),
itemdesc varchar(50),
itemqty int CONSTRAINT qty_check (itemqty>0 AND itemqty<5000)
CONSTRAINT supno CHECK(supno>0)
)
```

Multiple linked expressions
Column-level CHECK constraint
Logical expression
Validates column value
Table-level CHECK constraint

Figure 3-11: A CHECK constraint example.

You can define a `CHECK` constraint as a table-level or column-level constraint. The following table describes the syntax for adding a `CHECK` constraint.

Constraint Type	*Description*

Column-level `CHECK` constraint	Define using the `CONSTRAINT` keyword, followed by the constraint name, the `CHECK` keyword, and the logical expression or expressions you want to use to validate the column values. You can specify only a single `CHECK` constraint at the column level. You can't specify other columns of the table in the logical expression of a column-level `CHECK` constraint.
	`CREATE TABLE items`
	`(`
	`itemno char(5) CONSTRAINT PRIMARY KEY,`
	`supno char(5),`
	`deptno char(5),`
	`itemname varchar(20),`
	`itemqty int CONSTRAINT itchk CHECK(itemqty>0 AND itemqty<5000),`
	`itemtype char(4)`
	`)`

Table-level `CHECK` constraint	Syntax is similar to the column-level `CHECK` constraint. However, at the table level, you can specify multiple columns as part of the logical expression. You can't specify columns from other tables in the logical expression.
	`CREATE TABLE items`
	`(`
	`itemno char(5) CONSTRAINT PRIMARY KEY,`
	`supno char(5),`
	`deptno char(5),`
	`itemname varchar(20),`
	`itemqty int,`
	`itemtype char(4)`
	`CONSTRAINT itchk CHECK(itemqty>0 AND itemqty<5000)`
	`)`

Access the Checklist tile on your LogicalCHOICE course screen for reference information and job aids on How to Create a Table with Constraints.

ACTIVITY 3-2
Planning a Table with Constraints

Scenario

To attract new customers, the Fuller & Ackerman Publishing bookstore plans to host a number of competitions on its website. The website is being redesigned to include crossword puzzles and other games with big prize money to act as incentives. By making a contestant register with the website before allowing entry into the competition, the store plans to garner the contact details of a list of potential customers. The contact details include a participant's name, address, and phone number. This information can be used by the sales team to contact the potential customers later. You plan to store the registration information in a table called Potential_customers1. For information on the tables and column names in the FullerAckerman database, refer to the table structure in Appendix A.

1. **Which is the best method to avoid duplication of customer information?**
   - ○ Provide a message on the website that requests customers who have already registered to avoid registering again.
   - ○ Prevent users with identical names from registering with the website.
   - ○ If users with identical information register, add a prefix to the name of each user to differentiate between the users.
   - ○ Uniquely identify each user by defining the custnum column as the primary key for the table.

2. **When defining the columns of the Potential_customers1 table, which option represents the best method to ensure that a user provides all the required information when registering with the website?**
   - ○ Display a message on the website requesting users to fill all possible fields on the registration form.
   - ○ Specify simple names for the table columns.
   - ○ Reduce the number of columns in the Potential_customers1 table as compared to the Customers table.
   - ○ Use the NOT NULL keywords when defining the column values for the Potential_customers1 table.

3. **True or False? To ensure that potential customers enter valid cell phone numbers into the Potential_customers1 table, you need to add a `CHECK` constraint to the cellno column to check whether the length of the number specified is greater than or equal to 10.**
   - ☐ True
   - ☐ False

ACTIVITY 3-3
Creating Tables with Constraints

Before You Begin

- Ensure SQL Server Management Studio is open.
- On the **Standard** toolbar, select **New Query**.
- On the **SQL Editor** toolbar, in the **Database** drop-down list, ensure that the FullerAckerman database is selected.

Scenario

Now that you've planned the appropriate modifications for the Potential_customers1 table, you need to put them in place. For information on the tables and column names in the FullerAckerman database, refer to the table structure in Appendix A.

1. Create the Potential_customers1 table with column constraints for effective storage of potential customer information.
   a) Type the code to specify the `CREATE TABLE` clause with the name of the table as Potential_customers1 followed by the opening and closing parentheses:

```
CREATE TABLE potential_customers1
(
)
```

   b) Select within the parentheses and enter the code to define a column named custnum as the primary key for the table. Add a second column named referredby that identifies the custnum of the customer who referred the potential customer to the store:

```
CREATE TABLE potential_customers1
(
custnum varchar(5) PRIMARY KEY,
referredby varchar(5),
)
```

   c) Enter the custname and address columns and require that the potential customers enter these values:

```
CREATE TABLE potential_customers1
(
custnum varchar(5) PRIMARY KEY,
referredby varchar(5),
custname varchar(30) NOT NULL,
address varchar(50) NOT NULL,
)
```

   d) Enter the code to create the cellno column and its required constraint:

```
CREATE TABLE potential_customers1
(
custnum varchar(5) PRIMARY KEY,
referredby varchar(5),
custname varchar(30) NOT NULL,
address varchar(50) NOT NULL,
cellno varchar(15) CHECK (LEN(cellno)>=10),
)
```

   e) Enter the code to specify the column definition and the required constraints of the column that store the representative ID of the sales representative assigned to contact the potential customers:

```
CREATE TABLE potential_customers1
(
custnum varchar(5) PRIMARY KEY,
referredby varchar(5),
custname varchar(30) NOT NULL,
address varchar(50) NOT NULL,
cellno varchar(15) CHECK (LEN(cellno)>=10),
repid varchar(3) NOT NULL
)
```

f) Execute the query.

g) Observe that the command executed successfully.

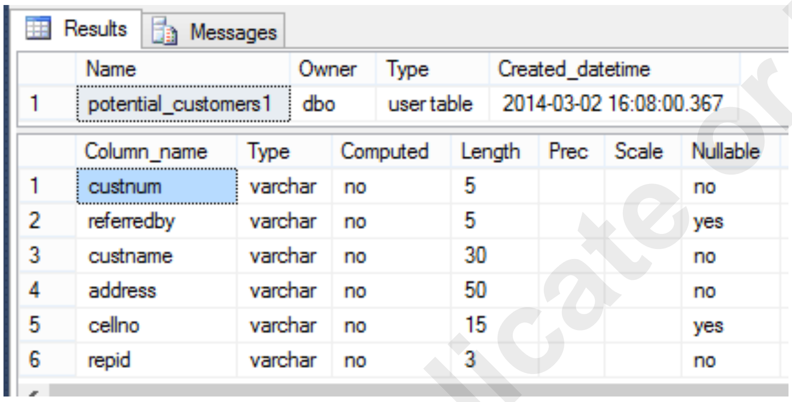
2. View the structure of the Potential_customers1 table.

a) Add the query to display the table structure of the Potential_customers1 table by using the `SP_HELP` stored procedure:

```
sp_help potential_customers1
```

b) Select the **SP_HELP** command then execute the query.

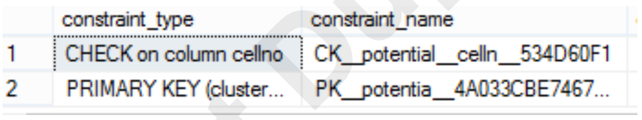
c) In the **Results** pane, scroll down to view the structure of the Potential_customers1 table.



	Name	Owner	Type	Created_datetime
1	potential_customers1	dbo	user table	2014-03-02 16:08:00.367

	Column_name	Type	Computed	Length	Prec	Scale	Nullable
1	custnum	varchar	no	5			no
2	referredby	varchar	no	5			yes
3	custname	varchar	no	30			no
4	address	varchar	no	50			no
5	cellno	varchar	no	15			yes
6	repid	varchar	no	3			no

d) Observe that you defined a primary key on the custnum column.



	constraint_type	constraint_name
1	CHECK on column cellno	CK__potential__celln__534D60F1
2	PRIMARY KEY (cluster...	PK__potentia__4A033CBE7467...

e) Close the **Query Editor** window without saving the query.

TOPIC C

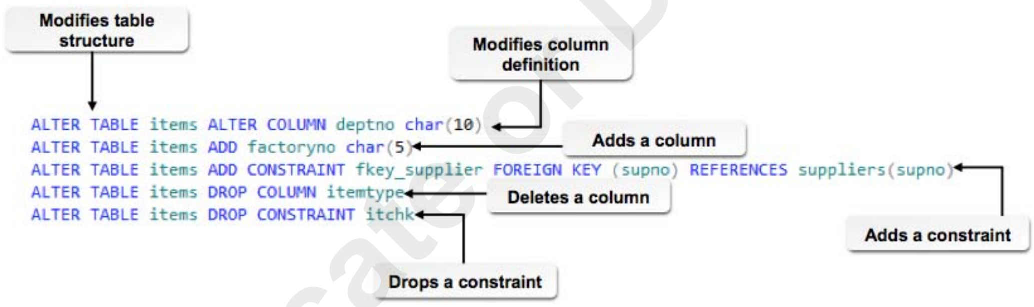
Modify a Table's Structure

You have created various table structures. Based on your requirements, there might be situations when you need to modify the structure of those tables by adding or dropping columns, adding or dropping constraints, or modifying a table's columns. In this topic, you will modify a table's structure to suit your needs.

Modifying a table's structure enables you to tailor a table to meet changing business requirements. As your organization's needs change, your tables must change to meet these needs.

The ALTER TABLE Statement

You modify the structure of a table by using the *ALTER TABLE statement*. By using this statement, you can modify the definition of a column, add columns or constraints to a table, and drop columns or constraints from a table.



```
ALTER TABLE items ALTER COLUMN deptno char(10)
ALTER TABLE items ADD factoryno char(5)
ALTER TABLE items ADD CONSTRAINT fkey_supplier FOREIGN KEY (supno) REFERENCES suppliers(supno)
ALTER TABLE items DROP COLUMN itemtype
ALTER TABLE items DROP CONSTRAINT itchk
```

Figure 3-12: An ALTER TABLE statement example.

Each structure modification task performed by the `ALTER TABLE` statement requires a different syntax. These structure modification tasks are detailed in the following table.

Structure Modification Task	*Syntax and Limitations*
Add columns	To add a column to a table, you specify the `ALTER TABLE` clause first, followed by the table name, the `ADD` keyword, the column name, and the definition of the new column. `ALTER TABLE table_name ADD column_name datatype`
Drop columns	To drop a column from a table, you specify the `ALTER TABLE` statement first, followed by the table name, the `DROP COLUMN` clause, and the name of the column you want to delete. You can't drop columns that are constrained by `PRIMARY KEY`, `FOREIGN KEY`, `UNIQUE`, `CHECK`, and `DEFAULT` constraints. Instead, you must delete the constraint first and then drop the column. `ALTER TABLE table_name DROP COLUMN column_name`

Structure Modification Task	*Syntax and Limitations*
Add constraints	To add a new constraint to a table, you specify the `ALTER TABLE` clause followed by the table name, the `ADD` keyword, and the constraint definition with the `CONSTRAINT` keyword. The syntax of the constraint definition is similar to the syntax used to define constraints in the `CREATE TABLE` statement. The constraint definition when adding a foreign key is similar to the syntax of a table-level `FOREIGN KEY` constraint used with the `CREATE TABLE` statement. `ALTER TABLE table_name ADD CONSTRAINT constraint_name FOREIGN KEY (column_name) REFERENCES table_name(column_name)`
Drop constraints	When dropping a constraint, you use the `ALTER TABLE` statement first, followed by the table name, the `DROP CONSTRAINT` clause, and the name of the constraint you want to drop. `ALTER TABLE table_name DROP CONSTRAINT constraint_name`
Modify the column definition	To modify a column's definition, use the `ALTER TABLE` clause first, followed by the table name, the `ALTER COLUMN` clause, the name of the column you want to modify, and the new column definition. You cannot modify the definition of computed columns, primary keys, foreign keys, columns with `CHECK` or `UNIQUE` constraints, and columns with the `text`, `ntext`, `image`, or `timestamp` data types. If a column has a default constraint, you cannot modify the data type of the column. However, you can modify the value and the size of the value. Modifying the data type of an `nchar` or `nvarchar` column to `char` or `varchar` might change the values stored in the column. `ALTER TABLE table_name ALTER COLUMN deptno char(10)`

Access the Checklist tile on your LogicalCHOICE course screen for reference information and job aids on How to Modify a Table's Structure.

ACTIVITY 3-4
Adding and Dropping Columns

Before You Begin

- Ensure SQL Server Management Studio is open.
- On the **Standard** toolbar, select **New Query**.
- On the **SQL Editor** toolbar, in the **Database** drop-down list, ensure that the FullerAckerman database is selected.

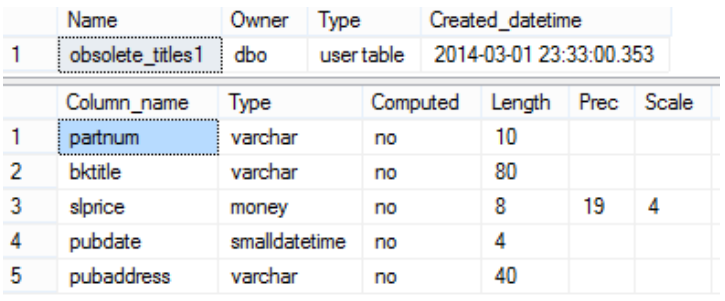
Scenario

After reviewing the monthly sales reports of the Fuller & Ackerman Publishing bookstore, the store manager wants to change some of the information displayed. The manager feels that knowing a book's development cost is unnecessary when generating reports of obsolete titles. Instead, the manager wants you to record the publisher's full address. For information on the tables and column names in the FullerAckerman database, refer to the table structure in Appendix A.

1. Add a column named pubaddress to the Obsolete_titles1 table.
   a) Enter the `ALTER TABLE` statement:
   ```
   ALTER TABLE obsolete_titles1
   ```
   b) Add the column pubaddress with the varchar data type and a maximum size of 40:
   ```
   ALTER TABLE obsolete_titles1 ADD pubaddress varchar(40)
   ```
   c) Execute the query to add the pubaddress column.

2. Drop the devcost column from the Obsolete_titles1 table.
   a) Modify the previous query to drop the devcost column:
   ```
   ALTER TABLE obsolete_titles1 DROP COLUMN devcost
   ```
   b) Execute the statement to drop the devcost column.

3. Check whether the modifications succeeded.
   a) Delete the previous query.
   b) Display the structure of the Obsolete_titles1 table by using the `SP_HELP` stored procedure to verify the changes:
   ```
   sp_help obsolete_titles1
   ```
   c) In the **Results** pane, view the changes in the structure of the Obsolete_titles1 table.



	Name	Owner	Type	Created_datetime
1	obsolete_titles1	dbo	user table	2014-03-01 23:33:00.353

	Column_name	Type	Computed	Length	Prec	Scale
1	partnum	varchar	no	10		
2	bktitle	varchar	no	80		
3	slprice	money	no	8	19	4
4	pubdate	smalldatetime	no	4		
5	pubaddress	varchar	no	40		

   d) Close the **Query Editor** window without saving the query.

ACTIVITY 3-5

Adding and Dropping Constraints

Data Files

Setup_Constraints1.sql

Setup_Constraints2.sql

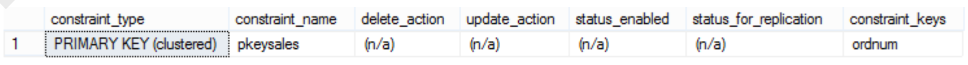
Before You Begin

- In SQL Server, using the **File** menu, navigate to the **C:\094006Data\Manipulating the Table Structure** folder and open the **Setup_Constraints1.sql** file.
- On the toolbar, in the **Database** drop-down list, ensure that the FullerAckerman database is selected.
- Execute the statements and close the file.
- In SQL Server, using the **File** menu, navigate to the **C:\094006Data\Manipulating the Table Structure** folder and open the **Setup_Constraints2.sql** file.
- On the toolbar, in the **Database** drop-down list, ensure that the FullerAckerman database is selected.
- Execute the statements and close the file.
- Open the **New Query** window.
- On the toolbar, in the **Database** drop-down list, ensure that the FullerAckerman database is selected.

Scenario

When reviewing the monthly sales reports of the Fuller & Ackerman Publishing bookstore, the store manager identifies certain discrepancies in the report. Some records in the Sales table are duplicated. Sales records show sales of books that do not exist in the Titles table. Some records of book sales show zero copies sold to a customer. You deleted these problem records as a temporary measure. However, the store manager asks you to ensure that such discrepancies do not occur in the future. For information on the tables and column names in the FullerAckerman database, refer to the table structure in Appendix A.

1. Add a primary key to the Sales table and check whether you added the key successfully.
   a) Enter the query to add a primary key called pkeysales to the Sales table:
   ```
   ALTER TABLE sales ADD CONSTRAINT pkeysales
   ```
   b) Specify the ordnum column with the `PRIMARY KEY` keyword:
   ```
   ALTER TABLE sales ADD CONSTRAINT pkeysales PRIMARY KEY (ordnum)
   ```
   c) Execute the statements to add the primary key.
   d) To verify your change, display the structure of the Sales table using the `SP_HELP` stored procedure:
   ```
   sp_help sales
   ```
   e) The last section in the **Results** pane displays the constraint information.



	constraint_type	constraint_name	delete_action	update_action	status_enabled	status_for_replication	constraint_keys
1	PRIMARY KEY (clustered)	pkeysales	(n/a)	(n/a)	(n/a)	(n/a)	ordnum

2. Link the Sales table to the Titles table and check whether the tables have been linked.
   a) Delete the previous query.

b) Enter the code to create a `FOREIGN KEY` constraint named `fkeytitles` to the Sales table:

```
ALTER TABLE sales ADD CONSTRAINT fkeytitles
```

c) Type the code for the `FOREIGN KEY` and `REFERENCES` keywords to link the partnum column of the Sales table to the equivalent column in the Titles table:

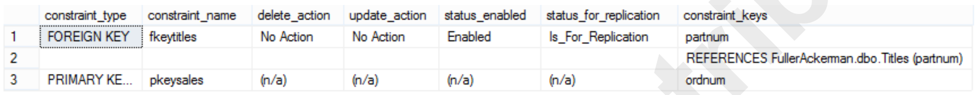
```
ALTER TABLE sales ADD CONSTRAINT fkeytitles
FOREIGN KEY (partnum) REFERENCES titles(partnum)
```

d) Execute the `ALTER TABLE` statement. You should see that the command completed successfully.

e) Display the structure of the Sales table:

```
sp_help sales
```

f) In the **Results** pane, verify that you added the `fkeytitles FOREIGN KEY` constraint to the Sales table.

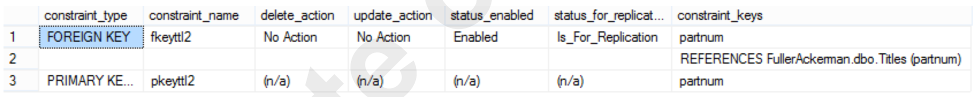


	constraint_type	constraint_name	delete_action	update_action	status_enabled	status_for_replication	constraint_keys
1	FOREIGN KEY	fkeytitles	No Action	No Action	Enabled	Is_For_Replication	partnum
2							REFERENCES FullerAckerman.dbo.Titles (partnum)
3	PRIMARY KE...	pkeysales	(n/a)	(n/a)	(n/a)	(n/a)	ordnum

3. Remove the link between the Titles and Titles2 tables by dropping the constraint named `fkeyttl2`. Verify that you successfully removed the `FOREIGN KEY` constraint from the Titles2 table.

a) Delete the previous query.

b) Display the structure of the Titles2 table and observe the constraints. The Titles2 table has a `FOREIGN KEY` constraint named `fkeyttl2` and a `PRIMARY KEY` constraint named `pkeyttl2`.



	constraint_type	constraint_name	delete_action	update_action	status_enabled	status_for_replicat...	constraint_keys
1	FOREIGN KEY	fkeyttl2	No Action	No Action	Enabled	Is_For_Replication	partnum
2							REFERENCES FullerAckerman.dbo.Titles (partnum)
3	PRIMARY KE...	pkeyttl2	(n/a)	(n/a)	(n/a)	(n/a)	partnum

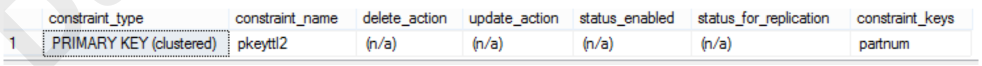
c) Edit the query to define the `ALTER TABLE` statement with the `DROP CONSTRAINT` clause to drop the constraint named `fkeyttl2` from the Titles2 table:

```
ALTER TABLE titles2 DROP CONSTRAINT fkeyttl2
```

d) Execute the query.

e) Execute a query to display the structure of the Titles2 table.

f) In the **Results** pane, verify that you successfully deleted the `fkeyttl2 FOREIGN KEY` constraint from the Titles2 table.



	constraint_type	constraint_name	delete_action	update_action	status_enabled	status_for_replication	constraint_keys
1	PRIMARY KEY (clustered)	pkeyttl2	(n/a)	(n/a)	(n/a)	(n/a)	partnum

4. **True or False? To ensure that the quantity value of books stored in a record of the Sales table is always above zero, add a `CHECK` constraint on the qty column of the Sales table to check whether the value stored in the column is greater than zero.**

☐ True

☐ False

5. Add a `CHECK` constraint to the qty column of the Sales table. Verify that you successfully added the constraint.

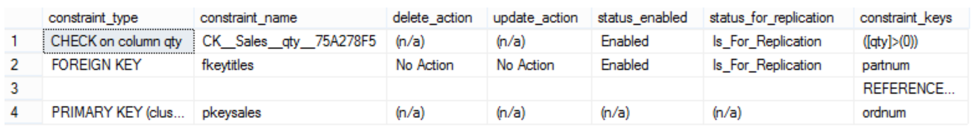
a) Delete the previous query.

b) Enter the following query to add a `CHECK` constraint:

```
ALTER TABLE sales ADD CHECK (qty>0)
```

c) Execute the statement.

d) Display the structure of the Sales table. You see the `CHECK` constraint listed in the constraints section of the results.



	constraint_type	constraint_name	delete_action	update_action	status_enabled	status_for_replication	constraint_keys
1	CHECK on column qty	CK__Sales__qty__75A278F5	(n/a)	(n/a)	Enabled	Is_For_Replication	([qty]>(0))
2	FOREIGN KEY	fkeytitles	No Action	No Action	Enabled	Is_For_Replication	partnum
3							REFERENCE...
4	PRIMARY KEY (clus...	pkeysales	(n/a)	(n/a)	(n/a)	(n/a)	ordnum

e) Close the **Query Editor** window without saving the query.

ACTIVITY 3-6
Modifying the Column Definition

Before You Begin

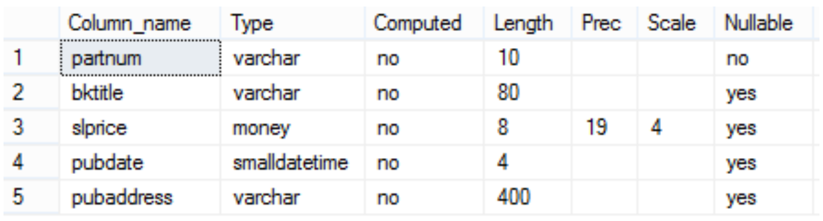
- Ensure SQL Server Management Studio is open.
- On the **Standard** toolbar, select **New Query**.
- On the **SQL Editor** toolbar, in the **Database** drop-down list, ensure that the FullerAckerman database is selected.

Scenario

To track and procure obsolete book titles in the least possible time, the store manager of the Fuller & Ackerman Publishing bookstore requested that you create a table that contains book information and the address of the publisher. You created the Obsolete_titles1 table and added the pubaddress column to store publisher addresses. As you store records in the table, you realize that the size of the pubaddress column is not large enough to store all publishers' addresses. You decide to increase the pubaddress column size to fulfill the store manager's requirements. For information on the tables and column names in the FullerAckerman database, refer to the table structure in Appendix A.

1. Increase the size of the pubaddress column from 40 to 400 to accommodate the full addresses of publishers.
   a) Enter the code for the `ALTER TABLE` statement with Obsolete_titles1 as the table name:
   ```
   ALTER TABLE obsolete_titles1
   ```
   b) Add the `ALTER COLUMN` clause of the `ALTER TABLE` statement with the new column size for the pubaddress column:
   ```
   ALTER TABLE obsolete_titles1 ALTER COLUMN pubaddress varchar(400)
   ```
   c) Execute the statement.

2. Verify that you successfully modified the size of the pubaddress column.
   a) Delete the previous query.
   b) Display the structure of the Obsolete_titles1 table:
   ```
   sp_help obsolete_titles1
   ```
   c) In the **Results** pane, observe the change in the column definition of the pubaddress column.



	Column_name	Type	Computed	Length	Prec	Scale	Nullable
1	partnum	varchar	no	10			no
2	bktitle	varchar	no	80			yes
3	slprice	money	no	8	19	4	yes
4	pubdate	smalldatetime	no	4			yes
5	pubaddress	varchar	no	400			yes

   d) Close the **Query Editor** window without saving the query.

TOPIC D

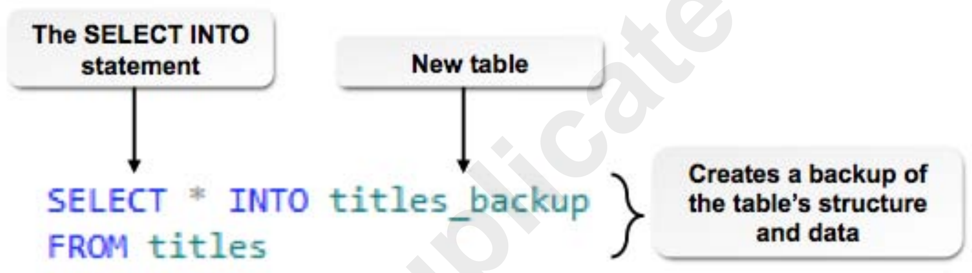
Back Up Tables

As your database grows, you will find that there is an increasing amount of information in the tables you created. To ensure that you don't lose any data, backing up the data in these tables becomes essential. In this topic, you will back up the data in tables.

Some tables in a database contain a large amount of information that you cannot easily replace. For example, if there is a problem with the server on which the data is stored, you could lose the data in these tables and you'll find it takes a great deal of effort and cost to replace the lost data. To reduce the amount of time required to back up this information, instead of backing up an entire database, you could back up only those tables with the most important information.

The SELECT INTO Statement

One strategy you can use to back up specific tables in a database is to use the *SELECT INTO statement*. The `SELECT INTO` statement enables you to create a backup of a table's structure and data in a new table. You can also copy the contents from multiple tables or views into a new table. However, you cannot use the `SELECT INTO` statement to back up a table that uses the `COMPUTE` clause because SQL Server will not store the results of the `COMPUTE` clause in the database. Therefore, when you create a new table using the `SELECT INTO` statement, any calculations produced during the execution of the `COMPUTE` clause will not appear in the new table.


The SELECT INTO statement
New table
Creates a backup of the table's structure and data

```
SELECT * INTO titles_backup
FROM titles
```

Figure 3-13: A SELECT INTO statement example.

Note: To further explore backing up, you can access the LearnTO **Back Up and Restore Databases** presentation from the **LearnTO** tile on the LogicalCHOICE Course screen.

Access the Checklist tile on your LogicalCHOICE course screen for reference information and job aids on How to Back Up Tables.

ACTIVITY 3-7

Backing Up a Table

Before You Begin

- Ensure SQL Server Management Studio is open.
- On the **Standard** toolbar, select **New Query**.
- On the **SQL Editor** toolbar, in the **Database** drop-down list, ensure that the FullerAckerman database is selected.

Scenario

You have information about some important books in the Titles table. You would prefer to back up the information for future reference. For information on the tables and column names in the FullerAckerman database, refer to the table structure in Appendix A.

1. Back up the Titles table into a new table.
   a) Enter the following query:
   ```
   SELECT * INTO titles_backup
   FROM titles
   ```
   b) Execute the query.

2. View the structure and data of the Titles_backup table.
   a) Delete the previous query.
   b) View the structure of the Titles_backup table:
   ```
   sp_help titles_backup
   ```
   c) View the contents of the Titles_backup table. You should see 92 rows:
   ```
   SELECT *
   FROM titles_backup
   ```
   d) Close the **Query Editor** window without saving the query.

TOPIC E

Delete Tables

Throughout this course so far, you have created, modified, and backed up tables. With time, you might find that some tables become obsolete and you no longer need them. Rather than having them take up space in your database, you can update your database by deleting such tables. In this topic, you will delete tables.

Some tables in a database become obsolete because of changing business conditions. For example, if an organization dealing with manufacturing diversifies into IT services and closes its parent manufacturing division, the suppliers table that stores information on past vendors and the constraints added to the table become obsolete, so you can delete them. You use SQL statements to reorganize your database by deleting unused tables and freeing up database space.

The DROP TABLE Statement

The *DROP TABLE statement* is a SQL statement that deletes a table from a database. The name of the table to be dropped must be specified after the `DROP TABLE` clause. In addition to the records in the table, the table's definition, which includes the column definitions and any constraints provided in the definition, is also deleted when you execute this statement. However, if a table targeted for deletion is being referenced by the foreign key of another table, the referencing `FOREIGN KEY` constraint or the referencing table must be dropped before the target table can be deleted.

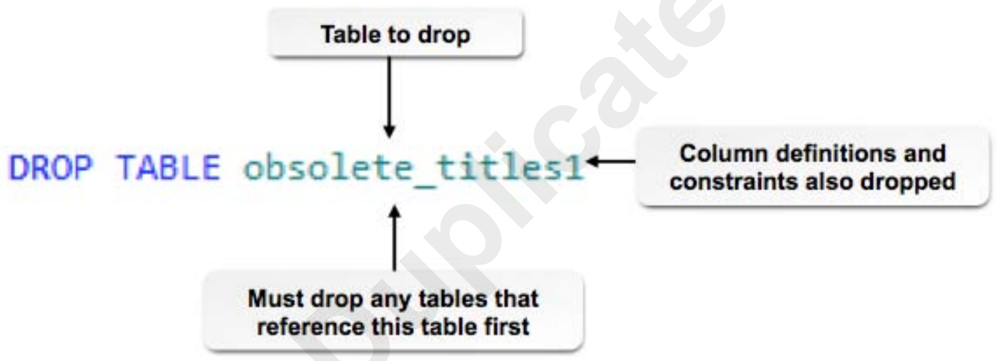


Table to drop
DROP TABLE obsolete_titles1
Column definitions and constraints also dropped
Must drop any tables that reference this table first

Figure 3-14: A DROP TABLE statement example.

Access the Checklist tile on your LogicalCHOICE course screen for reference information and job aids on How to Delete Tables.

ACTIVITY 3-8

Deleting Tables

Before You Begin

- Ensure SQL Server Management Studio is open.
- On the **Standard** toolbar, select **New Query**.
- On the **SQL Editor** toolbar, in the **Database** drop-down list, ensure that the FullerAckerman database is selected.

Scenario

In a database maintained by the Fuller & Ackerman Publishing bookstore, you created the Titles2 table to back up the information in the Titles table. With the implementation of a portable backup drive, the database administrator (DBA) for the store feels that the table is an unnecessary drain on the resources of the SQL server. The DBA wants you to delete the Titles2 table. For information on the tables and column names in the FullerAckerman database, refer to the table structure in Appendix A.

1. Delete the Titles2 table from the FullerAckerman database.
   a) Enter the following query:

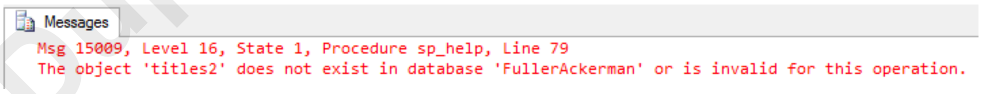
      ```
      DROP TABLE titles2
      ```

   b) Execute the `DROP TABLE` statement to delete the Titles2 table.

2. Verify that you deleted the Titles2 table.
   a) Delete the previous query.
   b) Attempt to display the structure of the Titles2 table:

      ```
      sp_help titles2
      ```

   c) Observe the error message.



```
Msg 15009, Level 16, State 1, Procedure sp_help, Line 79
The object 'titles2' does not exist in database 'FullerAckerman' or is invalid for this operation.
```

   d) Close the **Query Editor** window without saving the query.

Summary

As your organization's needs change, you must be prepared to modify the structure of its SQL tables or even delete unnecessary tables. Knowing how to insert new tables or new columns, modify existing columns, and delete columns or tables enables you to quickly respond to your organization's changing needs.

How should you map business requirements and table designs before creating tables?

What basic constraints should you implement when creating a table?

Note: Check your LogicalCHOICE Course screen for opportunities to interact with your classmates, peers, and the larger LogicalCHOICE online community about the topics covered in this course or other topics you are interested in. From the Course screen you can also access available resources for a more continuous learning experience.

4 Working with Views

Lesson Time: 1 hour, 10 minutes

Lesson Objectives

In this lesson, you will work with views. You will:

- Create a view.
- Manipulate the data in views.
- Create aliases.
- Modify and drop views.

Introduction

So far in the course, you've learned how to design advanced queries, modify the contents of tables, and create and modify the structure of tables. Another SQL feature you can use, called views, enables you to display only the necessary data instead of all the data in a table. You can use views to control exactly what data users can see from a table. In this lesson, you will create views, use them to manipulate data, modify views, and delete views.

Imagine you want a user to update your organization's employee addresses. However, you don't want this user to see the employees' salary information. To handle this situation, you can create a view on the employees table that shows only employee names and addresses, but not their salaries. By creating a view, you control exactly what information the user can see.

TOPIC A

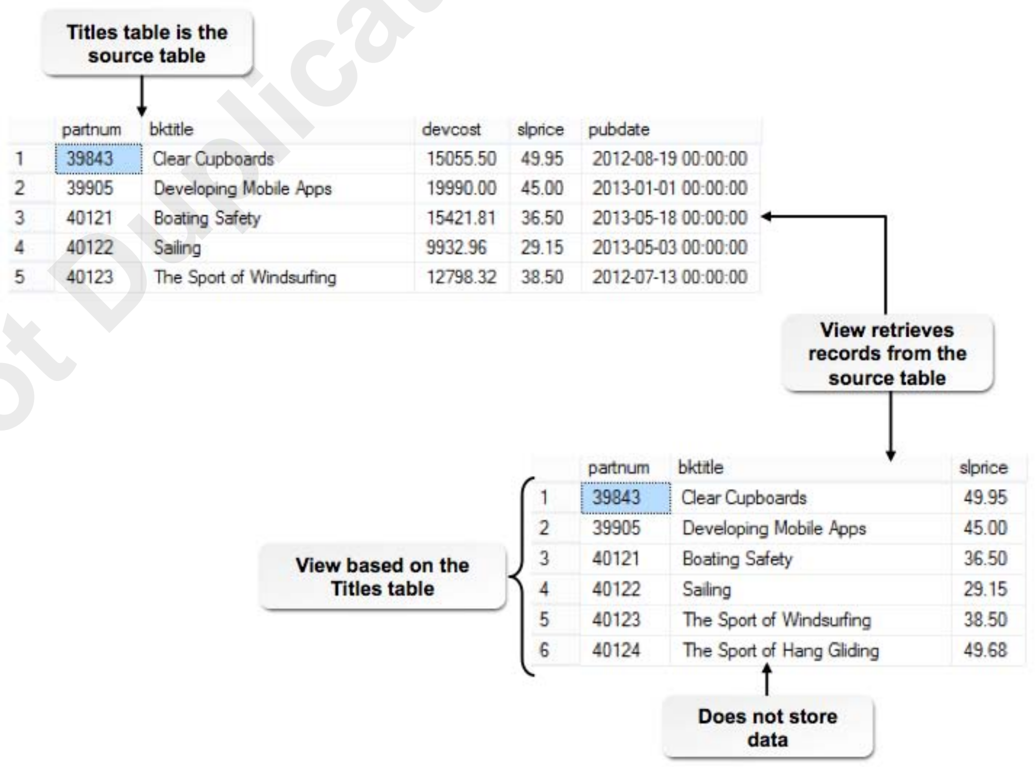
Create a View

An organizational database table might contain a large number of records. Reviewing a table with a large number of rows makes it difficult to pinpoint the information you need. You can simplify this task by creating views to display specific columns, records, or both, instead of displaying the entire table. In this topic, you will create views based on a single column and multiple columns.

In some cases, coding simple queries is not enough to find the records from a table. A table could contain a large number of columns or records. You might also need to frequently view records from multiple tables. In such scenarios, you can use views to display the data you want from a table or multiple tables. With views, you can generate an output that contains only certain columns from a single table or multiple tables. Views also allow you to display specific records from a single table or multiple tables.

Views

You use a *view* to define a virtual table that retrieves and displays records from one or more tables. Unlike database tables, Microsoft® SQL Server® 2012 doesn't store data in views; instead, you use them simply to display the records stored in a table. The table from which you retrieve records is called the source table. The view acts as an interface between a user and the source table. You can make changes to the data within a view and SQL Server will change the relevant data in the source table. You use views to retrieve specific rows and columns from a source table so that you can easily identify the data you need. You can also use views to display records from multiple tables as a single virtual table.


Titles table is the source table
View retrieves records from the source table
View based on the Titles table
Does not store data

	partnum	bktitle	devcost	slprice	pubdate
1	39843	Clear Cupboards	15055.50	49.95	2012-08-19 00:00:00
2	39905	Developing Mobile Apps	19990.00	45.00	2013-01-01 00:00:00
3	40121	Boating Safety	15421.81	36.50	2013-05-18 00:00:00
4	40122	Sailing	9932.96	29.15	2013-05-03 00:00:00
5	40123	The Sport of Windsurfing	12798.32	38.50	2012-07-13 00:00:00

	partnum	bktitle	slprice
1	39843	Clear Cupboards	49.95
2	39905	Developing Mobile Apps	45.00
3	40121	Boating Safety	36.50
4	40122	Sailing	29.15
5	40123	The Sport of Windsurfing	38.50
6	40124	The Sport of Hang Gliding	49.68

Figure 4-1: A sample view.

In the example, you see a view that displays only the partnum, bktitle, and slprice columns from the Titles table. The Titles table is thus the source table.

After you create a view, you can use it in `SELECT` statements just as you would a regular table. For example, to list all rows in the view named booklist, you would use this syntax:

