MCTS 70-433

Microsoft SQL Server 2008

Database Development

Reference by Simon Flüeli

Inhaltsverzeichnis

1 Chapter 1 Data Retrieval

TODO

2 Chapter2 Modifying Data – The Insert, Update, Delete and Merge Statements

2.1 Lesson 1: Modifying data by using INSERT, UPDATE, and DELETE Statements

2.1.1 Keywords

Keyword	Explanation
DML	Data Manipulation Language
Merge	Enhance the ability to perform INSERT, UPDATE and DELETE statements on a table based on the results of a query
Transaction	Transactions are key to providing a consistent reliable view of the data to all users at all time
Insert	Use the INSERT statement to add new rows to a table
Update	Use the UPDATE statement to modify rows in a table
Delete	Use the DELETE statement to remove rows from a table

2.1.2 Hints

Hint	Explanation
SET IDENTITY_INSERT ON	Overcome the limitation of an identity column when executing an insert or update statement

2.1.3 Exam objectives in this Chapter

- Modify data by using INSERT, UPDATE and DELETE statements
- Return data by using the OUTPUT clause
- Modify data by using MERGE statements
- Manage transactions

2.1.4 Inserting Data

Goal: Insert new rows to a database

Syntax

```
[ WITH <common_table_expression> [ , ...n] ]
INSERT
      [ TOP( expression ) [PERCENT] ]
      [ INTO ]
      { <object> | rowset_function_limited
           [ WITH (<Table_Hint_Limited> [...n])]
      }
{
      [(column_list)]
      [<OUTPUT Clause>]
      {VALUES({DEFAULT | NULL | expression} [,...n])[,...n]
      | derived_table
      | execute_statement
      | <dml_table_source>
      | DEFAULT VALUES
      }
}
[;]
```

- Include data for all columns or a partial list of columns
- When inserting data for all columns, it's not necessary to specify the column names
- Insert rows based on data selected from a different table by using the INSERT…SELECT statement

2.1.5 Updating data

Goal: Update rows in a database

Syntax

```
[ WITH <common_table_expression> [ , ...n] ]
UPDATE
      [ TOP( expression ) [PERCENT] ]
      { <object> | rowset_function_limited
            [ WITH (<Table_Hint_Limited> [...n])]
      }
SET
            { column_name = {expression | DEFAULT | NULL}
                  | {udt_column_name.{{property_name = expression
                                          | field_name = expression}
                                          | method_name (argument
[,...n])
                                    }
                  }
                  | column_name {.WRITE(expression, @Offset, @Length)}
                  | @variable = expression
                  | @variable = column = expression
                  | column_name {+= | -= | *= | /= | %= | &= | ^= | |=}
expression
                  | @variable {+= | -= | *= | /= | %= | &= | ^= | |=}
expression
                  | @variable = column {+= | -= | *= | /= | %= | &= | ^= | |=}
expression
            } [,...n]
      [<OUTPUT Clause>]
            [FROM{<table_source>}[,...n]]
            [WHERE {<search_condition>
                        | {[CURRENT OF
                              {{[GLOBAL] cursor_name}
                                    | cursor_variable_name
                              }
                        ]
                  }
            }
      ]
      [OPTION(<query_hint> [,...n])]
[;]
```

Best Practices
Build the logic of an UPDATE or DELETE statement as a SELECT statement and verify its logic before modifying it to run as an UPDATE or DELETE statement.

2.1.6 Deleting Data

Goal: Removing rows from a database

Syntax

```
[WITH <common_table_expression> [,...n]]
DELETE
      [TOP (expression) [PERCENT]]
      [FROM]
      {<object> | rowset_function_limited
            [WITH (<table_hint_limited> [...n])]
      }
      [<OUTPUT Clause>]
      [FROM <table_sources>[,...n]]
      [WHERE {<search_condition>
                  | {[CURRENT OF
                              {{[GLOBAL] cursor_name}
                                    | cursor_variable_name
                              }
                        ]
                  }
            }
      ]
      [OPTION(<Query Hint>[,...n])]
[;]
```

- The DELETE command can remove rows in one table base on information returned from a joined table.

2.1.7 The TRUNCATE TABLE statement

Like a DELETE statement without a WHERE clause, the TRUNCATE TABLE statement can also be used to remove all data from a table.

Differences

- The DELETE statement logs information on each row deleted, while the TRUNCATE TABLE statement only creates entries for the deallocation of the data pages
- The TRUNCATE STATEMENT executes more quickly and requires fewer resources because of the minimal logging
- If an identity column exists in the table, the TRUNCATE TABLE command resets the identity seed value

Syntax

```
TRUNCATE TABLE <tablename>
```

Neither the DELETE statement without a WHERE clause nor the TRUNCATE TABLE statement affect the schema structure of the table or related objects.