```
SELECT *
FROM booklist
```

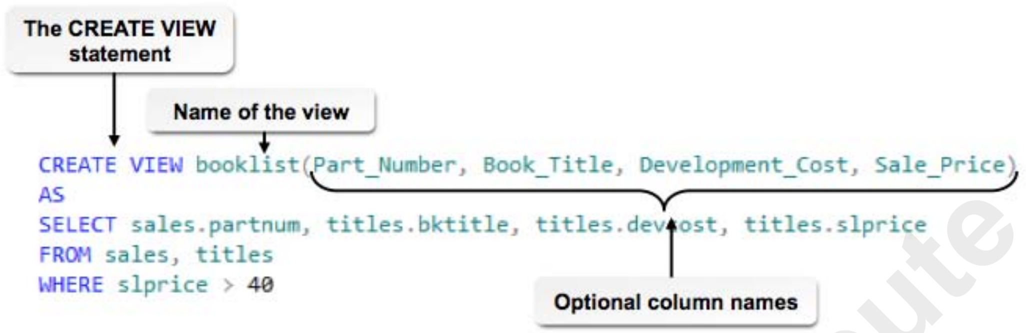
Note: To further explore the views included with SQL Server 2012, you can access the LearnTO **Use the SQL Information Schema Views** presentation from the **LearnTO** tile on the LogicalCHOICE Course screen.

The CREATE VIEW Statement

In SQL Server, you use the *CREATE VIEW statement* to create views. You assign a name to the view immediately after the `CREATE VIEW` statement. You then add a `SELECT` statement following the view name and the `AS` keyword. This `SELECT` statement retrieves and displays records in the form of a virtual table. You can create a view based on a single table or multiple tables.

There are different types of views you can choose from. These view types are detailed in the following table.

Type of View	*Description*
Simple views	Creates a view based on a single table. Using this view, you can insert and update records in a source table or delete records from the source table. When inserting, updating, or deleting records through a simple view, you might make changes that violate the condition you defined in the `SELECT` statement when you created the view. For example, you might create a view that displays all customers who live in Rhode Island. If you update a row and change the state a customer lives in to Pennsylvania, SQL Server won't display that customer in the view. If you want to prevent users from changing data that then prevents SQL Server from displaying a row in the view, use the `WITH CHECK OPTION` clause after the `SELECT` statement when you create the view. The `SELECT` statement of simple views cannot contain aggregate functions.
Complex views	Creates a view based on multiple source tables. You cannot insert, update, or delete records in this type of view. You can create complex views based on tables in other databases or even other database servers. You can include aggregate functions in the `SELECT` statement of complex views. If you use an aggregate function, you must provide a name for the column in the view. You define the name of the column as part of the `CREATE VIEW` statement. You can also define specific column names if you want to display names that are different from the column names in the source table.



```
CREATE VIEW booklist(Part_Number, Book_Title, Development_Cost, Sale_Price)
AS
SELECT sales.partnum, titles.bktitle, titles.devcost, titles.slprice
FROM sales, titles
WHERE slprice > 40
```

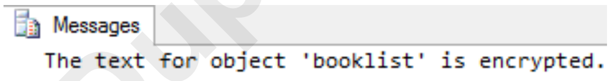
Figure 4-2: A CREATE VIEW statement example.

In the example, the `CREATE VIEW` statement creates a view named booklist that contains the partnum column from the Sales table and the bktitle, devcost, and slprice columns from the Titles table. In addition, the `CREATE VIEW` statement defines column names that SQL Server displays when you display the view instead of the column names from the tables. You can display the structure of a view by executing the `sp_helptext` *`view_name`* stored procedure.

You need to follow some rules when you define the `SELECT` statement for a view:

- You can't use an `ORDER BY` clause in the `SELECT` statement for a view unless you include the `TOP` keyword followed by the number of rows you want to see.
- You can't include a `COMPUTE` or `COMPUTE BY` clause in the view's `SELECT` statement.
- SQL Server does not permit you to use aggregate functions such as `AVG` in the `SELECT` statement for a simple view.
- You also can't use the `INTO` keyword after the `SELECT` statement for a view. (Remember, you use `SELECT INTO` to create a copy of a table and its data.

SQL Server includes support for encrypting the definition of a view. You should encrypt a view's definition if you do not want users to be able to see how you defined the view. You might encrypt a view when you don't want users to be able to identify the structure of the source table(s) that comprise the view. When you encrypt a view, SQL Server cannot display its structure when you use the `sp_helptext` *`view_name`* stored procedure. SQL Server displays the following message instead:



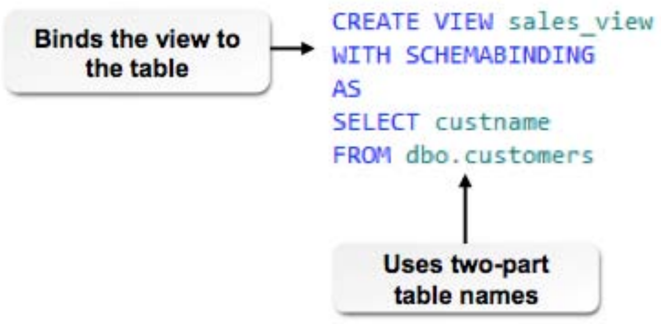
```
The text for object 'booklist' is encrypted.
```

The syntax for encrypting a view is:

```
CREATE VIEW viewname
WITH ENCRYPTION
AS SELECT column1, column2, column3
FROM table1, table2
```

Schema Binding

When you create a view, one of the options you can include is *SCHEMABINDING*. You use `SCHEMABINDING` to protect the source table on which you based the view. `SCHEMABINDING` prevents users from modifying the source table on which you based a view because changes to the source table's structure can affect the view's definition. When you use the `SCHEMABINDING` option, SQL Server does not permit you to make changes to the source table if they would affect a view. Instead, if you need to change the structure of the source table, you must first drop or change the view so that it is no longer using the `SCHEMABINDING` option.



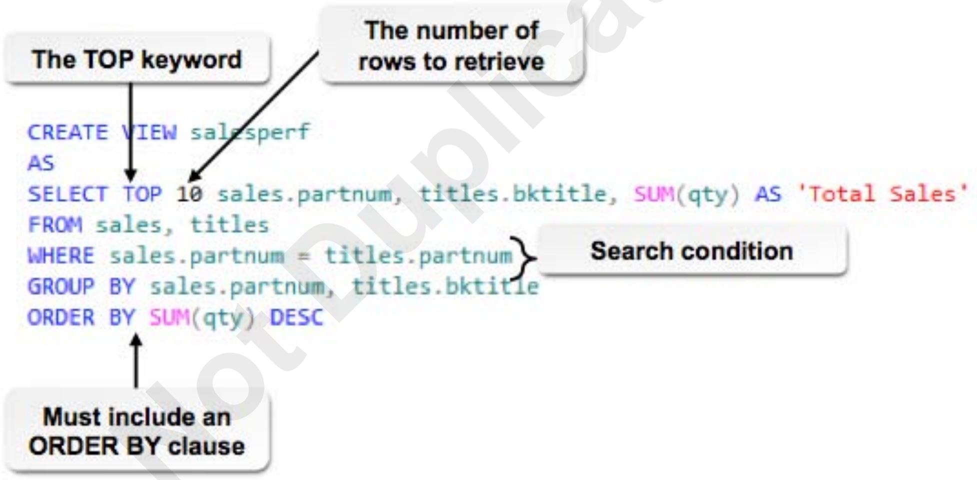
```
CREATE VIEW sales_view
WITH SCHEMABINDING
AS
SELECT custname
FROM dbo.customers
```

Figure 4-3: Sample code for a SCHEMABINDING statement.

In this example, the `CREATE VIEW` statement defines a view named sales_view with the `SCHEMABINDING` option. The view is based on the Customers table and displays only the custname column. When you use the `SCHEMABINDING` option, you must specify the two-part table name, which consists of the table's owner followed by the table name.

The TOP Keyword

You can use the *TOP* keyword followed by a number or a percentage in a `SELECT` statement to limit the number of records SQL Server displays. You can retrieve a specific number of records by indicating a constant value. To retrieve a specific percentage of records, you must specify a constant value followed by the `PERCENT` keyword. If the `SELECT` statement in a `CREATE VIEW` statement contains an `ORDER BY` clause, you must specify the `TOP` keyword with the `SELECT` statement.



```
CREATE VIEW salesperf
AS
SELECT TOP 10 sales.partnum, titles.bktitle, SUM(qty) AS 'Total Sales'
FROM sales, titles
WHERE sales.partnum = titles.partnum
GROUP BY sales.partnum, titles.bktitle
ORDER BY SUM(qty) DESC
```

The TOP keyword
The number of rows to retrieve
Search condition
Must include an ORDER BY clause

Figure 4-4: Sample code for the TOP keyword.

This example creates a view named salesperf that displays the top 10 sales (based on the quantity sold) from the Sales table. Notice that you must use the `ORDER BY` clause in descending order to list the records with the greatest quantity sold.

The following code block shows the use of the `TOP` keyword along with the `CREATE VIEW` clause:

```
CREATE VIEW viewname [(columnname1, columnname2, columnname3)]
AS
SELECT TOP value [PERCENT] SUM(columnname3) AS 'Colname',columnname2,
columnname3
```

```
FROM tablename1 [, tablename2]
WHERE columnname4=columnname5
GROUP BY columnname1, columnname2
ORDER BY SUM(columnname3) DESC
```

Access the Checklist tile on your LogicalCHOICE course screen for reference information and job aids on How to Create a View.

ACTIVITY 4-1
Creating Views

Before You Begin

- Ensure SQL Server Management Studio is open.
- On the **Standard** toolbar, select **New Query**.
- On the **SQL Editor** toolbar, in the **Database** drop-down list, ensure that the FullerAckerman database is selected.

Scenario

It has been a few months since you joined the Fuller & Ackerman Publishing bookstore as the SQL programmer to maintain their database. At frequent intervals, you must generate a list of the top 20 books sold in all the branches of the chain. The list usually displays the part number and title, and the total quantity sold for each book. In the past, you've used a complex query to generate this list each time. However, many managers at the head office require this list and want an easy method of viewing the latest list of books.

In addition, the store managers have requested some changes to the EasySearch software they use in the stores. Store employees use this software to help customers identify the books they need. Store managers noticed that customers usually search for books priced between $20 and $40, so they want you to add a medium price range option to the EasySearch software. This option must run a query that generates the book title, part number, and sale price of books that fall into this category.

To resolve these issues, you need to work with the Sales and Titles tables. For information on the tables and column names in the FullerAckerman database, refer to the table structure in Appendix A.

1. **True or False? To implement the medium price option in the EasySearch software, you need to create a view that retrieves the part number, book title, and sale price of books in the Titles table that are priced between $20 and $40.**

   ☐ True

   ☐ False

2. Create a view named mediumprice that retrieves the required columns from the Titles table. Verify that you successfully created the view.

   a) Enter the statement to create the view:

   ```
   CREATE VIEW mediumprice
   ```

   b) Enter the `SELECT` statement to retrieve the relevant columns from the Titles table:

   ```
   CREATE VIEW mediumprice
   AS
   SELECT partnum, bktitle, slprice
   FROM titles
   ```

   c) Add the search condition to the query so that the view retrieves those books that fall in the $20 to $40 price range:

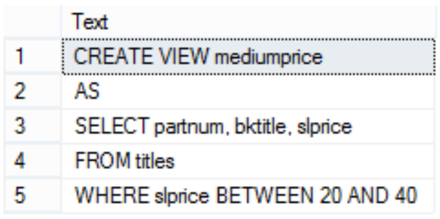
   ```
   CREATE VIEW mediumprice
   AS
   SELECT partnum, bktitle, slprice
   FROM titles
   WHERE slprice BETWEEN 20 AND 40
   ```

   d) Execute the statements to create the view.

e) Write and execute a `SELECT` statement to retrieve the view. You should see that there are 71 books in the medium price range:

```
SELECT *
FROM mediumprice
```

f) Write and execute the `sp_helptext mediumprice` stored procedure to review the view definition.



	Text
1	CREATE VIEW mediumprice
2	AS
3	SELECT partnum, bktitle, slprice
4	FROM titles
5	WHERE slprice BETWEEN 20 AND 40

3. **To generate a list of the top 20 books based on sales volume, you need to create a view that displays the part number, book title, and total sales quantities of the top 20 books based on the sales volume. Which is the correct logic you should use to group and sort records to display the required output?**

○ Group and sort the records based on the total sales quantity of each book. Display the records in descending order.

○ Group records based on the part number of the Sales table and the book title of the Titles table. Sort the output based on the total quantity value sold for each book and display the records in descending order.

○ Group the records based on the book title. Sort the records based on the part number of each book and display the records in descending order.

○ Group the records based on the part number of each book. Sort the records based on the book title and display the records in descending order.

4. Create a view named salesperf to generate the top 20 list of books sold.

a) Delete the previous queries.

b) Write the statements to create a view named salesperf that retrieves the part number, book title, and total sales quantity of the top 20 books sold:

```
CREATE VIEW salesperf
AS
SELECT TOP 20 sales.partnum, titles.bktitle, SUM(sales.qty)
FROM sales, titles
WHERE sales.partnum = titles.partnum
```

c) Use the `GROUP BY` statement to group the information based on partnum from the Sales table and bktitle from the Titles table. Use an `ORDER BY SUM` clause to display the quantity in descending order:

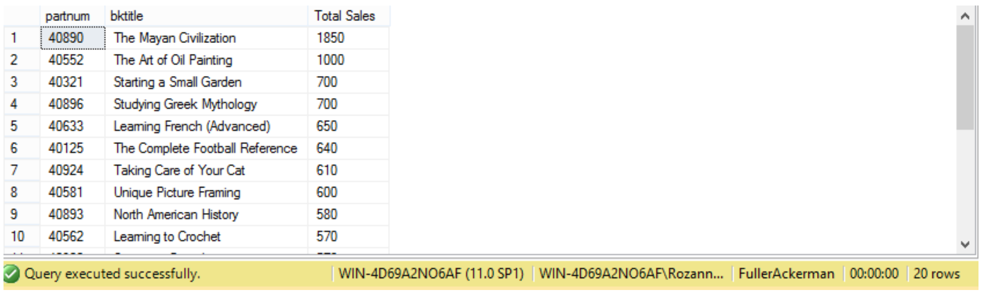
```
CREATE VIEW salesperf
AS
SELECT TOP 20 sales.partnum, titles.bktitle, SUM(sales.qty) AS 'Total
Sales'
FROM sales, titles
WHERE sales.partnum = titles.partnum
GROUP BY sales.partnum, titles.bktitle
ORDER BY SUM(sales.qty) DESC
```

d) Execute the code block to create the `salesperf` view.

e) Enter the query to display the view:

```
SELECT *
FROM salesperf
```

f) You should see that the view displays 20 rows, which are the top 20 books sold.



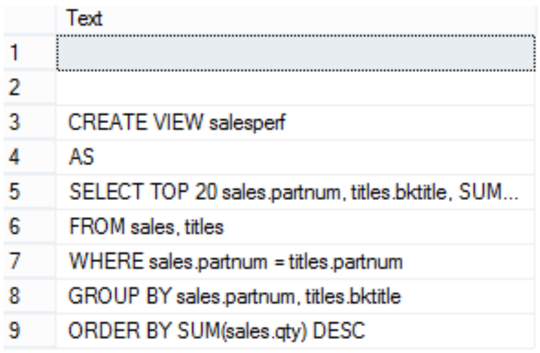
	partnum	bktitle	Total Sales
1	40890	The Mayan Civilization	1850
2	40552	The Art of Oil Painting	1000
3	40321	Starting a Small Garden	700
4	40896	Studying Greek Mythology	700
5	40633	Learning French (Advanced)	650
6	40125	The Complete Football Reference	640
7	40924	Taking Care of Your Cat	610
8	40581	Unique Picture Framing	600
9	40893	North American History	580
10	40562	Learning to Crochet	570

Query executed successfully. | WIN-4D69A2NO6AF (11.0 SP1) | WIN-4D69A2NO6AF\Rozann... | FullerAckerman | 00:00:00 | 20 rows

g) Write and execute a query statement to display the definition of the `salesperf` view:

```
sp_helptext salesperf
```

h) The definition of the `salesperf` view displays.



	Text
1	
2	
3	CREATE VIEW salesperf
4	AS
5	SELECT TOP 20 sales.partnum, titles.bktitle, SUM...
6	FROM sales, titles
7	WHERE sales.partnum = titles.partnum
8	GROUP BY sales.partnum, titles.bktitle
9	ORDER BY SUM(sales.qty) DESC

i) Close the **Query Editor** window without saving the query.

ACTIVITY 4-2

Creating Views with Schema Binding

Before You Begin

- Ensure SQL Server Management Studio is open.
- On the **Standard** toolbar, select **New Query**.
- On the **SQL Editor** toolbar, in the **Database** drop-down list, ensure that the FullerAckerman database is selected.

Scenario

You are a trainee programmer at the Fuller & Ackerman Publishing bookstore. Your human resources department head asked you for the list of customer names that a salesperson could use. You created a view to display the list of customer names that the salesperson could refer to. But, the database administrator, who was unaware of this, altered the schema of the source table. As a result, the view that you created became unusable at a crucial time. To prevent such problems from occurring in the future, you decide to create the view so that SQL Server prevents modifications to the source table. For information on the tables and column names in the FullerAckerman database, refer to the table structure in Appendix A.

1. Create a view named sales_view with schema binding on the Customers table.
   a) Enter the `CREATE VIEW` statement:
   ```
   CREATE VIEW sales_view
   ```
   b) Add the `WITH SCHEMABINDING` option to schema bind the view:
   ```
   CREATE VIEW sales_view
   WITH SCHEMABINDING
   ```
   c) Add the code to display the custname column from the Customers table. Type **`CustName`** to specify the column that should be displayed in the view:
   ```
   CREATE VIEW sales_view
   WITH SCHEMABINDING
   AS
   SELECT CustName
   FROM dbo.customers
   ```
   d) Execute the `CREATE VIEW` statement.
   e) Verify that the view functions properly by executing this query:
   ```
   SELECT *
   FROM sales_view
   ```
2. Attempt to alter the structure of the Customers table.
   a) Delete the previous query.
   b) Enter the following query to attempt to modify the structure of the Customers table:
   ```
   ALTER TABLE customers
   ALTER COLUMN CustName varchar (30) NOT NULL
   ```
   c) Execute the `ALTER TABLE` query.

d) Observe the error message you see when you attempt to modify the structure of a table that is schema bound to a view.

```
Msg 5074, Level 16, State 1, Line 1
The object 'sales_view' is dependent on column 'CustName'.
Msg 4922, Level 16, State 9, Line 1
ALTER TABLE ALTER COLUMN CustName failed because one or more objects access this column.
```

e) Close the **Query Editor** window without saving the query.

TOPIC B

Manipulate Data in Views

After you create a view, you might need to manipulate the data displayed in the view. SQL Server enables you to insert, modify, and delete rows while using a view. In this topic, you will learn how to manipulate the data in views.

You observed data in a table using views. But what if you need to change the data you see in a view? You can use a view to insert or update records in tables, or delete the records from the table on which you based a view.

Insert, Modify, and Delete Data Through Views

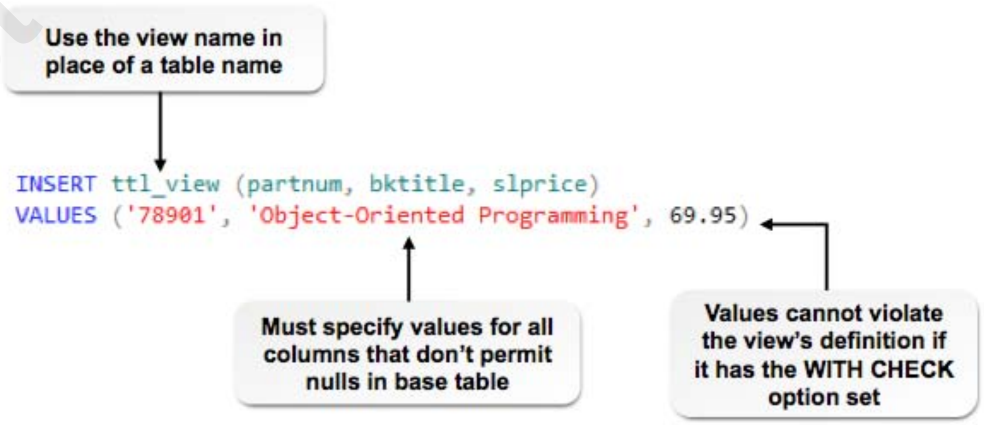
SQL Server includes support for inserting, modifying, and deleting data through views. You can use the `INSERT`, `UPDATE`, and `DELETE` statements to work with data displayed in a view just as you can use those same statements to work with a table's data. The syntax is identical with the exception that you use a view name in place of the table name.

There are some limitations you can encounter when attempting to manipulate data through a view. These limitations include:

- If you created the view using the `WITH CHECK OPTION`, you can't make changes to the view's data that will violate the view's definition. For example, if you create a view that displays all books with a price between $20 and $30 and you use the `WITH CHECK OPTION`, you cannot use the `UPDATE` statement to change a book's price to $35 because this change violates the definition of the view.
- You can insert, modify, or delete data through a view only if the view is based on a single table. SQL Server does not permit you to manipulate data through a view based on multiple tables.
- If the base table for the view includes columns that don't permit null values and the view does not include those columns, you will receive an error when you attempt to add a row because you have not provided values for those columns.

You insert a row in a view by using the `INSERT` statement. Although you insert the row using the view, SQL Server actually stores the inserted row in the table on which the view is based. Here is the syntax for the `INSERT` statement using a view:

```
INSERT view_name(column1, column2, column3)
VALUES (value for column1, value for column2, value for column3)
```


Use the view name in place of a table name
Must specify values for all columns that don't permit nulls in base table
Values cannot violate the view's definition if it has the WITH CHECK option set

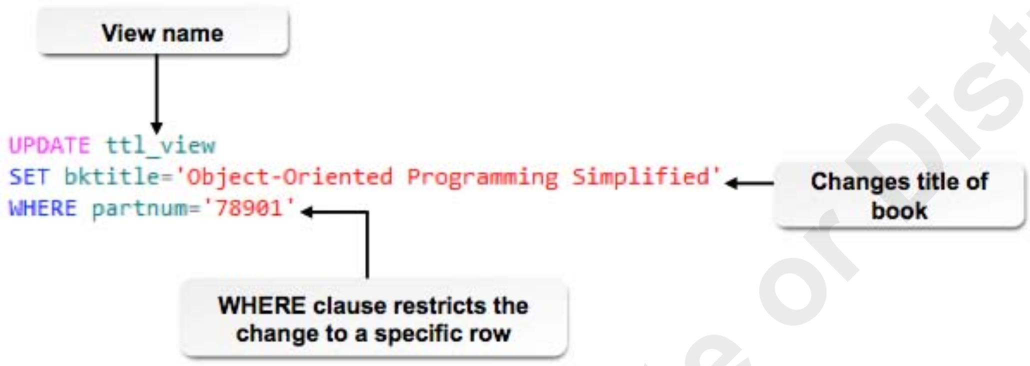
```
INSERT ttl_view (partnum, bktitle, slprice)
VALUES ('78901', 'Object-Oriented Programming', 69.95)
```

Figure 4-5: An example of inserting data through a view.

In the example, the INSERT statement inserts a new row using the ttl_view view. The new row has a part number of 78901, a title of "Object-Oriented Programming," and a sale price of $69.95. Although the INSERT statement uses the view name ttl_view, SQL Server actually inserts the row into the table on which the view is based: the Titles table.

You can modify a row's contents through a view as long as your changes don't violate the view's definition. Just as with the INSERT statement, SQL Server updates the table on which the view is based. Keep in mind that you must include a WHERE clause to select the row(s) you want to modify or the UPDATE statement will modify all rows in the table. Use this syntax to update a row through a view:

```
UPDATE view_name
SET column1=value
WHERE search_condition
```



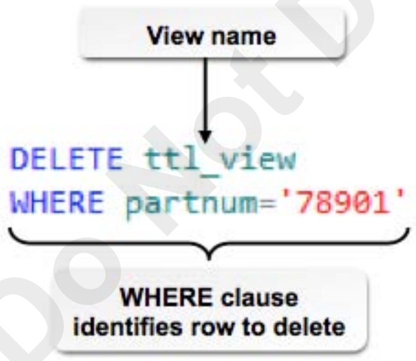
```
UPDATE ttl_view
SET bktitle='Object-Oriented Programming Simplified'
WHERE partnum='78901'
```

Figure 4-6: An example of modifying data through a view.

In this example, the UPDATE statement changes the title of the book with the part number of 78901 to "Object-Oriented Programming Simplified."

Finally, you can delete rows through a view by using the DELETE statement. Be sure to use a WHERE clause to identify the row(s) you want to delete. Use this syntax to delete a row through a view:

```
DELETE view_name
WHERE search_condition
```



```
DELETE ttl_view
WHERE partnum='78901'
```

Figure 4-7: An example of deleting data through a view.

In this example, the `DELETE` statement deletes the row with the part number of 78901 using the view.

Access the Checklist tile on your LogicalCHOICE course screen for reference information and job aids on How to Manipulate Data in Views.

ACTIVITY 4-3

Manipulating Data in Views

Before You Begin

- Ensure SQL Server Management Studio is open.
- On the **Standard** toolbar, select **New Query**.
- On the **SQL Editor** toolbar, in the **Database** drop-down list, ensure that the FullerAckerman database is selected.

Scenario

The store manager of the Fuller & Ackerman Publishing bookstore has been getting reports from salespeople that a number of new titles are not listed in the database. The salespeople have compiled a list of titles missing from the database, as shown in the following list. In addition, a salesperson let you know that the sale price of the book with part number 40256 should be $30, not $35. The book with part number 40234 does not exist in the store, but is listed as available. Because customers view the part number, book title, and sale price when they use the EasySearch computer, the store manager wants you to implement these changes right away.

Books containing erroneous information include the following.

Part Number	*Book Title*	*Sale Price*
40256	How to Play Violin (Intermediate)	35
40257	How to Play Violin (Advanced)	39

1. True or False? You could use the mediumprice view to insert, update, and delete records from the Titles table.

   ☐ True

   ☐ False

2. Insert the new books into the Titles table by using the `mediumprice` view. Verify that you successfully added the rows.

   a) Write the statements to insert the new titles “How to Play Violin (Intermediate)” and “How to Play Violin (Advanced)” using the `mediumprice` view:

   ```
   INSERT mediumprice (partnum, bktitle, slprice)
   VALUES ('40256', 'How to Play Violin (Intermediate)', 35)

   INSERT mediumprice (partnum, bktitle, slprice)
   VALUES ('40257', 'How to Play Violin (Advanced)', 39)
   ```

   b) Execute the statements to insert the records.

   c) Delete the previous queries.

   d) Write a statement to query the `mediumprice` view to verify that you inserted the two new books:

   ```
   SELECT *
   FROM mediumprice
   WHERE partnum IN ('40256', '40257')
   ```

   e) Execute the `SELECT` statement. SQL Server displays the two new rows.

3. Change the sales price of the book with part number 40256 by using the `mediumprice` view. Verify that your change succeeded.
   a) Delete the previous queries.
   b) Update the sale price of the book with part number 40256 using the `mediumprice` view. Execute the query:
   ```
   UPDATE mediumprice
   SET slprice=30
   WHERE partnum='40256'
   ```
   c) Write and execute a statement to retrieve the information for the book with partnum 40256 to make sure you modified its sale price:
   ```
   SELECT *
   FROM mediumprice
   WHERE partnum='40256'
   ```
4. Delete the record for the book with part number 40234 using the `mediumprice` view. Verify this change.
   a) Delete the previous queries.
   b) Write and execute a statement to delete the record with part number 40234:
   ```
   DELETE mediumprice
   WHERE partnum='40234'
   ```
   c) To verify whether the deletion succeeded, attempt to retrieve the deleted record:
   ```
   SELECT *
   FROM mediumprice
   WHERE partnum='40234'
   ```
   d) Close the **Query Editor** window without saving the query.
5. **True or False? The mediumprice view displays records retrieved from the Titles table. Although you deleted the record with part number 40234 from the mediumprice view, the record still exists in the Titles table.**

   ☐ True

   ☐ False

TOPIC C

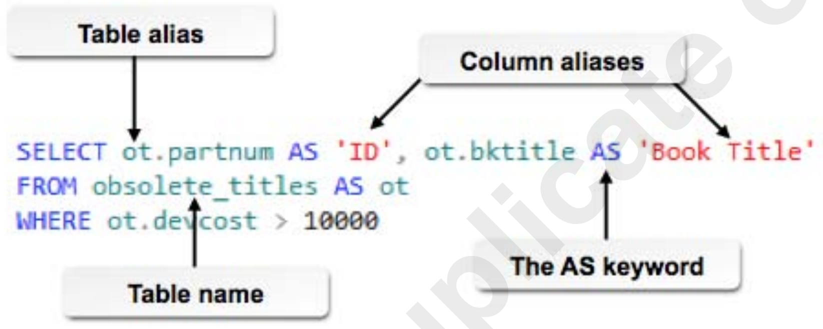
Create Aliases

By default, SQL displays the names of columns as the headings for columns within views. As an alternative, you can define different names for the columns you include in the view. You do so by assigning aliases to the columns. In addition, you can assign aliases to table names. You use aliases for table names to reduce the amount of typing you must do for complex queries. In this topic, you will create aliases.

You are creating a complex view that refers to multiple tables and columns. You want an easy way to identify the contents of the columns and also to simplify the syntax of the query. Alias names enable you to create alternate names for columns and tables for easy manipulation of data.

Aliases

An *alias* is a reference you use for a table or a column. An alias helps you avoid confusion, especially when you write a complex query that pulls data from two or more tables and both tables have a column with the same name. For example, in the FullerAckerman database, both the Titles and the Sales tables contain a column named partnum. You define aliases by specifying a name after the `AS` keyword. If you want to use a space in the alias, you must enclose the name within single quotes.



```
SELECT ot.partnum AS 'ID', ot.bktitle AS 'Book Title'
FROM obsolete_titles AS ot
WHERE ot.devcost > 10000
```

Figure 4-8: Sample code for creating aliases.

In this example, the query assigns the aliases of 'ID' to the partnum column and 'Book Title' to the bktitle column. In addition, the query assigns the table alias of 'ot' to the Obsolete_titles table.

Access the Checklist tile on your LogicalCHOICE course screen for reference information and job aids on How to Create Aliases.

ACTIVITY 4-4
Creating Aliases

Before You Begin

- Ensure SQL Server Management Studio is open.
- On the **Standard** toolbar, select **New Query**.
- On the **SQL Editor** toolbar, in the **Database** drop-down list, ensure that the FullerAckerman database is selected.

Scenario

You want to create a list of book titles and their part numbers from the Obsolete_titles table. You don't want to have to type the Obsolete_titles table name in full when you define the columns you want to include in the `SELECT` statement. In addition, you would like the results set to display more detailed names for the columns instead of the abbreviated column names defined for the table. For information on the tables and column names in the FullerAckerman database, refer to the table structure in Appendix A.

Write a query to retrieve the partnum and bktitle columns from the Obsolete_titles table. Use aliases for both the column names and the table name.

a) Enter the following query:

```
SELECT ot.partnum AS 'Part Number', ot.bktitle AS 'Book Title'
FROM obsolete_titles AS ot
ORDER BY ot.bktitle
```

b) Execute the query. You see the seven books contained in the Obsolete_titles table.

c) Close the **Query Editor** window without saving the query.

TOPIC D

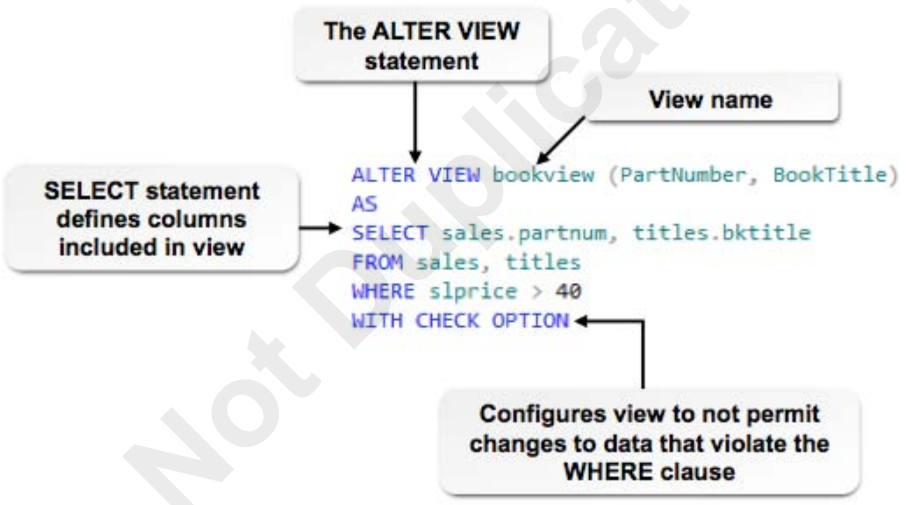
Modify and Delete Views

You have created views and manipulated the data within the view. It's possible that you might need to change the structure of the view because of changing business needs. SQL Server enables you to modify the structure of a view by using the `ALTER VIEW` statement. You can also delete a view by using the `DROP VIEW` statement. In this topic, you will modify the view structure by adding and deleting columns, and deleting views.

The columns you defined when creating a view might become useless if you do not need to view them frequently. In addition, you may need to add columns to a view if you want to view the data in these columns frequently. Some views might become obsolete and you should delete them. You can use SQL to restructure views or drop them altogether.

The ALTER VIEW Statement

You use the *ALTER VIEW statement* to modify the definition of a view. By defining a new `SELECT` statement with the `ALTER VIEW` statement, you modify the structure of the view. You specify the name of the view in the `ALTER VIEW` clause. Then, add a new `SELECT` statement followed by the `AS` keyword. You can optionally include the `WITH CHECK OPTION` clause in the `ALTER VIEW` statement to have SQL Server verify that the data displayed by the view complies with the `WHERE` clause (search condition) of your `SELECT` statement.