2.1.8 Lesson Summary

- The INSERT statement allows you to add new rows to a table
- The UPDATE statement allows you to make changes to the existing data in a table
It allows you not only to modify the value in a column, it also allows you to add or remove a value from a single column in the table without affecting the rest of the row being modified
- The DELETE statement allows you to remove one or more rows from a table

2.2 Lesson 2: Enhancing DML functionality with the OUTPUT clause and MERGE statements

You can use the OUTPUT clause and MERGE statement to enhance the functionality provided by DML statements.

The OUTPUT clause allows you to return information from rows affected by an INSERT, UPDATE or DELETE statement.

The MERGE statement provides you with the ability to perform an INSERT, UPDATE or DELETE operation on a target table based on a set of rules that are determined by a row comparison between the target table and a source table.

2.2.1 Using the OUTPUT clause

Keywords

Keyword	Explanation
OUTPUT	The OUTPUT clause gives you the ability to access the inserted and deleted tables that in versions previous to SQL Server 2005 were accessible only through triggers ➔ Functionality that was previously performed through triggers can be handled by stored procedures instead

Syntax

```
<OUTPUT_CLAUSE> ::=
{
      [OUTPUT <dml_select_list> INTO {@table_variable | output_variable}
[(column_list)]]
      [OUTPUT <dml_select_list>]
}
<dml_select_list> ::=
{<column_name | scalar_expression}[[AS] column_alias_identifier]
      [,...n]

<column_name> ::=
{DELETED | INSERTED | from_table_name}.{* | column_name}
      | $action
```

OUTPUT clause samples

Additional Infos:

```
USE Northwind

CREATE TABLE Audit
(AuditID int Primary Key IDENTITY
  , InsertedDate DateTime2 DEFAULT getdate()
  , InsertedID int);


DECLARE @InsertedID int ;

INSERT INTO Employees
(LastName, FirstName, Title)
OUTPUT getdate(), inserted.EmployeeID INTO Audit
VALUES ('Ralls', 'Kim', 'Support Rep');

SELECT * from Audit;

SELECT * FROM Employees;
```

	AuditID	InsertedDate	InsertedID
1	1	2010-03-07 17:05:25.5430000	10

	EmployeeID	LastName	FirstName	Title	TitleOfCourtesy	BirthDate	HireDate	Address
1	1	Davolio	Nancy	Sales Representative	Ms.	1948-12-08 00:00:00.000	1992-05-01 00:00:00.000	507 - 20th Ave. E. Apt. 2A
2	2	Fuller	Andrew	Vice President, Sales	Dr.	1952-02-19 00:00:00.000	1992-08-14 00:00:00.000	908 W. Capital Way
3	3	Leverling	Janet	Sales Representative	Ms.	1963-08-30 00:00:00.000	1992-04-01 00:00:00.000	722 Moss Bay Blvd.
4	4	Peacock	Margaret	Sales Representative	Mrs.	1937-09-19 00:00:00.000	1993-05-03 00:00:00.000	4110 Old Redmond Rd.
5	5	Buchanan	Steven	Sales Manager	Mr.	1955-03-04 00:00:00.000	1993-10-17 00:00:00.000	14 Garrett Hill
6	6	Suyama	Michael	Sales Representative	Mr.	1963-07-02 00:00:00.000	1993-10-17 00:00:00.000	Coventry House Miner Rd.
7	7	King	Robert	Sales Representative	Mr.	1960-05-29 00:00:00.000	1994-01-02 00:00:00.000	Edgeham Hollow Winchester Way
8	8	Callahan	Laura	Inside Sales Coordinator	Ms.	1958-01-09 00:00:00.000	1994-03-05 00:00:00.000	4726 - 11th Ave. N.E.
9	9	Dodswo...	Anne	Sales Representative	Ms.	1966-01-27 00:00:00.000	1994-11-15 00:00:00.000	7 Houndstooth Rd.
10	10	Ralls	Kim	Support Rep	NULL	NULL	NULL	NULL

Archive:

```
DELETE FROM [Order Details]
OUTPUT deleted.* INTO OrderDetailsArchive
FROM Orders
INNER JOIN [Order Details]
      ON Orders.OrderID = [Order Details].OrderID
WHERE OrderDate < '12-01-1997'
```

2.2.2 Using the MERGE statement

Keywords

Keyword	Explanation
CDC	Change data capture

The MERGE statement, along with CDC, which were both introduces in SQL Server 2008, greatly enhance the functionality for data warehouses and staging databases.

The MERGE statement gives you the ability to compare rows in a source and destination table.

Syntax

```
[WITH <common_table_expression> [,...n]]
MERGE
      [TOP(expression)[PERCENT]]
      [INTO] target_table [WITH (<merge_hint>)][[AS] table_alias]
      USING <table_source>
      ON <merge_search_condition>
      [WHEN MATCHED [AND <clause_search_condition>]
            THEN <merge_matched>]
      [WHEN NOT MATCHED [BY TARGET] [AND <clause_search_condition>]
            THEN <merge_not_matched>]
      [WHEN NOT MATCHED BY SOURCE [AND <clause_search_condition>]
            THEN <merge_matched>]
      [<output_clause>]
      [OPTION(<query_hint> [,...n])]
```

The following options can be defined as part of the *MERGE* statement syntax:

- ***[INTO] <target_table>*** Defines the table or view where the rows returned by the *WHEN* clauses will be inserted, updated, or deleted. This table or view is also used to match data against rows in the *<table_source>* based on the *<clause_search_condition>*. If the *<target_table>* is a view, all conditions for updating a view must be met for the *MERGE* statement to succeed.
- ***[AS] table_alias*** Defines an alias that can be used to minimize typing or make a command more readable by shortening table names referenced multiple times within the command.
- ***USING <table_source>*** Defines the table, view, or expression from which the rows that are matched to the target table come.
- ***ON <merge_search_condition>*** Specifies the conditions that should be used to define whether the rows in the two tables match. Similar to the *ON* clause in a *JOIN* operation, this could simply be `<table1_id> = <table2_id>`.

- ***WHEN MATCHED THEN <merge_matched>*** Defines the action to be performed on the rows in the target table where a match exists between the source and target rows based on the *ON <merge_search_condition>* clause and any additional conditions specified as part of the *WHEN MATCHED THEN <merge_matched>* clause. For an *UPDATE* to succeed, the source row must match only one target row. A single *MERGE* statement can have up to two *WHEN MATCHED* clauses joined by an *AND* operator. When two *WHEN MATCHED* clauses are defined, one must perform an *UPDATE* and one a *DELETE*. In addition, the second *WHEN MATCHED* clause is performed only where the first is not. For example, assume the target table includes only rows for products that are in stock. It does not include rows for products with 0 or negative inventory. The source table includes stock inventory changes and additions and subtractions from stock (represented as positive and negative integers). The first *WHEN MATCHED* clause includes a condition of adding the quantities of matched rows and then verifying if they are greater than 0. If yes, the quantity in the target table is set to the sum of the source quantity and the target quantity. The second *WHEN MATCHED* clause has a condition of the sum being less than or equal to 0 and if that is so, it deletes the row from the target table.
- ***WHEN NOT MATCHED [BY TARGET] THEN <merge_not_matched>*** Specifies that a row be inserted into the target table if a matched row is not found and if any additional conditions defined in the *WHEN NOT MATCHED [BY TARGET]* clause. Only one *WHEN NOT MATCHED [BY TARGET]* clause may exist in a *MERGE* statement.
- ***WHEN NOT MATCHED BY SOURCE THEN <merge_matched>*** Defines an *UPDATE* or *DELETE* action on rows that exist in the target table but not in the source. Like the *WHEN MATCHED* clause, a *MERGE* statement can include up to two *WHEN NOT MATCHED BY SOURCE* clauses, and when two exist, one must be defined as an *UPDATE* and the other is defined as a *DELETE*. When no rows are returned by the source table, columns in the source table cannot be referenced in the *<merge_matched>* clause or an error 207 (invalid column name) is returned.

2.2.3 MERGE statement samples

Move data from one table to another

```
MERGE INTO Sales.SalesOrderDetailHistory AS SODH
      USING Sales.SalesOrderDetail AS SOD
            ON SODH.salesorderid = SOD.salesorderid
            AND SODH.SalesOrderDetailID = SOD.SalesOrderDetailID
      WHEN NOT MATCHED BY TARGET THEN
            INSERT (Linetotal, SalesOrderID, SalesOrderDetailID,
                        CarrierTrackingNumber, OrderQty,
                        ProductID, SpecialOfferID, UnitPrice,
                        UnitPriceDiscount, rowguid,
                        ModifiedDate, Cancelled)
            VALUES(Linetotal, SalesOrderID, SalesOrderDetailID,
                        CarrierTrackingNumber, OrderQty,
                        ProductID, SpecialOfferID, UnitPrice,
                        UnitPriceDiscount, rowguid,
                        ModifiedDate, DEFAULT)
      WHEN NOT MATCHED BY SOURCE THEN
            UPDATE SET SODH.Cancelled = 'True'
```

2.3 Lesson 3: Managing Transactions

2.3.1 Definition

Transactions are frequently defined as a set of actions that succeed or fail as a whole.

Four major functions

Keyword	Description
Atomicity	When two or more pieces of information are involved in a transaction, either all the pieces are committed or none of them are committed.
Consistency	At the end of a transaction, either a new and valid form of the data exists or the data is returned to its original state. Returning data to its original state is part of the rollback functionality provided by SQL Server transactions.
Isolation	During a transaction (before it is committed or rolled back), the data must remain in an isolated state and not be accessible to other transactions. In SQL Server, the isolation level can be controlled for each transaction.
Durability	After a transaction is committed, the final state of the data is still available even if the server fails or is restarted. This functionality is provided through checkpoints and the database recovery process performed at startup in SQL Server.

Implicit transactions

A transaction starts automatically when any of the following commands are executed:
ALTER TABLE, CREATE, DELETE, DENY, DROP, FETCH, GRANT, INSERT, OPEN, REVOKE, SELECT, TRUNCATE TABLE, or UPDATE.