```
ALTER VIEW bookview (PartNumber, BookTitle)
AS
SELECT sales.partnum, titles.bktitle
FROM sales, titles
WHERE slprice > 40
WITH CHECK OPTION
```

The ALTER VIEW statement
View name
SELECT statement defines columns included in view
Configures view to not permit changes to data that violate the WHERE clause

Figure 4-9: Sample code for an ALTER VIEW statement.

In the example, the `ALTER VIEW` statement modifies the `bookview` view and changes its `SELECT` statement to include just the partnum and bktitle columns. In addition, the `WHERE` clause for the view configures it to display only those books with prices higher than $40.

The DROP VIEW Statement

When you want to delete a view, you use the *DROP VIEW statement*. If you need to delete a table, SQL Server requires that you delete all views based on that table first. To delete a view, use the `DROP VIEW` statement followed by the name of the view.

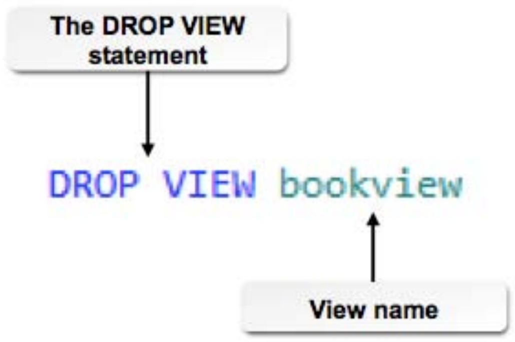


Figure 4-10: Sample code for a DROP VIEW statement.

Access the Checklist tile on your LogicalCHOICE course screen for reference information and job aids on How to Modify and Delete Views.

ACTIVITY 4-5
Modifying and Deleting Views

Data File

ModifyDropStart.sql

Before You Begin

- From the **C:\094006Data\Working with Views** folder, open **ModifyDropStart.sql** in SQL Server, and execute it to create a view named slsperf, then close the file.
- Ensure SQL Server Management Studio is open.
- On the **Standard** toolbar, select **New Query**.
- On the **SQL Editor** toolbar, in the **Database** drop-down list, ensure that the FullerAckerman database is selected.

Scenario

To monitor the performance of salespeople in the Fuller & Ackerman Publishing bookstore, you developed a view named slsperf for the store manager that retrieves the representative ID, name, and total quantity sold by the top 10 salespeople. The store manager just announced that each of the top three salespeople will be rewarded with a bonus based on their sales volume at the end of each month. The manager wants you to change the view named slsperf to list only the top three salespeople. In addition, the manager has decided to remove the medium price range view from the customer EasySearch software. For information on the tables and column names in the FullerAckerman database, refer to the table structure in Appendix A.

---

1. **True or False? To retrieve information on the top three salespeople, you need to change the number specified after the `TOP` keyword in the `SELECT` statement of the `slsperf` view from 10 to 3.**

   ☐ True

   ☐ False

2. **Modify the `slsperf` view to retrieve the top three salespeople instead of the top 10. Verify that the view functions correctly.**

   a) Enter the query to modify the `slsperf` view:

   ```
   ALTER VIEW slsperf
   AS
   SELECT
   ```

   b) Add the commands to retrieve the representative ID, first name, last name, and total sales quantity of the top three sales representatives from the `slsperf` view:

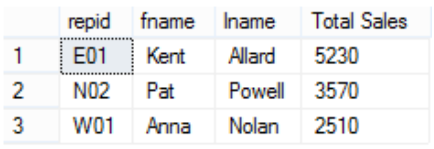
   ```
   ALTER VIEW slsperf
   AS
   SELECT TOP 3 sales.repid, slspers.fname, slspers.lname, sum(qty) AS
   'Total Sales'
   FROM sales, slspers
   WHERE sales.repid=slspers.repid
   GROUP BY sales.repid, slspers.fname, slspers.lname
   ORDER BY sum(qty) DESC
   ```

   c) Execute the `ALTER VIEW` statement.

   d) Verify that the view functions properly:

```
SELECT *
FROM slsperf
```

e) You should see that the view returns only the top three salespeople.

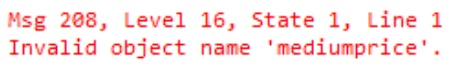


	repid	fname	lname	Total Sales
1	E01	Kent	Allard	5230
2	N02	Pat	Powell	3570
3	W01	Anna	Nolan	2510

3. Drop the `mediumprice` view and verify that you deleted it successfully.
   a) Delete the previous queries.
   b) Execute the query `DROP VIEW mediumprice` to drop the `mediumprice` view.
   c) Attempt to retrieve all records in the `mediumprice` view to verify that the view no longer exists:

```
SELECT *
FROM mediumprice
```

   d) SQL Server displays an error message.



```
Msg 208, Level 16, State 1, Line 1
Invalid object name 'mediumprice'.
```

   e) Close the **Query Editor** window without saving the query.

Summary

Views enable you to display only certain columns, rows, or both. Thus, you can use views to manage exactly what information users can and cannot see. You will find that views provide a convenient way to hide confidential information from users or mask the complexity of a query.

How does your organization use views?

Does your organization use views to generate reports? Why?

Note: Check your LogicalCHOICE Course screen for opportunities to interact with your classmates, peers, and the larger LogicalCHOICE online community about the topics covered in this course or other topics you are interested in. From the Course screen you can also access available resources for a more continuous learning experience.

5 Indexing Data

Lesson Time: 30 minutes

Lesson Objectives

In this lesson, you will create indexes on table columns and drop inefficient indexes. You will:

- Create indexes on table columns.
- Drop an index.

Introduction

Another type of database object you can create is an index. SQL Server uses indexes to optimize query performance. Just as searching through the range of names at the top of a phone book page is faster than searching through each person listed on every page, so is searching through an index for a database table faster than scanning the entire table.

How SQL Server retrieves the data that satisfies a query can have a huge impact on your database server's performance, especially if you have very large tables. One of the strategies you can use to improve the efficiency of SQL Server is to create indexes to support your most frequently executed queries. Indexes enable you to optimize performance by reducing the amount of data SQL Server must search through to satisfy a query.

TOPIC A

Create Indexes

The first step in optimizing the performance of your queries is to create indexes on your tables. In this topic, you'll learn how to create indexes by using the `CREATE INDEX` statement.

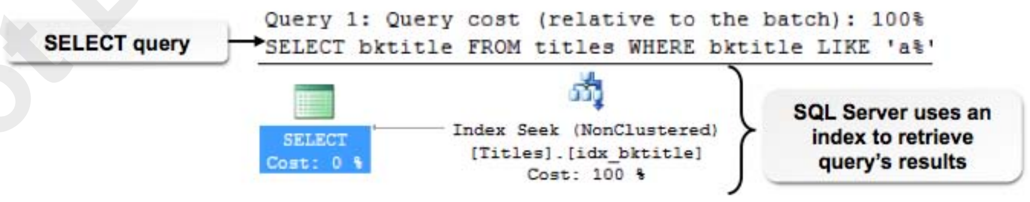
If you need to find information on a specific topic in a textbook, you could find it using one of three methods. First, you could read through every page of the book until you find the required information. Second, you could browse through the table of contents. Or third, you could use the index to access the exact page that contains the information you need. A SQL index has the same function for a table as a book index: to speed up the retrieval of information.

Indexes

When you execute a query, Microsoft® SQL Server® can retrieve the query's information either by performing a table scan or by accessing an *index*. A table scan means that SQL Server goes through a table one row at a time to find the rows for the results set of a query. As you can imagine, table scans are inefficient, especially when you have tables that contain a large number of rows. In contrast to a table scan, when SQL Server uses an index, it can quickly search for the rows that satisfy your query. Index searches can have a significant impact on both the performance of queries and the performance of your server as a whole. Just as using the index in the back of a book enables you to quickly find the information you need, so does an index enable SQL Server to quickly identify the information that satisfies a query.

You can create an index on one or more columns in a table. The column on which you base an index is called the index key. SQL Server stores the values that make up the index key in the index. When you execute a query for which SQL Server can use an index, it automatically uses the index to retrieve the query's results set. The index contains the contents of the index key column, along with row pointers that point to the original rows in the table on which you based the index. If your query selects columns that are not stored in the index, SQL Server uses the row pointers to retrieve the data from the other columns.

The following figure shows a query execution plan. SQL Server uses the query execution plan to explain how it retrieves the data that satisfies a query. You read a query execution plan from right to left. In this example, the query execution plan shows that SQL Server uses an index named titles.idx_bktitle to retrieve the `SELECT` query's results set.



```
Query 1: Query cost (relative to the batch): 100%
SELECT bktitle FROM titles WHERE bktitle LIKE 'a%'
```

SELECT query
SELECT
Cost: 0 %
Index Seek (NonClustered)
[Titles].[idx_bktitle]
Cost: 100 %
SQL Server uses an index to retrieve query's results

Figure 5-1: Using an index to retrieve the results of a query.

Types of Indexes

SQL Server supports two types of indexes: clustered and nonclustered. With a clustered index, SQL Server stores the table itself in order by the clustered index key. By default, SQL Server creates a table's clustered index using the table's primary key. For example, earlier in the course, you configured the partnum column in the Titles table as the primary key for the table. When you added the primary key constraint, SQL Server automatically reorganized the table so that it stored the table in order by the partnum column. If you do not define a primary key for a table, you can create a clustered index that defines the order in which SQL Server stores the table's rows.

In contrast to a clustered index, SQL Server stores nonclustered indexes as separate database objects. The contents of the index include the index key column(s) plus pointers to the original rows in the table. When SQL Server uses a nonclustered index to satisfy a query, it retrieves the index keys from the index and then uses the row pointers to retrieve the rest of the data for a given set of rows.

You can configure both clustered and nonclustered indexes as unique. When you create an index as unique, you prevent SQL Server from storing identical values in the index key column(s). By default, SQL Server creates a unique clustered index for a table when you assign a primary key constraint to that table.

The following table describes the types of indexes you can create and the SQL statement you use to create them.

Type of Index	*Description*
Clustered index	Physically sorts the values in a column based on the index key values. You can define only one clustered index for a table. This type of index is effective in organizing data in columns where a range of values is stored, such as employee IDs or serially numbered primary key values. When you create a primary key, SQL Server automatically creates a clustered index on the primary key column. The syntax to create a clustered index is: `CREATE [UNIQUE] CLUSTERED INDEX indexname ON tablename(columnname1,[ columnname2])`
Nonclustered index	Does not physically sort the values in the table. Instead, SQL Server stores the index key for each row in the table, along with a pointer to the table row, in the index. You can define a maximum of 249 nonclustered indexes for a table. The syntax to create a nonclustered index is: `CREATE [UNIQUE] NONCLUSTERED INDEX indexname ON tablename(columnname1 [, columnname2])`

The Query Optimizer

You can significantly improve the performance of SQL Server by reducing disk input/output (I/O) operations by implementing indexes. Keep in mind that in a computer, the slowest component is typically the hard disk. Thus, any strategy that minimizes how often a computer accesses the hard disk will improve an application's performance.

When you execute a query, SQL Server uses a component called the query optimizer to generate an execution plan for retrieving the data that satisfies the query. Choices available to the query optimizer include:

- A table scan. With a table scan, SQL Server examines or scans every row in the table and determines whether each row belongs in the results set. A table scan uses the most disk I/O and has the greatest impact on the server's performance.
- An index scan. With an index scan, SQL Server examines every row in the index to identify which rows belong in the results set. If necessary, SQL Server then uses the record pointers in the index to retrieve the rest of the data for the rows. For example, if you execute the query `SELECT bktitle, partnum FROM titles`, and you created an index on the bktitle column of the Titles table, the query optimizer can use the index to retrieve the title of each book and then use the record pointers in the index to retrieve the part number columns from the table itself. An index scan uses less system resources than a table scan but is more resource-intensive than an index seek.

- An index seek. With an index seek, SQL Server searches the index to retrieve only certain rows. The query optimizer chooses an index seek only when you use a `SELECT` query with a `WHERE` clause. If possible, SQL Server attempts to satisfy the results of the query from the index alone. SQL Server can satisfy the results of a query from only the index if the index keys contain the columns you specify after the `SELECT` statement. For this reason, when you design the indexes for a table, you should first consider the types of queries you will execute most often. Creating indexes to support these queries will improve your server's overall performance.

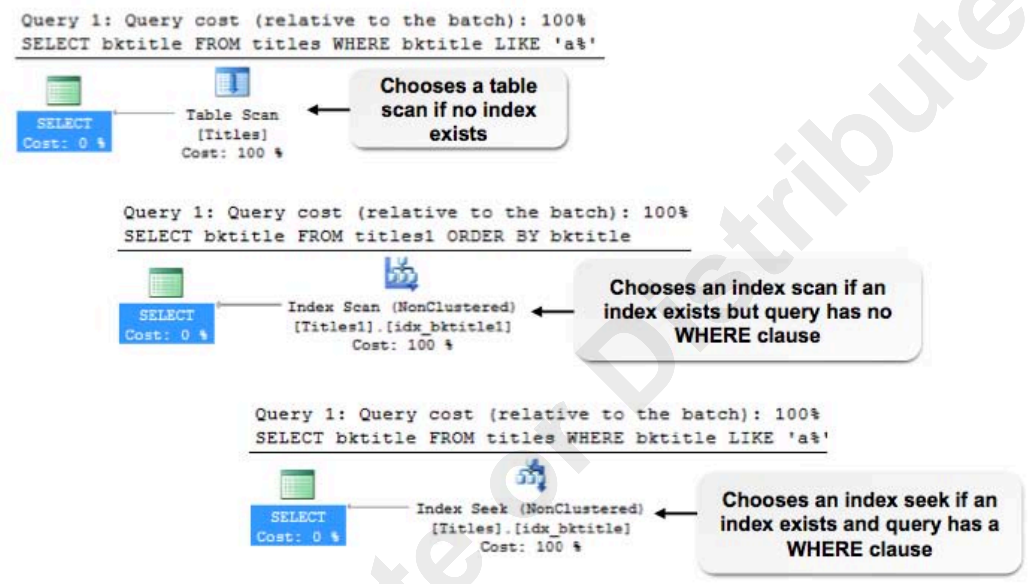
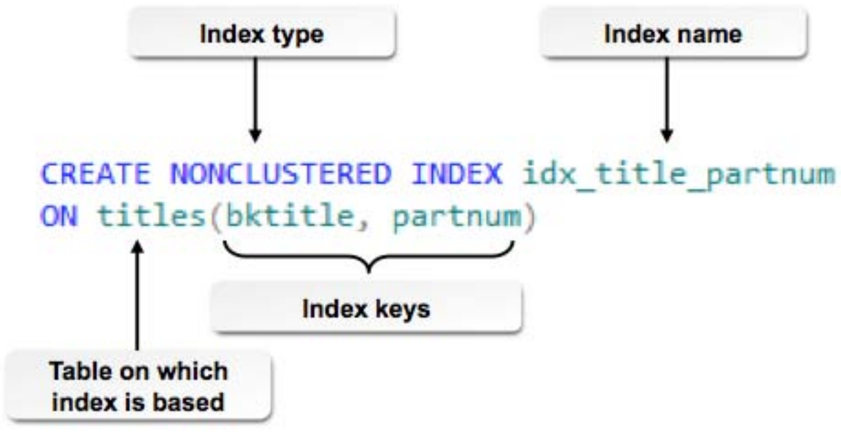

```
Query 1: Query cost (relative to the batch): 100%
SELECT bktitle FROM titles WHERE bktitle LIKE 'a%'
```
SELECT
Cost: 0 %
Table Scan
[Titles]
Cost: 100 %
Chooses a table scan if no index exists
```
Query 1: Query cost (relative to the batch): 100%
SELECT bktitle FROM titles1 ORDER BY bktitle
```
SELECT
Cost: 0 %
Index Scan (NonClustered)
[Titles1].[idx_bktitle1]
Cost: 100 %
Chooses an index scan if an index exists but query has no WHERE clause
```
Query 1: Query cost (relative to the batch): 100%
SELECT bktitle FROM titles WHERE bktitle LIKE 'a%'
```
SELECT
Cost: 0 %
Index Seek (NonClustered)
[Titles].[idx_bktitle]
Cost: 100 %
Chooses an index seek if an index exists and query has a WHERE clause

Figure 5-2: The query optimizer choices.

	Note: To further explore execution plans, you can access the LearnTO **Analyze SQL Server Execution Plans** presentation from the **LearnTO** tile on the LogicalCHOICE Course screen.

The CREATE INDEX Statement

You use the *CREATE INDEX statement* to create a clustered or nonclustered index on one or more table columns. The columns on which you create the index become the index's keys. Define the name of the index immediately after the `CREATE INDEX` statement. Next, you use the `ON` keyword followed by the name of the table that contains the column(s) on which you want to create the index. Finally, you write the name of the column or columns on which you want to create the index enclosed within parentheses. If you specify multiple columns, you must separate the names with commas.