The transaction is active until you manually issue a COMMIT or ROLLBACK statement.

Enable implicit transactions: SET IMPLICIT_TRANSACTIONS ON

2.3.2 Defining explicit transactions

- Typically defined within stored procedures
- Started when a BEGIN TRANSACTION statement is executed
- Completed by issuing either a COMMIT TRANSACTION or ROLLBACK TRANSACTION statement

Note

Although the ROLLBACK Statement returns the data to its prior state, some functionalities, such as seed values for identity columns, are not reset.

When transactions are nested (verschachtelt) a ROLLBACK statement rolls back to the outermost nested transactions.
If you want to roll back only a portion of a transaction, you can define savepoints by using the SAVE TRANSACTION savepoint_name Statement and then referencing the savepoint name in the ROLLBACK Statement. By doing this, you are telling the Database Engine to roll back data changes only to the point where you issued the SAVE TRANSACTION statement with the same name.

2.3.3 Understanding locking

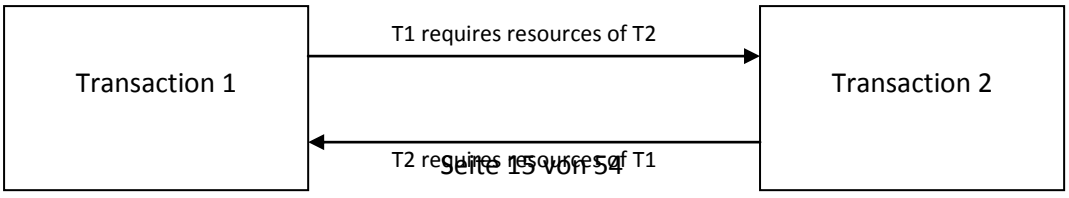
- Locks are used to prevent problems caused by multiple users accessing the same data at the same time or that a certain level of temporary inconsistency is acceptable for concurrent reads during an update, and no read locks are issued so that better concurrency and faster performance can be achieved

Locking modes

Keyword	Description
Shared (S)	Placed on resources for read (SELECT) operations. Shared locks are compatible with other shared locks. Shared locks are not compatible with exclusive locks. When the Isolation level is set to REPEATABLE READ or higher, or a locking hint is used, the shared locks are retained for the duration of the transaction. Otherwise, shared locks are released as soon as the read is completed.
Update (U)	Placed on resources where a shared (S) lock is required, but the need to upgrade to an exclusive (X) lock is anticipated. Only one transaction at a time can obtain an update lock on a resource. When modification to the resource is required, the update lock is upgraded to an exclusive lock.
Exclusive (X)	Placed on resources for data modification. An exclusive lock is not compatible with any other type of lock. Only the NOLOCK hint or the READ UNCOMMITTED isolation level overrides an exclusive lock's functionality.
Intent (IS, IX, SIX)	Placed on resources to improve performance and locking efficiency by placing intent (IS, IX, SIX) locks at a high-level object (such as a table) before placing shared (S) or exclusive (X) locks at a lower level (such as the page level).
Schema (Sch-M, Sch-S)	Schema modification (Sch-M) locks are placed on objects during schema modification operations, such as adding a new column to a table. Schema stability (Sch-S) locks are placed on objects while queries are being compiled or executed. Sch-M locks block all other operations until the lock is released. Sch-S locks are not compatible with Sch-M locks.
Bulk Update (BU)	Placed on tables for bulk insert. These locks allow multiple bulk insert threads to access the table but do not allow other processes to access the table. These locks are enabled by either using the TABLOCK hint or by using the sp_tableoption stored procedure to enable the Table lock on bulk load table option.
Key-range	Placed on a range of rows to protect against phantom insertions and deletions in a record set that is being accessed by a transaction. These locks are used by transactions using the SERIALIZABLE transaction isolation level.

2.3.4 Deadlocks

In a deadlock situation, two transactions are holding resources that each of the two transactions requires before completion.



You can use the following best practices to reduce deadlock situations and blocking issues
- Keep transactions short
- Collect and verify input data from users before opening a transaction
- Access resources in the same order whenever possible within transactions
- Keep transactions in a single batch
- Where appropriate, use a lower isolation level or row versioning-based isolation level
- Access the least amount of data possible in the transaction

2.3.5 Understanding reports on lock status

Options for viewing lock status

Option	Info
SQL Profiler	
System Monitor	
sys.dm_tran_locks	Provides detailed if normation about each lock that is currently being held on the instance
Activity Monitor in SSMS	

Resources

Keyword	Description
Row Identifier (RID)	A row identifier used to define a lock on a singele row located in a heap
KEY	The range of keys in an index used to define a lock on key ranges
PAGE	An 8-kilobyte (KB) page from tables or indexes
EXTENT	A group of eight contiguous pages within a table or index
HoBT	A heap or a balanced tree (B-tree) index
TABLE	An entire table, made up of both data and index pages
FILE	An entire database file
APPLICATION	An application-specified resource
METADATA	Used for metadata locks
ALLOCATION_UNIT	A single allocation unit
DATABASE	An entire database, including all data files

Heap: Storage method for a table without a clustered index

2.3.6 Using DBCC Log

Returns information about the information contained in the current transaction log

Code

```
DBCC LOG(<databasename>, <output identifier>)
```

Output identifier

Key	Description
0	Returns minimal information, including the current Log Sequence Number (LSN), operation, context, transaction ID and log block generation
1	Returns all the information from the previous level, as well as flags and record length information
2	Returns all the information from the previous level, as well as the object name, index name, page ID and slot ID
3	Returns a full set of information about the operation
4	Returns a full set of information about the operation, as well as a hex dump oft he current transaction log row

2.3.7 Setting transaction isolation levels

Syntax

```
SET TRANSACTION ISOLATION LEVEL
      { READ UNCOMMITTED
      | READ COMMITTED
      | REPEATABLE READ
      | SNAPSHOT
      | SERIALIZABLE
      }
[;]
```

Isolation Levels

Keyword	Description
READ UNCOMMITTED	Allows statements to read rows that were updated by a transaction before the rows are committed to the database. This isolation level minimizes contention but allows dirty reads and nonrepeatable (phantom) reads
READ COMMITTED	Allows statements within the current connection and transaction to experience nonrepeatable (phantom) reads but prevents dirty reads (data updated by another connection's open transaction). This is the default setting for SQL Server 2008
REPEATABLE READ	Does not allow transactions to read noncommitted modified data (dirty reads) and ensures that shared locks are maintained until the current transaction is completed
SNAPSHOT	Requires the ALLOW_SNAPSHOT_ISOLATION database option tob e set to ON. The SNAPSHOT isolation level takes a snapshot of the data at the time the data is read into the transaction but does not hold locks on the data. Updates can occur on the data from other transactions, but the current transaction does not see those updates reflected in subsequent reads of the original data. If the current transaction modifies data, those modifications are visible only to the current transaction
SERIALIZABLE	Does not allow data to be read that has been modified but not committed by other transactions. In addition, no other transactions can update data that has been read by the current transaction until the current transactions is complete. The SERIALIZABLE isolation level protects against phantom reads but causes the highest level of blocking and contention

2.3.8 Summary

- A transaction is a set of actions that make up an atomic unit of work and must succeed or fail as a whole

- By default, implicit transactions are not enabled. When implicit transactions are enabled, a number of statements automatically begin a transaction. The developer must execute a COMMIT or ROLLBACK statement to complete the transaction

- Explicit transactions start with a BEGIN TRANSACTION statement and are completed by either a ROLLBACK TRANSACTION or COMMIT TRANSACTION statement

- Issuing a ROLLBACK command when transactions are nested rolls back all transactions to the outermost BEGIN TRANSACTION statement, regardless of previously issued COMMIT statements for nested transactions

- SQL Server uses a variety of lock modes, including shared (S), exclusive (X), and intent (IS, IX, SIX) to manage data consistency while multiple transactions are being processed concurrently
- SQL Server 2008 supports the READ UNCOMMITTED, READ COMMITTED, REPEATABLE, READ, SNAPSHOT, and SERIALIZABLE isolation levels

2.4 Chapter summary

- DML statements such as INSERT, UPDATE and DELETE allow you to handle the data storage and retrieval requirements of your organization
- The MERGE statement and OUTPUT clause allow you to handle the data storage and retrieval requirements of your organization

- The MERGE statement and OUTPUT clause allow you to increase the functionality of you OLTP database environment as well as data warehouse and reporting environments. These options, in addition to CDC, provide a means to compare rows and set the UPDATE, INSERT, or DELETE logic based on those comparisons

- Transactions and locks provide the means by which many users can access and update data concurrently on a server running SQL Server while receiving a consistent view of the data

3 Chapter 3 Tables, data types, and declarative data integrity

3.1 Lesson 1: Working with tables and data types

3.1.1 Data Types

- Sql Server system data types
- User-defined types (UDTs) or SQL Common Language Runtime (SQLCLR) types

3.1.2 String data types

Data Type	Comments
char	fixed length
varchar	variable length
nchar	fixed length
nvarchar	variable length
text	deprecated (avoid using it)
ntext	deprecated (avoid using it)

3.1.3 Exact numeric types

Data Type	Storage size	Possible values	Comments
tinyint	1 byte	0 to 255	Equal to the byte data type in most programming languages, cannot store negative values
smallint	2 bytes	-32768 to 32767	A signed 16-bit integer
int	4 bytes	-2'147'483,648 to 2'147'483'647	A signed 32-bit integer
bigint	8 bytes	-2E63 to 2E63-1	A signed 64-bit integer
decimal (precision, scale)	5 to 17 bytes depending on precision	-10E38 + 1 to 10E38-1	A decimal number containing up to 38 digits
numeric (precision, scale)	Functionally equivalent to the decimal data type		

The precision defines the total number of digits that the data type holds, supporting a maximum precision of 38.
The scale defines how many of the digits defined by the precision are used as decimals.