```
CREATE NONCLUSTERED INDEX idx_title_partnum
ON titles(bktitle, partnum)
```

Figure 5-3: A CREATE INDEX statement example.

	Access the Checklist tile on your LogicalCHOICE course screen for reference information and job aids on How to Create Indexes.

ACTIVITY 5-1
Creating Indexes

Before You Begin

- Ensure SQL Server Management Studio is open.
- On the **Standard** toolbar, select **New Query**.
- On the **SQL Editor** toolbar, in the **Database** drop-down list, ensure that the FullerAckerman database is selected.

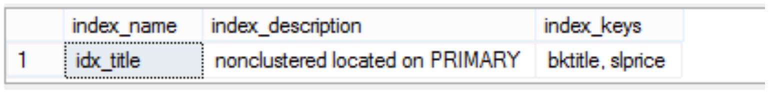
Scenario

With the holiday season approaching, book sales are expected to increase, and management at the Fuller & Ackerman Publishing bookstore have asked you to prepare to generate sales reports and other statistical reports based on sales figures. Because of the anticipated rise in sales figures, the number of titles stocked for sale will increase. The database administrator estimates some reduction in the efficiency of queries when retrieving records from the Sales table for creating reports. Typically, sales reports include the part numbers, customer numbers, and representative IDs. The database administrator also feels that the number of queries to the Titles table will increase during the holidays because of customers searching for specific books. When customers search for books, the search returns the part number, book title, and sale price of a book. As a preemptive measure, the administrator wants you to create indexes to make these queries as efficient as possible. For information on the tables and column names in the FullerAckerman database, refer to the table structure in Appendix A.

1. **You have defined primary keys for the Sales and Titles tables. Considering this fact, which logic can you use to improve the performance of queries that retrieve records from the Sales and Titles tables?**
   - ○ Limit the number of records stored in the Sales and Titles tables.
   - ○ Create clustered indexes on each of the frequently retrieved columns of the Sales and Titles tables.
   - ○ Create nonclustered indexes on frequently retrieved columns of the Sales and Titles tables.
   - ○ Create unique indexes on frequently retrieved columns of the Sales and Titles tables.

2. Create a nonclustered index on the frequently queried columns of the Titles table. Verify that you successfully created the index.
   a) Enter the portion of the query to create the nonclustered index and assign a name to it. Use `CREATE NONCLUSTERED INDEX idx_title ON` to specify the `CREATE INDEX` statement and the `ON` keyword with the index name idx_title:
   ```
   CREATE NONCLUSTERED INDEX idx_title
   ON
   ```
   b) Specify Titles as the table and bktitle and slprice as the index keys:
   ```
   CREATE NONCLUSTERED INDEX idx_title
   ON titles(bktitle, slprice)
   ```
   c) Execute the `CREATE INDEX` statement to create the index.
   d) Verify that you successfully created the index:
   ```
   sp_help titles
   ```

e) In the **Results** pane, observe the index information for the Titles table. You can see that you created a nonclustered index on the bktitle plus slprice columns.



	index_name	index_description	index_keys
1	idx_title	nonclustered located on PRIMARY	bktitle, slprice

3. Create a nonclustered index on the frequently used columns of the Sales table. Verify that you successfully created the index.

a) Delete the previous queries.

b) Create a nonclustered index named idx_sales on the custnum, partnum, and repid columns of the Sales table:

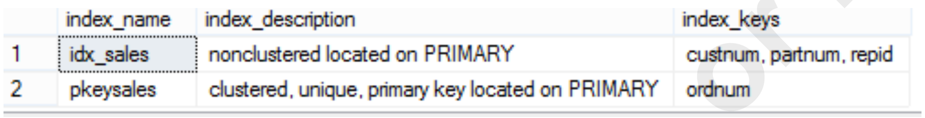
```
CREATE NONCLUSTERED INDEX idx_sales
ON sales(custnum, partnum, repid)
```

c) Execute the `CREATE INDEX` statement.

d) Verify that you successfully created the index:

```
sp_help sales
```

e) You should see the index you just created, idx_sales, plus the clustered index named pkeysales you created when you defined a primary key constraint on the Sales table.



	index_name	index_description	index_keys
1	idx_sales	nonclustered located on PRIMARY	custnum, partnum, repid
2	pkeysales	clustered, unique, primary key located on PRIMARY	ordnum

4. Create a unique index on the partnum column of the Titles table. Verify that you successfully created the index.

a) Delete the previous queries.

b) Create a unique index named idx_partnum on the partnum column of the Titles table:

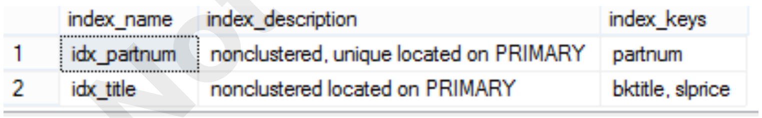
```
CREATE UNIQUE INDEX idx_partnum
ON titles(partnum)
```

c) Execute the `CREATE INDEX` statement to create the idx_partnum index.

d) Display the structure of the Titles table to observe whether you successfully created the idx_partnum index:

```
sp_help titles
```

e) You see the two indexes you created in this activity: idx_partnum and idx_title.



	index_name	index_description	index_keys
1	idx_partnum	nonclustered, unique located on PRIMARY	partnum
2	idx_title	nonclustered located on PRIMARY	bktitle, slprice

f) Close the **Query Editor** window without saving the query.

TOPIC B

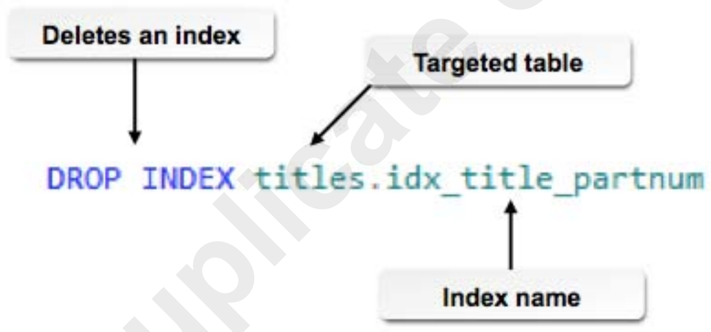
Drop Indexes

One of the server maintenance tasks you might need to perform is to delete indexes that you no longer find useful. You can delete an index by using the `DROP INDEX` statement. In this topic, you will learn to use the `DROP INDEX` statement to delete indexes from your database.

There is a cost to SQL Server for maintaining indexes. In essence, because SQL Server stores the index keys in the index, indexes duplicate the data stored in your tables. After analyzing the types of queries users execute on the server, you might find that the cost of maintaining an index exceeds the benefit it offers. For this reason, you need to know how to delete indexes that you no longer need.

The DROP INDEX Statement

You use the *DROP INDEX statement* to delete an index created on a table. You use this statement to delete all clustered indexes except for a clustered index created when you defined a primary key or `UNIQUE` constraint for the table. To delete a clustered index created for a primary key or `UNIQUE` constraint, you must drop the constraint, not the index. To delete an index, use `DROP INDEX` followed by the table name on which you based the index, a period, and then the name you assigned to the index. If you aren't sure of the name of an index, execute `sp_help` *`table_name`* to obtain a list of the indexes defined on the table.



```
DROP INDEX titles.idx_title_partnum
```

Figure 5-4: A DROP INDEX statement example.

Access the Checklist tile on your LogicalCHOICE course screen for reference information and job aids on How to Drop Indexes.

ACTIVITY 5-2
Dropping Indexes

Data File

DrpIndex_Starter.sql

Before You Begin

- From the **C:\094006Data\Indexing Data** folder, open the **DrpIndex_Starter.sql** file.
- In SQL Server, on the toolbar, in the **Database** drop-down list, verify that you've selected the FullerAckerman database.
- Execute the script and then close the **Query Editor** window.
- Select **New Query** to open a new query window.
- On the toolbar, in the **Database** drop-down list, ensure that the FullerAckerman database is selected.

Scenario

The database administrator for the Fuller & Ackerman Publishing bookstore created indexes named pot_cust_ind_nonclus and ob_ttl_ind_nonclus on tables that record potential customer and obsolete title information. However, after reviewing the performance of the indexes and their general usage, the administrator concluded that the cost to SQL Server to maintain the indexes outweighs their benefit. The administrator wants you to drop the indexes from the Potential_customers and Obsolete_titles tables. For information on the tables and column names in the FullerAckerman database, refer to the table structure in Appendix A.

1. Drop the index named pot_cust_ind_nonclus from the Potential_customers table. Verify that you successfully deleted the index.
   a) Type the following query to delete the pot_cust_ind_nonclus index:
      ```
      DROP INDEX potential_customers.pot_cust_ind_nonclus
      ```
   b) Execute the `DROP INDEX` statement to drop the index.
   c) Verify that you successfully deleted the index:
      ```
      sp_help potential_customers
      ```
   d) In the **Results** pane, scroll down. You no longer see the pot_cust_ind_nonclus index listed for the table.

2. Drop the index named ob_ttl_ind_nonclus from the Obsolete_titles table and verify this change.
   a) Delete the previous query.
   b) Write the query to drop the index:
      ```
      DROP INDEX obsolete_titles.ob_ttl_ind_nonclus
      ```
   c) Execute the `DROP INDEX` statement to drop the index.
   d) Verify that you successfully deleted the index:
      ```
      sp_help obsolete_titles
      ```
   e) You should not see the index listed for this table.
   f) Close the **Query Editor** window without saving the query.

Summary

You can create and drop indexes as your organization's business requirements change. You use indexes to optimize the performance of the queries your users execute most often.

How effective do you think indexing will be in speeding up record retrieval in the databases maintained by your organization?

Which index type might you implement on a table with employee data for your organization?

Note: Check your LogicalCHOICE Course screen for opportunities to interact with your classmates, peers, and the larger LogicalCHOICE online community about the topics covered in this course or other topics you are interested in. From the Course screen you can also access available resources for a more continuous learning experience.

6 Managing Transactions

Lesson Time: 25 minutes

Lesson Objectives

In this lesson, you will manage transactions. You will:

- Create and roll back transactions.
- Commit transactions.

Introduction

One of the strategies you can use to make sure all changes in a SQL query occur successfully is to explicitly mark the beginning and the end of transactions. By explicitly marking the beginning of a transaction, if an error occurs or the power fails, you can easily undo the transaction by rolling it back. You use transactions when you want a query's changes to be atomic: either all the changes succeed, or they all fail. You implement transactions by using the `BEGIN TRANSACTION` statement. After you've made the necessary changes and no errors occur, you use the `COMMIT TRANSACTION` statement. On the other hand, if an error occurs, you can undo your changes by executing the `ROLLBACK TRANSACTION` statement. In this lesson, you will learn how to create, roll back, and commit transactions.

Imagine you are the database administrator for a large online bookstore's online transaction processing (OLTP) system. At any given moment, you could have hundreds or even thousands of transactions occurring. Now think about what types of changes a new customer might make when buying a book: the customer must register with your website, define their mailing address, and give their credit card information. In the background, your system must verify that the credit card number is valid and that the customer's transaction is approved. In addition, your system must pull the book out of inventory and generate a packing list for the warehouse to ship the book. If anything goes wrong while processing all the changes for customer's purchase, what will happen? In this scenario, you can easily undo the changes that have occurred if you use the `ROLLBACK TRANSACTION` statement.

TOPIC A

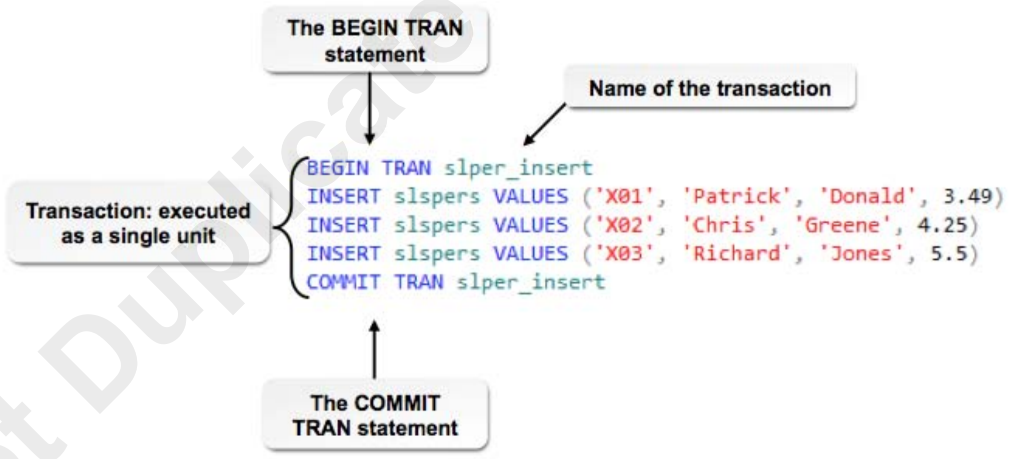
Create Transactions

You implement transactions by using the BEGIN TRANSACTION statement. During the transaction, if an error occurs, you can undo the transaction's changes by using the ROLLBACK TRANSACTION statement. In this topic, you will create and roll back a transaction.

When you pay a bill online, you typically see a message box that asks you to confirm whether you want to proceed with the payment. If you do not want to proceed with the payment, you can cancel the payment. The ROLLBACK TRANSACTION feature works in a similar manner. After inserting and updating records, if you want to cancel the data that you have inserted, you can use the ROLLBACK TRANSACTION statement. However, to use the rollback feature, you first need to create transactions.

Transactions

A *transaction* is a collection of SQL statements that you execute as a single unit. Use the BEGIN TRANSACTION or BEGIN TRAN statement to mark the beginning of a transaction, followed by an optional name for the transaction. Next, enter your SQL statements you want to execute as part of the transaction. Finally, you complete the transaction by adding a COMMIT TRANSACTION or COMMIT TRAN statement. You typically use transactions within stored procedures instead of when you're typing ad hoc queries in Microsoft® SQL Server®.


The BEGIN TRAN statement
Name of the transaction
Transaction: executed as a single unit
The COMMIT TRAN statement

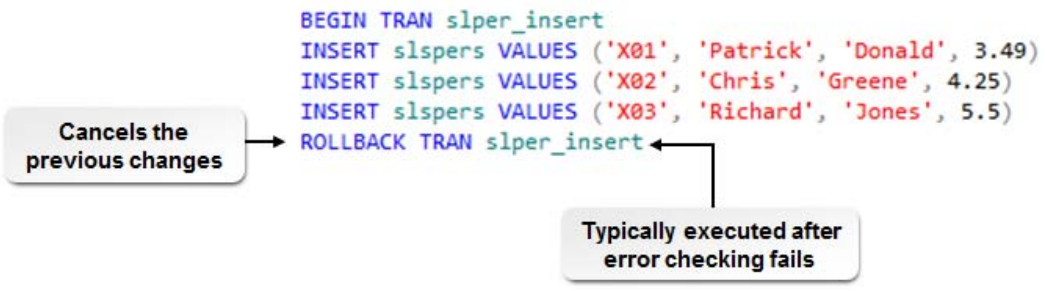
```
BEGIN TRAN slper_insert
INSERT slspers VALUES ('X01', 'Patrick', 'Donald', 3.49)
INSERT slspers VALUES ('X02', 'Chris', 'Greene', 4.25)
INSERT slspers VALUES ('X03', 'Richard', 'Jones', 5.5)
COMMIT TRAN slper_insert
```

Figure 6-1: Sample code for executing a transaction.

In the example, the BEGIN TRAN statement creates a transaction named slper_insert. The next three lines of code insert three new rows into the Slspers table. Finally, the COMMIT TRAN statement configures SQL Server to create the three new rows in the Slspers table.

The ROLLBACK TRAN Statement

If a problem occurs during a transaction, you use the *ROLLBACK TRAN statement* to cancel the transaction. SQL Server rolls back or undoes all of the changes made by the transaction. Keep in mind that you typically use transactions when you create a stored procedure that you plan to call from a program and you use error-checking to determine whether you should commit the transaction or roll it back.



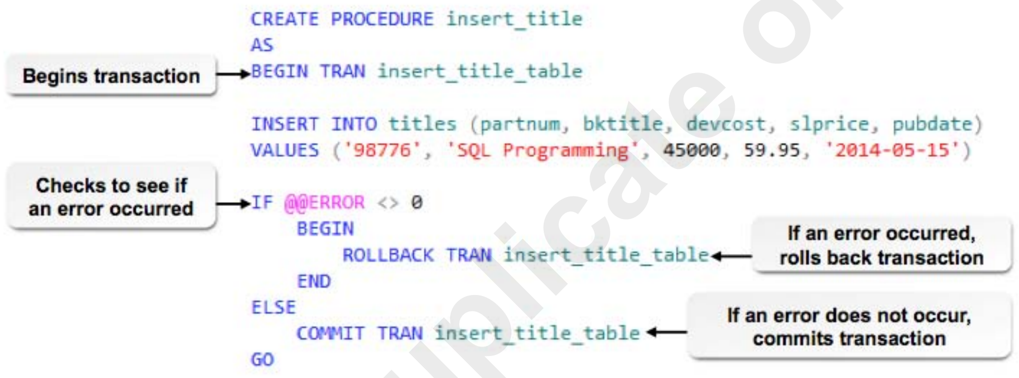
```
BEGIN TRAN slper_insert
INSERT slspers VALUES ('X01', 'Patrick', 'Donald', 3.49)
INSERT slspers VALUES ('X02', 'Chris', 'Greene', 4.25)
INSERT slspers VALUES ('X03', 'Richard', 'Jones', 5.5)
ROLLBACK TRAN slper_insert
```

Cancels the previous changes
Typically executed after error checking fails

Figure 6-2: A ROLLBACK TRAN statement example.

In the example, the `ROLLBACK TRAN` statement prevents the insertion of the three rows in the transaction.

In this next example, the SQL query creates a stored procedure named insert_title that begins a transaction named insert_title_table. Next, the stored procedure inserts a row into the Titles table. The stored procedure then checks to see if an error occurred by using the statement `IF @@ERROR <> 0`. SQL Server uses a system variable, `@@ERROR`, to store error codes when an error occurs. If no error occurs, `@@ERROR` has a value of 0. Thus, if `@@ERROR` is not equal to zero, an error occurred, and the stored procedure rolls back the transaction. If the statement `@@ERROR <> 0` is false, meaning `@@ERROR` is zero, no error occurred and the stored procedure commits the transaction.