3.1.4 Decimal storage requirements

Precision	Storage
1 to 9	5 bytes
10 to 19	9 bytes
20 to 28	13 bytes
29 to 38	17 bytes

3.1.5 Handling date and time

Data Type	Storage size	Possible values	Comments
datetime	8 bytes	January 1, 1753, through December 31, 9999, with time accuracy down to every third millisecond	Mainly available for backwards compatibility. Use datetime2, date, time or datetimeoffset whenever possible
smalldatetime	4 bytes	January 1, 1900, through June 6, 2079, with time accuracy down to every minute	Mainly available for backwards compatibility. Use datetime2, date, time or datetimeoffset whenever possible
datetime2 (fractional seconds precision)	Between 6 and 8 bytes	January 1, 0001, through December 31, 9999, with time accuracy down to the specified fractional seconds precision	Use when both date and time are required and time zone offset is not required
datetimeoffset (fractional seconds precision)	Between 8 and 10 bytes	January 1, 0001, through December 31, 9999, with time accuracy down to the specified fractional seconds precision and time zone offset between -14:00 and +14:00	Use when date, time, and time zone offset are required
date	3 bytes	January1, 0001, through December 31, 9999	Use when only a date is required
time (fractional seconds precision)	Between 3 and 5 bytes	00:00:00 to 23:59:59, with accuracy down to the specified fraction of a second	Use when only a time is required

3.1.6 Table basics

- Permanent tables
- Local temporary tables
- Global temporary tables
- Table variables

3.1.7 Creating a table

Before you can create a table, you need a schema in which to create the table.

Example

```
CREATE SCHEMA Sales
GO

CREATE TABLE Sales.Customers
(
      CustomerId INT NOT NULL,
      Name NVARCHAR(50) NOT NULL
)
```

Syntax of the CREATE TABLE statement

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

3.1.8 Table and column Names (Identifiers)

Standard identifiers

The first character must be a letter or an underscore (_), not a digit.

Exception

The first character can also be an at sign (@) or a number sign (#), but both of these have special meanings, as follows:

@ defines a variable or parameter

Note
@@ doesn't mean anything other than @, and it should not be used because many system functions begin with @@

defines a temporary object (that is, the object is available only from the current connection)

defines a global temporary object (that is, the object is available from any connection in the same instance)

Subsequent characters can include letters, digits, the at sign (@), the dollar sign ($), the number sign (#), and the underscore (_).

The identifier must not be a T-SQL reserved word.

Embedded spaces or special characters are not allowed.

3.1.9 Choosing data types

- Always use the data type that requires the least amount of disk space
- Do not use a data type if there is a chance that it will not cover your application's future needs
- Use variable-length data type, such as nvarchar, rather than a fixed-length data type, such as nchar
- Use fixed-length data type if the column's value is updated frequently
- Avoid using the datetime and smalldatetime data type (use the new date, time, datetime2 data type)
- Use varchar(max), nvarchar(max) and varbinary(max) instead of text, ntext and image data type
- Use rowversion instead of timestamp
- Only use varchar(max), nvarchar(max), varbinary(max) and xml if a data type with a specified size cannot be used.
- Use the float or real data types only if the precision provided by decimal is insufficient

3.1.10 Identity

All tables should have one column or a combination of columns that uniquely identifies rows in the table. This is called the primary key.

Syntax

```
<column name><data type> IDENTITY(<seed>, <increment>) NOT NULL
```

Seed: Starting point for generating numbers
Increment: Value by which the key is incremented (or decremented, if negative)

Identity columns cannot allow NULL values.

An IDENTITY property can be specified only when creating a new column.
An existing column cannot be modified to use the IDENTITY property.

3.1.11 Compression

Data compression is implemented in two levels: row and page.

Configure a table to use page-level compression

```
ALTER TABLE HR.Employees
      REBUILD
      WITH(DATA_COMPRESSION = PAGE)
```

If you turn on row-level compression, SQL Server changes the format used to store rows.
In simple terms, this row format converts all data types to variable-length data types.
It also uses no storage space to store NULL values.

Page-level compression includes row-level compression and adds page-level compression using page dictionary and column prefixing. Page dictionary simply introduces pointers between rows in the same page to avoid storing redundant data.

Example:

Row 01: John Kane
Row 02: John Woods
Row 03: John Kane

If this page used page dictionary, it would look like this:

Row 01: John Kane
Row 02: John Woods
Row 03: 01

3.1.12 Lesson summary

Creating tables is about defining columns, choose the right data type and to implement data integrity.

Data integrity needs to be a part of your table definition from the beginning to make sure that you protect your data from faults.