```
CREATE PROCEDURE insert_title
AS
BEGIN TRAN insert_title_table

INSERT INTO titles (partnum, bktitle, devcost, slprice, pubdate)
VALUES ('98776', 'SQL Programming', 45000, 59.95, '2014-05-15')

IF @@ERROR <> 0
    BEGIN
        ROLLBACK TRAN insert_title_table
    END
ELSE
    COMMIT TRAN insert_title_table
GO
```

Begins transaction
Checks to see if an error occurred
If an error occurred, rolls back transaction
If an error does not occur, commits transaction

Figure 6-3: A transaction in a stored procedure.

Note: To further explore programming stored procedures, you can access the LearnTO **Create and Use a Stored Procedure** presentation from the **LearnTO** tile on the LogicalCHOICE Course screen.

Access the Checklist tile on your LogicalCHOICE course screen for reference information and job aids on How to Create Transactions.

ACTIVITY 6-1
Creating a Transaction

Data Files

Insert_Records1.txt

Insert_Records2.txt

Before You Begin

- Ensure SQL Server Management Studio is open.
- On the **Standard** toolbar, select **New Query**.
- On the **SQL Editor** toolbar, in the **Database** drop-down list, ensure that the FullerAckerman database is selected.

Scenario

With a new set of salespeople joining the Fuller & Ackerman Publishing bookstore, you have been given the task of entering their information into the Slspers table. As you are creating the statements for inserting data, you are called to an urgent meeting by your manager. After you return, you continue inserting data. However, you realize that the records you inserted after returning from the meeting contain erroneous information. So, you decide to undo the records. For information on the tables and column names in the FullerAckerman database, refer to the table structure in Appendix A.

1. **True or False? You have created the statements to insert three records. To implement the work you have done and save it, you would place the statements within a `BEGIN TRAN` block and save your changes using a `COMMIT TRAN` statement.**

   ☐ True

   ☐ False

2. Create and save a transaction named slperInsert.
   a) Enter the query to begin the transaction:
   ```
   BEGIN TRAN slperInsert
   ```
   b) Navigate to the **C:\094006Data\Managing Transactions** folder and open the **Insert_Records1.txt** file in Notepad. Copy the contents, and paste them into the **Query Editor** window.
   c) On the next line, commit the transaction:
   ```
   COMMIT TRAN slperInsert
   ```
   d) Execute the statements to commit the transaction.

3. Create a new transaction named slperInsert2 and add three new salespeople to the Slspers table.
   a) Delete the previous statements.
   b) Begin a new transaction named slperInsert2:
   ```
   BEGIN TRAN slperInsert2
   ```
   c) Navigate to the **C:\094006Data\Managing Transactions** folder and open the **Insert_Records2.txt** file in Notepad. Copy the contents, and paste them after the `BEGIN TRAN slperInsert2` statement.
   d) You realize you've made a mistake and don't want to add the three people to the Slspers table. On the next line, roll back the transaction.
   ```
   ROLLBACK TRAN slperInsert2
   ```
   e) Execute the statements.

f) To verify whether the records you rolled back exist in the table, query the Slspers table. Use the `IN` operator in the search condition of the query to search based on the representative ID values in the inserted records:

```
SELECT *
FROM slspers
WHERE repid IN ('Y01', 'Y02', 'Y03')
```

g) Execute the `SELECT` statement. Because you rolled back the transaction, the results set contains no rows.

h) Close the **Query Editor** window without saving the query.

4. True or False? The records inserted before the meeting also contain incorrect information. You can roll back the transaction slperInsert by using the **`ROLLBACK TRAN`** statement.

☐ True

☐ False

TOPIC B

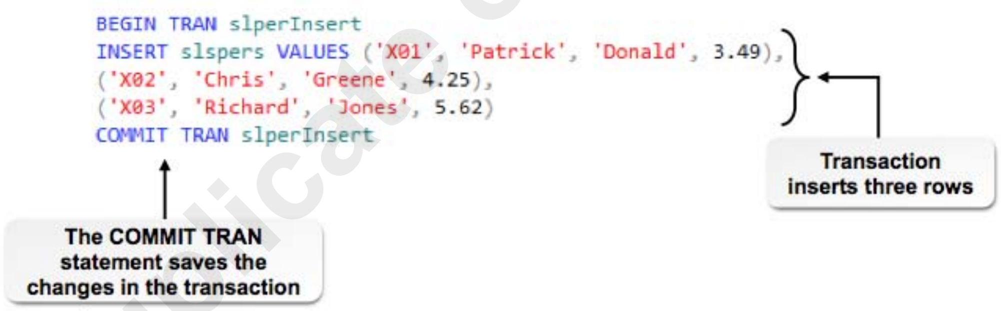
Commit Transactions

In the last topic, you created and rolled back transactions. When you are ready to write the transactions to a database, you can commit the transactions. In this topic, you will commit transactions.

When you use an important command in Windows®, it prompts you to confirm that you want to execute the command. For example, when you delete all the files in a folder, Windows prompts you to confirm the deletion. Similarly, when you execute a set of transactions, SQL enables you to confirm you want to save your changes by using the `COMMIT TRAN` statement.

The COMMIT TRAN Statement

When you execute a transaction, you use the *COMMIT TRAN statement* to save the data modifications made by the SQL statements within the transaction. The `COMMIT TRAN` statement saves the data modifications made by all SQL statements executed after the `BEGIN TRAN` statement. If you assigned a name to the transaction in the `BEGIN TRAN` statement, you must put the name of the transaction after the `COMMIT TRAN` statement. After you execute the `COMMIT TRAN` statement for a transaction, you cannot roll back the data modifications made by the SQL statements in the transaction.



```
BEGIN TRAN slperInsert
INSERT slspers VALUES ('X01', 'Patrick', 'Donald', 3.49),
('X02', 'Chris', 'Greene', 4.25),
('X03', 'Richard', 'Jones', 5.62)
COMMIT TRAN slperInsert
```

Transaction inserts three rows
The COMMIT TRAN statement saves the changes in the transaction

Figure 6-4: Sample code for the COMMIT TRAN statement.

Access the Checklist tile on your LogicalCHOICE course screen for reference information and job aids on How to Commit Transactions.

ACTIVITY 6-2
Committing Transactions

Data File

Insert_Records3.txt

Before You Begin

- Ensure SQL Server Management Studio is open.
- On the **Standard** toolbar, select **New Query**.
- On the **SQL Editor** toolbar, in the **Database** drop-down list, ensure that the FullerAckerman database is selected.

Scenario

With a new set of salespeople joining the Fuller & Ackerman Publishing bookstore, you have been given the task of entering their information into the Slspers table. The information for the new salespeople is listed in the following table. You want to be sure that all your changes are saved to the table successfully. For information on the tables and column names in the FullerAckerman database, refer to the table structure in Appendix A.

Representative ID	*First Name*	*Last Name*	*Commission Rate*
A01	Patrick	Donald	4.39
A02	Chris	Greene	2.45
A03	Richard	Jones	6.52
B01	John	Webber	5.69
B02	Ellisa	Peterson	3.49
B03	Julie	Tuscano	5.59

Create a transaction to insert the records into the Slspers table.

a) Enter the statement to begin the transaction:

```
BEGIN TRAN slspersInsert
```

b) Navigate to the **C:\094006Data\Managing Transactions** folder and open the **Insert_Records3.txt** file in Notepad. Copy its contents and paste them into the **Query Editor** window:

```
BEGIN TRAN slspersInsert
INSERT slspers VALUES ('A01', 'Patrick', 'Donald', 4.39),
('A02', 'Chris', 'Greene', 2.45),
('A03', 'Richard', 'Jones', 6.52),
('B01', 'John', 'Webber', 5.69),
('B02', 'Ellisa', 'Peterson', 3.49),
('B03', 'Julie', 'Tuscano', 5.59)
```

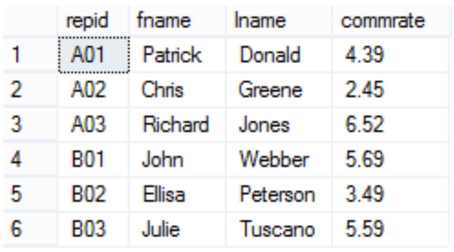
c) On the next line, add the statement to commit the transaction:

```
BEGIN TRAN slspersInsert
INSERT slspers VALUES ('A01', 'Patrick', 'Donald', 4.39),
('A02', 'Chris', 'Greene', 2.45),
('A03', 'Richard', 'Jones', 6.52),
('B01', 'John', 'Webber', 5.69),
                     ('B02', 'Ellisa', 'Peterson', 3.49),
                     ('B03', 'Julie', 'Tuscano', 5.59)
                     COMMIT TRAN slspersInsert
```

d) Execute the statements.
e) Delete the previous query.
f) To verify that you successfully inserted the rows, execute the following query:

```
SELECT *
FROM slspers
WHERE repid IN ('A01', 'A02', 'A03', 'B01', 'B02', 'B03')
```

g) You should see that you inserted six new rows.



	repid	fname	lname	commrate
1	A01	Patrick	Donald	4.39
2	A02	Chris	Greene	2.45
3	A03	Richard	Jones	6.52
4	B01	John	Webber	5.69
5	B02	Ellisa	Peterson	3.49
6	B03	Julie	Tuscano	5.59

h) Close SQL Server Management Studio.

Summary

You create and commit transactions to make sure that all the changes within a query complete successfully, or you roll back transactions to make sure that all of the changes are reverted. By using transactions, you ensure that the data in your database is valid and consistent across all tables.

How do you think you might implement transactions in your organization?

What benefit does using transactions offer you?

Note: Check your LogicalCHOICE Course screen for opportunities to interact with your classmates, peers, and the larger LogicalCHOICE online community about the topics covered in this course or other topics you are interested in. From the Course screen you can also access available resources for a more continuous learning experience.

Course Follow-Up

Organizations use SQL databases to store one of their most precious resources: data. Organizations use their data to track customers' sales, monitor inventory, and design marketing strategies. To support these activities, you must know how to not only query the data, but also how to perform tasks such as creating and populating tables, managing indexes to optimize performance, and implementing views to simplify users' access to data.

In this course, you used SQL statements and their clauses to perform a number of complex tasks. You used subqueries to retrieve and display records. You inserted, updated, and deleted rows in tables using advanced querying techniques. In addition, you modified the structure of tables in a database. By using views, you displayed specific parts of a table and multiple tables. You also indexed the columns in a table to streamline the performance of SQL Server. Finally, you created, saved, and rolled back transactions; you used transactions to ensure that SQL Server saves either all your changes or none of the changes. Implementing transactions ensures the validity of your database.

What's Next?

You are encouraged to explore *SQL Querying: Advanced* further by actively participating in any of the social media forums set up by your instructor or training administrator through the **Social Media** tile on the LogicalCHOICE Course screen.

A The FullerAckerman Database

The database used in this book, called the FullerAckerman database, is being used in a hypothetical bookstore company called Fuller & Ackerman Publishing. The following tables make up the FullerAckerman database:

- The Customers table describes each of Fuller & Ackerman Publishing's customers.
- The Sales tables describes each book sale. The table named Sales1 is a copy of this table.
- The Slspers table describes each salesperson working at Fuller & Ackerman Publishing.
- The Titles table describes each book produced by Fuller & Ackerman Publishing. The table named Titles1 is a copy of this table.
- The Obsolete_titles table describes all books that are out of print.
- The Potential_customers table describes any possible new customers for Fuller & Ackerman Publishing.

Structure of the Slspers Table

Column Name	*Data Type*	*Description*
repid	nvarchar (3) NULL	The representative ID is an alphanumeric ID that uniquely identifies each salesperson.
fname	nvarchar (10) NULL	The first name of a salesperson.
lname	nvarchar (20) NULL	The last name of a salesperson.
commrate	float NULL	The commission rate earned by each salesperson.

Structure of the Sales and Sales1 Tables

Column Name	*Data Type*	*Description*
ordnum	nvarchar (5) NULL	The order number that uniquely identifies each sale.
sldate	smalldatetime NULL	The date of sale.
qty	int NULL	The quantity of units sold for an order.
custnum	nvarchar (5) NULL	The customer number.
partnum	nvarchar (5) NULL	The part number that uniquely identifies the book sold.
repid	nvarchar (3) NULL	The representative ID that uniquely identifies the salesperson who contributed to the sale.

Structure of the Titles and Titles1 Tables

Column Name	*Data Type*	*Description*
partnum	nvarchar (5) NULL	The part number of a book that uniquely identifies the book.
bktitle	nvarchar (40) NULL	The title of a book.
devcost	money NULL	The cost of developing the book.
slprice	money NULL	The sale price of a book.
pubdate	smalldatetime NULL	The publishing date of a book.

Structure of the Customers Table

Column Name	*Data Type*	*Description*
custnum	nvarchar (5) NULL	The customer number that uniquely identifies a customer.
referredby	nvarchar (5) NULL	The customer number that represents a customer who referred the bookstore.
custname	nvarchar (30) NULL	The name of the customer.
address	nvarchar (25) NULL	The address of the customer.
city	nvarchar (20) NULL	The city where the customer resides.
state	nvarchar (2) NULL	The state where the customer's city is located.
zipcode	nvarchar (12) NULL	The zip code of the customer's state.
repid	nvarchar (3) NULL	The representative ID that identifies the salesperson who sold books to the customer.

Structure of the Obsolete_titles Table

Column Name	*Data Type*	*Description*
partnum	varchar (10) NOT NULL	The part number of a book that uniquely identifies the book.
bktitle	varchar (40) NOT NULL	The title of a book.
devcost	money	The cost of developing the book.
slprice	money	The sale price of a book.
pubdate	smalldatetime	The publishing date of a book.

Structure of the Potential_customers Table

Column Name	*Data Type*	*Description*
custnum	varchar (5) NULL	The customer number that uniquely identifies a customer.
referredby	varchar (5) NULL	The customer number that represents a customer who recommended the bookstore.
custname	varchar (30) NULL	The name of the customer.
address	varchar (50) NULL	The address of the customer.

Column Name	*Data Type*	*Description*
mobileno	varchar (10) NOT NULL	The customer's cell phone number.
repid	varchar (3) NULL	The representative ID that identifies the salesperson who sold books to the customer.

Lesson Labs

Lesson labs are provided for certain lessons as additional learning resources for this course. Lesson labs are developed for selected lessons within a course in cases when they seem most instructionally useful as well as technically feasible. In general, labs are supplemental, optional unguided practice and may or may not be performed as part of the classroom activities. Your instructor will consider setup requirements, classroom timing, and instructional needs to determine which labs are appropriate for you to perform, and at what point during the class. If you do not perform the labs in class, your instructor can tell you if you can perform them independently as self-study, and if there are any special setup requirements.

Lesson Lab 1-1
Using Subqueries to Perform Advanced Querying

Activity Time: 15 minutes

Before You Begin

- Ensure SQL Server Management Studio is open.
- On the **Standard** toolbar, select **New Query**.
- On the **SQL Editor** toolbar, in the **Database** drop-down list, ensure that the FullerAckerman database is selected.

Scenario

The top management of the Fuller & Ackerman Publishing bookstore chain has been monitoring the progress of sales in the store. The financial results of the bookstore chain show that the store has performed exceedingly well. Management wants to reward its employees for their contributions, and has decided to give a bonus to all employees. In addition, the top performer from the Sales department is to be rewarded with a special bonus and a raise. The vice president of sales wants you to identify the sales representative who has sold the most copies of books for the year. Use the Sales and Slspers tables to generate the required output. To identify the column names in the table, refer to Appendix A.

1. Determine the logic to write a subquery that generates information on the sales representatives who have sold the maximum number of copies.
2. Frame the outer query to display the names and representative IDs of each sales representative.
3. Frame the inner levels of queries and execute the subquery to group records by repid.
4. Execute the query. Kent Allard is the sales representative who has sold the most copies of books for the year.

Lesson Lab 2-1
Manipulating Table Data

Activity Time: 15 minutes

Before You Begin

- Ensure SQL Server Management Studio is open.
- On the **Standard** toolbar, select **New Query**.
- On the **SQL Editor** toolbar, in the **Database** drop-down list, ensure that the FullerAckerman database is selected.

Scenario

You are inserting the details of an old book into the Obsolete_titles table. The book's part number is 39999, the title is "Learn to Play the Violin," the development cost is $5,000, sales price is $45, and the publication date is 11/05/2010. Later, you realize that you have entered the book title wrong. The book title should be "Learn to Play the Mandolin." So, you modify this record by replacing the wrong book title with the correct one. After a few months, you learn that this book has no value in the market and you decide to remove the title from the Obsolete_titles table.

1. Insert the record for the book into the Obsolete_titles table and verify whether the record is inserted into the table.

2. Modify the book title and verify whether you've successfully changed the record.

3. Delete the record and verify that you deleted it successfully.

Lesson Lab 3-1

Manipulating the Table Structure

Activity Time: 15 minutes

Before You Begin

- Ensure SQL Server Management Studio is open.
- On the **Standard** toolbar, select **New Query**.
- On the **SQL Editor** toolbar, in the **Database** drop-down list, ensure that the FullerAckerman database is selected.

Scenario

The Fuller & Ackerman Publishing bookstore store manager wants to hire several people on a contract basis for a short period during the holidays. The store manager wants you to ensure that the database contains information on the contract ID, first name, last name, and salary of the contractors. You need to create a table called Contractors with the table structure given. After creating the table, you want to ensure that each record is unique and that the salary value for a contractor is always greater than zero. You have also been instructed by the SQL Server administrator to delete the Contractors table from the database after the holidays when the contract expires. The columns you must insert include:

- ctrid varchar (5) NOT NULL
- fname varchar (10)
- lname varchar (10)
- salary money

1. Create the Contractors table based on the column information provided. Verify that you successfully created the table.
2. Add a constraint to the table to ensure the uniqueness of each record. Verify that you added the `PRIMARY KEY` constraint.
3. Add a constraint to prevent salary values below zero from being stored in the table. Verify that you added the `CHECK` constraint.
4. Drop the table and verify that you successfully dropped it.

Lesson Lab 4-1

Working with Views

Activity Time: 15 minutes

Before You Begin

- Ensure SQL Server Management Studio is open.
- On the **Standard** toolbar, select **New Query**.
- On the **SQL Editor** toolbar, in the **Database** drop-down list, ensure that the FullerAckerman database is selected.

Scenario

To celebrate the tenth anniversary of the bookstore, Fuller & Ackerman Publishing plans to mail gifts and discount coupons to its customers. You have been advised by the public relations manager of the bookstore that the courier company servicing the bookstore's mail might require an accurate list of all customer names and their addresses. To implement a method for generating this list in a short span of time, you need to create a view named custview based on the custnum, custname, and address columns of the Customers table. A few weeks after implementing the anniversary plan, you receive a request from the courier company to provide the full address of each customer, including the city, state, and zip codes. Later, the SQL Server administrator asks you to delete the view you created because you no longer need it.

1. Create a view named `custview` to retrieve the customer number, customer name, and address in the Customers table.
2. Verify that you successfully created the `custview` view.
3. Modify the structure of the `custview` view to add the city, state, and zip code information to the address displayed by the view.
4. Query the `custview` view to verify your changes.
5. Delete the `custview` view.
6. Verify that you successfully deleted the `custview` view by attempting to retrieve records from the view.

Lesson Lab 5-1
Indexing Data

Activity Time: 15 minutes

Before You Begin

- Ensure SQL Server Management Studio is open.
- On the **Standard** toolbar, select **New Query**.
- On the **SQL Editor** toolbar, in the **Database** drop-down list, ensure that the FullerAckerman database is selected.

Scenario

The number of customers buying from the Fuller & Ackerman Publishing bookstore has increased tremendously over the last decade of the bookstore's existence and has resulted in a large number of records in the Customers table. To optimize the performance of queries that use the Customers table, you decide to index the table. After reviewing the most frequent queries, you find that most queries have `WHERE` clauses that query the city and state columns. Later, you find that the cost of maintaining this index exceeds the benefit it offers. The database administrator asks you to delete it.

1. Create a nonclustered index named idx_cust on the columns most commonly used for searching the Customers table.
2. Verify that you successfully created the index.
3. Delete the idx_cust index.
4. Verify that you successfully deleted the idx_cust index from the Customers table.

Lesson Lab 6-1
Managing Transactions

Activity Time: 15 minutes

Data File

- Lab_Starter.sql

Before You Begin

- Ensure SQL Server Management Studio is open.
- On the **Standard** toolbar, select **New Query**.
- On the **SQL Editor** toolbar, in the **Database** drop-down list, ensure that the FullerAckerman database is selected.

Scenario

You have joined the Fuller & Ackerman Publishing bookstore as a SQL programmer. The store manager just gave you a list of books that have become obsolete. However, the manager warns you that the information in the list is not accurate and that you may need to undo the additions you make to the table. You want to be sure you can undo your changes if the list of books changes, so you decide to use a transaction. After you add these records to the Obsolete_titles table, you are instructed by the manager to undo the changes and use a new list with the correct information. The manager asks you to ensure that you add the information in this new list to the Obsolete_titles table.

1. In the **C:\094006Data\Managing Transactions** folder, open the **Lab_Starter.sql** file in SQL Server Management Studio. Below the comment `Incorrect Records`, use your mouse to select the `BEGIN TRAN obsInsert` statement through the third `INSERT` statement. On the toolbar, select **Execute** to begin to insert records into the Obsolete_titles table.

2. The store manager tells you she gave you the wrong list of books. With your mouse, select the `ROLLBACK TRAN obsInsert` statement and execute it to undo your changes.

3. Verify that you did not insert the rows listed below the comment `Incorrect Records`.

4. With your mouse, select the statements marked under the comment `Correct Records`, starting with the `BEGIN TRAN` statement through the `COMMIT TRAN` statement to insert the correct records into the Obsolete_titles table. On the toolbar, select **Execute** to begin and commit your changes.

5. Verify that you inserted the new rows into the Obsolete_titles table based on the part numbers.

Solutions

ACTIVITY 1-1: Searching Based on an Unknown Value

1. **Which task do you need to perform to identify commission rates that are above the average commission rate for the sales team?**
   - ○ Generate the average commission rate for the team.
   - ○ Generate the commission rate for each sales representative.
   - ○ Generate the maximum commission rate received by a sales representative.
   - ◉ Compare the commission rate of each sales representative with the average commission rate for the team.

3. **How would you identify books that are priced above the average book price? (Choose two.)**
   - ☑ Generate the average sale price of books, which is an unknown value.
   - ☐ Add the total price of all books and divide the total price by two.
   - ☑ Compare the average sale price with the sale price of each book in the store.
   - ☐ Compare the average sale price with the minimum sale price of the books.

ACTIVITY 1-2: Searching Based on Multiple Unknown Values

1. **True or False? To generate the list of books that have never been sold from the store, you need to verify whether the part number of each book is present in any of the sales recorded in the Sales table.**
   - ☑ True
   - ☐ False

3. **Which task do you need to perform to generate a list of sales representatives who have not made a single book sale?**

   ○ Verify whether the part number of each book is present in any of the sales recorded in the Sales table.

   ◉ Verify whether the representative ID of each representative is present in any of the sales recorded in the Sales table.

   ○ Verify whether the customer ID of each customer is present in any of the sales recorded in the Sales table.

   ○ Verify whether the commission rate of each representative is present in any of the sales recorded in the Sales table.

ACTIVITY 1-3: Searching by Comparing with Unknown Values

1. **Which is the best possible logic you can use to identify customers who fall under the Silver Card plan?**

   ◉ Use an inner query to generate a list of total quantities purchased by each customer and use the outer query's search condition to compare whether the quantity of each purchase is greater than the minimum value in the list of values generated by the inner query.

   ○ Use a subquery to generate a list of total purchase quantities for each customer.

   ○ Use a subquery to generate a list of the top five total purchase quantities for each customer.

   ○ Generate the list of customers whose total purchase quantities are above the average purchase made by a customer in the store.

ACTIVITY 1-4: Searching Based on the Existence of a Record

1. **Which is the appropriate logic to generate the list of sales representatives who have sold books to Gold Card customers? (Choose two.)**

   ☑ Create an inner query that generates records from the Sales table where the quantity of each sale is higher than 400 and where the representative IDs of the Sales and the Slspers tables match.

   ☐ Create an inner query that generates records from the Sales table where the quantity of each sale is less than 400.

   ☐ Create an outer query that verifies whether the records generated by the inner query match with the records to be retrieved.

   ☑ Create an outer query that verifies whether the records generated by the inner query exist. Based on this check, the outer query must generate the records with information on each sales representative.

ACTIVITY 1-5: Generating Output Using Correlated Subqueries

1. **Which is the possible logic you can use to generate the list of top customers of the bookstore? (Choose two.)**

   ☐ Create a subquery with the inner query generating the list of customer numbers and part numbers. Use the output from the inner query in the outer query to generate book titles and customer names.

   ☑ Create a correlated subquery where the outer query generates the book title, customer name, and address when the value 500 is found in a list of values supplied by the inner query.

   ☑ Create a correlated subquery where the inner query generates the list of quantities for those book titles where the customer numbers are matching in the Sales and Customers tables and the part numbers are matching in the Sales and Titles tables.

   ☐ Create a query that generates a list of customer names and book titles.

ACTIVITY 1-6: Filtering Grouped Data Within a Subquery

1. **Which options represent the possible logic needed to generate the list of sales representatives who have equaled or exceeded the target? (Choose two.)**

   ☑ Create an outer query that searches for the representative ID returned by the inner query.

   ☑ Create an inner query that returns a list of representative IDs. The representative IDs must be listed after grouping them based on the repid column and after filtering the grouped IDs based on the condition that the total quantity values sold by a representative is greater than or equal to 2,375.

   ☐ Create an outer query that searches for the total quantity values for each representative returned by the inner query.

   ☐ Create an inner query that generates a list of representative IDs after grouping the records based on the qty column.

ACTIVITY 1-7: Generating Output Using Nested Subqueries

1. **True or False? The most practical method of generating the required customer details is to create a query to identify books that are priced above $49, create another query to identify the customer of each of the books generated by the first query, and use the result of both queries to identify the customer details manually.**

   ☐ True

   ☑ False

2. **True or False? To generate the required customer details, create a nested subquery with the innermost level of the subquery generating the list of part numbers of books that are priced above $49, a second level of the subquery that uses the part numbers to list the customer numbers of customers who bought the books, and an outer query that uses the customer numbers.**

   ☑ True

   ☐ False

ACTIVITY 2-1: Inserting Data into a Table

1. **True or False? For books where the development cost is unavailable, you need to specify the column names and the corresponding values. For the other books, you need to specify only the values that you want to insert.**
   - ☑ True
   - ☐ False

4. **True or False? To verify whether you have successfully entered the records into the Titles table, you need to query the table to retrieve all records that have part numbers between 39906 and 39911.**
   - ☑ True
   - ☐ False

ACTIVITY 2-2: Modifying Data

1. **Identify the correct logic you should use to set the development cost to $21,000 for titles with part numbers 39906, 39907, and 39908.**
   - ○ Update the value of the devcost column for all records in the Titles table.
   - ○ Update the value of the devcost column for six books that were recently purchased.
   - ○ Create a query to generate a list of all part numbers. Update the devcost column value for part numbers that start with the value 399.
   - ◉ Update the value of the devcost column for the records of books with part numbers 39906, 39907, and 39908.

ACTIVITY 2-3: Deleting Data from a Table

1. **True or False? To delete the record for the title "History of the Internet", you need to specify the part number of the book as a value for the search condition of the DELETE statement.**
   - ☑ True
   - ☐ False

ACTIVITY 3-2: Planning a Table with Constraints

1. **Which is the best method to avoid duplication of customer information?**
   - ○ Provide a message on the website that requests customers who have already registered to avoid registering again.
   - ○ Prevent users with identical names from registering with the website.
   - ○ If users with identical information register, add a prefix to the name of each user to differentiate between the users.
   - ◉ Uniquely identify each user by defining the custnum column as the primary key for the table.

2. **When defining the columns of the Potential_customers1 table, which option represents the best method to ensure that a user provides all the required information when registering with the website?**
   - ○ Display a message on the website requesting users to fill all possible fields on the registration form.
   - ○ Specify simple names for the table columns.
   - ○ Reduce the number of columns in the Potential_customers1 table as compared to the Customers table.
   - [✓ Use the NOT NULL keywords when defining the column values for the Potential_customers1 table.]

3. **True or False? To ensure that potential customers enter valid cell phone numbers into the Potential_customers1 table, you need to add a CHECK constraint to the cellno column to check whether the length of the number specified is greater than or equal to 10.**
   - ☑ True
   - ☐ False

ACTIVITY 3-5: Adding and Dropping Constraints

4. **True or False? To ensure that the quantity value of books stored in a record of the Sales table is always above zero, add a CHECK constraint on the qty column of the Sales table to check whether the value stored in the column is greater than zero.**
   - ☑ True
   - ☐ False

ACTIVITY 4-1: Creating Views

1. **True or False? To implement the medium price option in the EasySearch software, you need to create a view that retrieves the part number, book title, and sale price of books in the Titles table that are priced between $20 and $40.**
   - ☑ True
   - ☐ False

3. **To generate a list of the top 20 books based on sales volume, you need to create a view that displays the part number, book title, and total sales quantities of the top 20 books based on the sales volume. Which is the correct logic you should use to group and sort records to display the required output?**
   - ○ Group and sort the records based on the total sales quantity of each book. Display the records in descending order.
   - [✓ Group records based on the part number of the Sales table and the book title of the Titles table. Sort the output based on the total quantity value sold for each book and display the records in descending order.]
   - ○ Group the records based on the book title. Sort the records based on the part number of each book and display the records in descending order.
   - ○ Group the records based on the part number of each book. Sort the records based on the book title and display the records in descending order.

ACTIVITY 4-3: Manipulating Data in Views

1. True or False? You could use the mediumprice view to insert, update, and delete records from the Titles table.
   - ☑ True
   - ☐ False

5. True or False? The mediumprice view displays records retrieved from the Titles table. Although you deleted the record with part number 40234 from the mediumprice view, the record still exists in the Titles table.
   - ☐ True
   - ☑ False

ACTIVITY 4-5: Modifying and Deleting Views

1. True or False? To retrieve information on the top three salespeople, you need to change the number specified after the TOP keyword in the SELECT statement of the slsperf view from 10 to 3.
   - ☑ True
   - ☐ False

ACTIVITY 5-1: Creating Indexes

1. You have defined primary keys for the Sales and Titles tables. Considering this fact, which logic can you use to improve the performance of queries that retrieve records from the Sales and Titles tables?
   - ○ Limit the number of records stored in the Sales and Titles tables.
   - ○ Create clustered indexes on each of the frequently retrieved columns of the Sales and Titles tables.
   - ◉ Create nonclustered indexes on frequently retrieved columns of the Sales and Titles tables.
   - ○ Create unique indexes on frequently retrieved columns of the Sales and Titles tables.

ACTIVITY 6-1: Creating a Transaction

1. True or False? You have created the statements to insert three records. To implement the work you have done and save it, you would place the statements within a BEGIN TRAN block and save your changes using a COMMIT TRAN statement.
   - ☑ True
   - ☐ False

4. True or False? The records inserted before the meeting also contain incorrect information. You can roll back the transaction slperInsert by using the ROLLBACK TRAN statement.

   ☐ True

   ☑ False

Glossary

alias
A reference used for a table or a column; generally used to refer to a column when another column of the same name exists in another table.

ALTER TABLE statement
A SQL statement that modifies the structure of a table.

ALTER VIEW statement
A SQL statement that modifies a view.

APPLY operator
A relational operator that enables you to apply a table expression to all rows of an outer table.

CHECK constraint
A constraint that validates the values stored in a column based on a condition you specify with the constraint.

COMMIT TRAN statement
A SQL statement that saves the data modifications by SQL statements in a transaction.

composite key
A primary key containing multiple columns.

constraint
A validation mechanism you implement on a column or a table to ensure data integrity.

correlated subquery
A SQL statement that SQL Server executes simultaneously with the outer query. Unlike other subqueries, the correlated subquery depends on the input from the outer query to return values.

CREATE INDEX statement
A SQL statement that creates an index on one or more table columns.

CREATE TABLE statement
A SQL statement you use to create a table.

CREATE VIEW statement
A SQL statement that creates views in SQL.

data integrity
The state of the information in a database where all values stored in the database are correct.

DEFAULT constraint
A constraint that specifies a default value for a column.

DELETE statement
A SQL statement that deletes records from a table.

DROP INDEX statement
A SQL statement that deletes an index created on a table.

DROP TABLE statement
A SQL statement that deletes a table from a database.

DROP VIEW statement
A SQL statement that deletes a view from a database.

EXISTS operator
A logical operator that you use in the condition of an outer query to check for the existence of records returned by the subquery.

foreign key
A column that links the records in a table with the primary key of another table.

FOREIGN KEY constraint
A constraint that creates a foreign key on a table.

GROUP BY clause
A clause you can use to group rows in the output based on the content of one or more columns.

HAVING clause
A clause to specify a search condition based on an aggregate value.

IN operator
A logical operator that you use in the search condition of a query to search for a value in a list of values.

index
A data organization mechanism that helps in the speedy retrieval of records.

inner query
A SQL statement that is contained within an outer query. You use an inner query when the value needed by the outer query's condition is unknown.

INSERT statement
A statement to insert a record into a table.

MERGE statement
A SQL statement that you use to merge the data in two tables or columns.

modified comparison operator
A comparison operator whose function you modify by combining it with the ANY or ALL logical operator.

nested subquery
A SQL statement that contains multiple levels of inner queries.

outer query
A query that depends on the values returned by another query used to display records.

OUTPUT clause
A SQL statement used to get information from rows that are influenced by the INSERT, UPDATE, and DELETE statements.

primary key
A column or a combination of columns that store values to uniquely identify each record in a table. You cannot store null values in the primary key for a table.

PRIMARY KEY constraint
A constraint that creates a primary key on a column or a combination of columns in a table. SQL Server does not permit null values in a primary key column or columns.

ROLLBACK TRAN statement
A SQL statement that cancels the modification of data implemented by the SQL statements in a transaction.

SCHEMABINDING
An option you can use in the `CREATE VIEW` SQL statement when you create a view to prevent changes to the table on which the view is based.

SELECT INTO statement
A SQL statement that creates a backup of a table's structure and data into a new table.

subquery
A query that is contained within an outer query. You use a subquery when the value needed by the outer query's condition is unknown.

TOP keyword
A keyword that you use in a `SELECT` statement to retrieve a number or percentage of rows from a table.

transaction
A collection of SQL statements that SQL executes as a single unit.

TRUNCATE TABLE statement
A SQL statement that deletes all records in a table but retains the table structure.

UNIQUE constraint
A constraint that implements uniqueness in the values stored in a column. You can store null values in columns to which you assign this constraint.

UPDATE statement
A SQL statement that modifies the data in a column of a table.

view
A virtual table that retrieves and displays records from a base table.

Index

094006S rev 1.0
ISBN-13 978-1-4246-2234-4
ISBN-10 1-4246-2234-4
90000
9 781424 622344