3.2 Lesson 2: Declarative Data Ingegrity

3.2.1 Validating Data

Two ways to validate data integrity
- declarative data integrity (set of rules on a table (also called: constraints)
- procedural data integrity (let procedure / trigger validate data)

Note
Don't use roles because they'll be removed from SQL Server in a future release

3.2.2 Implementing declarative data integrity

- Implemented by using constraints

Constraint types

Name	Description
Primary key	Identify a column or combination of columns that uniquely identifies a row in a table
Unique	Identify a column or combination of columns that uniquely identifies a row in a table
Foreign key	Identify a column or combination of columns whose values must exists in another column […] in the same table or another table - Usually greatly benefits from being indexed
Check	Set of rules that must be validated prior to data being allowed into a table Advantages - Simple to implement (similar to a WHERE clause) - Checked automatically - Improve performance Example (Product must have a non-negative price): ALTER TABLE Products ADD CHECK(Price >= 0.0) Disadvantages - Error messages cannot be replaced by more user-friendly error messages - A check constraint cannot "see" the previous value of a column
Default	Defines the default value for a column if no data is provided

3.2.3 Extending check constraints with User-Defined Functions

The expression in a check constraint can contain most of the logic that you can use in a WHERE clause (including NOT, AND, and OR). It can call scalar UDFs but it is not allowed to contain subqueries directly.

3.3 Chapter Summary

- Always consider which data types you are using because changing your mind later can be more difficult than you think
- Consider using user-defined data types to simplify selecting the correct data type when creating tables and to avoid data type mismatches in your database
- Having appropriate names, as defined in a naming guidelines document for objects and columns, is very important to make sure that the naming in your database is consistent
- consider compressing large tables to save disk space and memory, as well as possibly increasing performance
- Implement constraints to verify data integrity
- Implement constraints to support the optimizer
- Consider using UDFs in check constraints to implement advanced data integrity

4 Chapter 4 Using additional query techniques

4.1 Lesson 1: Building recursive queries with CTEs

CTEs (Common table expressions) allow you to iterate across a result set to solve one of the more difficult challenges within TSQL, efficiently executing a recursive query.

4.1.1 Common Table Expressions

A CTE is defined with two parts
- A WITH clause containing a SELECT statement that generates a valid table
- An outer SELECT statement that references the table expression

Example

```
WITH EmpTitle AS
(SELECT JobTitle, COUNT(*) AS numtitles
 FROM HumanResources.Employee
 GROUP BY JobTitle)
SELECT b.BusinessEntityID, b.JobTitle, a.numtitles
FROM EmpTitle AS a
INNER JOIN HumanResources.Employee AS b
      ON a.JobTitle = b.JobTitle
```

A recursive CTE expands the definition of the table expression and consists of two parts
- An anchor query, which is the source of the recursion, along with a UNION ALL statement and a second query, which recourses across the anchor query
- An outer query, which references the routine and specify the number of recursion levels

Example

```
DECLARE @EmployeeToGetOrgFor INTEGER = 126;

WITH EMP_cte(BusinessEntityID, OrganizationNode, FirstName, LastName,
            JobTitle, RecursionLevel)
AS (SELECT e.BusinessEntityID, e.OrganizationNode, p.FirstName,
            p.LastName, e.JobTitle, 0
      FROM HumanResources.Employee AS e
      INNER JOIN Person.Person AS p
            ON p.BusinessEntityID = e.BusinessEntityID
      WHERE e.BusinessEntityID = @EmployeeToGetOrgFor
      UNION ALL
      SELECT e.BusinessEntityID, e.OrganizationNode, p.FirstName,
            p.LastName, e.JobTitle, RecursionLevel + 1
      FROM HumanResources.Employee AS e
      INNER JOIN EMP_cte
            ON e.OrganizationNode = EMP_cte.OrganizationNode.GetAncestor(1)
      INNER JOIN Person.Person AS p
            ON p.BusinessEntityID = e.BusinessEntityID)

SELECT EMP_cte.RecursionLevel, EMP_cte.BusinessEntityID,
            EMP_cte.FirstName, EMP_cte.LastName,
            EMP_cte.OrganizationNode.ToString() AS OrganizationNode,
            p.FirstName AS 'ManagerFirstName',
            p.LastName AS 'ManagerLastName'
FROM EMP_cte
INNER JOIN HumanResources.Employee AS e
      ON EMP_cte.OrganizationNode.GetAncestor(1) = e.OrganizationNode
INNER JOIN Person.Person AS p
      ON p.BusinessEntityID = e.BusinessEntityID
ORDER BY RecursionLevel, EMP_cte.OrganizationNode.ToString()
OPTION(MAXRECURSION 25);
```

	RecursionLevel	BusinessEntityID	FirstName	LastName	OrganizationNode	ManagerFirstName	ManagerLastName
1	0	126	Jimmy	Bischoff	/3/1/12/5/	Pilar	Ackerman
2	1	121	Pilar	Ackerman	/3/1/12/	Peter	Krebs
3	2	26	Peter	Krebs	/3/1/	James	Hamilton
4	3	25	James	Hamilton	/3/	Ken	Sánchez

4.1.2 Caution: Recursion Levels

If the iterative query does not reach the bottom of the hierarchy by the time the MAXRECURSION values has been exhausted, you receive an error message.

4.1.3 Quick Check

1. What are the two parts of a CTE?
   A CTE has a WITH clause that contains a SELECT statement, which defines a table, along with an outer SELECT statement, which references the CTE
2. What are the two parts of a recursive CTE?
   A recursive CTE has an anchor query, which is the source of the recursion, along with a UNION ALL
   statement and a second query, which recourses across the anchor query; and an outer query, which
   references the CTE and specifies the maximum recursion levels

4.1.4 Lesson Summary

- A recursive CTE contains two SELECT statements within the WITH clause, separated by the UNION ALL keyword. The first query defines the anchor for the recursion, and the second query defines the data set that is to be iterated across
- If a CTE is contained within a batch, all statements preceding the WITH clause must be terminated with a semicolon
- The outer query references the CTE and specifies the maximum recursion

4.2 Lesson 2: Implementing subqueries

Subqueries allow you to:
- Nest one query within another to build complex routines
- Retrieve data sets that would be impossible to construct without resorting to a multistep process

4.2.1 Noncorrelated Subqueries

- Independent of the outer query within which it is contained
- Allow you to write more dynamic code

Example

```
SELECT *
FROM Customers.Customer AS a
INNER JOIN Customer.CustomerAdress AS b
      ON a.CustomerID = b.CustomerID
WHERE b.City IN (SELECT c.City FROM Customer.CityRegion AS c
                        INNER JOIN Customer.Region AS d
                             ON c.RegionID = d.RegionID
                        WHERE d.Region = 'RegionX')
```

4.2.2 Correlated subqueries

- Depends upon and references columns from the outer query
- The inner query depends upon the values from the outer query

Example

```
SELECT a.ProductID, a.ListPrice
FROM Production.Product AS a
WHERE EXISTS(SELECT 1 FROM Sales.SalesOrderDetail AS b
                    WHERE b.ProductID = a.ProductID)
```

4.2.3 Quick Check

1. What is the difference between a correlated and a noncorrelated subquery?
   A noncorrelated subquery is a query that is embedded within another query but does not reference any columns from the outer query. A correlated subquery is embedded within another query and references columns within the outer query
2. What is a derived table?
   A derived table is a SELECT statement that is embedded within a FROM clause

4.2.4 Lesson summary

- Noncorrelated subqueries are independent queries that are embedded within an outer query and are used to retrieve a scalar value or list of values that can be consumed by the outer query to make code more dynamic
- Correlated subqueries are queries that are embedded within an outer query but reference values within the outer query

4.3 Lesson 3: Applying ranking functions

- Used to provide simple analytics (statistical ordering / segmentation)

4.3.1 Ranking Data

Ranking function	Description
ROW_NUMBER	Assigns a number from 1 to n based on a user-specified sorting order
RANK	Assigns the same value to each row that is tied and then skips to the next value, leaving a gap in the sequence corresponding to the number of rows that were tied
DENSE_RANK	Assigns the same value to each duplicate but does not produce gaps in the sequence
NTILE	Used to divide a result set into approximately equal groups

Example ROW_NUMBER

```
SELECT p.FirstName, p.LastName,
            ROW_NUMBER() OVER (ORDER BY SalesYTD DESC) AS 'RowNumber',
            s.SalesYTD, s.TerritoryID
FROM Sales.SalesPerson AS s
INNER JOIN Person.Person AS p
      ON s.BusinessEntityID = p.BusinessEntityID
INNER JOIN Person.BusinessEntityAddress AS ba
      ON p.BusinessEntityID = ba.BusinessEntityID
INNER JOIN Person.Address AS a
      ON a.AddressID = ba.AddressID
WHERE s.TerritoryID IS NOT NULL
```

Example RANK

```
SELECT a.ProductID, b.Name, a.LocationID, a.Quantity,
RANK() OVER(PARTITION BY a.LocationID ORDER BY a.Quantity DESC) AS 'Rank'
FROM Production.ProductInventory AS a
INNER JOIN Production.Product AS b
      ON a.ProductID = b.ProductID
ORDER BY 'Rank'
```

Example DENSE_RANK

```
SELECT a.ProductID, b.Name, a.LocationID, a.Quantity,
DENSE_RANK() OVER(PARTITION BY a.LocationID ORDER BY a.Quantity DESC) AS
'Rank'
FROM Production.ProductInventory AS a
INNER JOIN Production.Product AS b
      ON a.ProductID = b.ProductID
ORDER BY 'Rank'
```

Example NTILE

```
SELECT p.FirstName, p.LastName,
            NTILE(4) OVER(ORDER BY s.SalesYTD DESC) AS QuarterGroup,
            s.SalesYTD, a.PostalCode
FROM Sales.SalesPerson AS s
INNER JOIN Person.Person AS p
      ON s.BusinessEntityID = p.BusinessEntityID
INNER JOIN Person.BusinessEntityAddress AS ba
```

4.3.2 Quick Check

1. What is the difference between RANK and DENSE_RANK?
   RANK assigns the same number to ties but leaves gap in the sequence corresponding to the number of rows that were tied. DENSE_RANK assigns the same number to ties but does not create a gap in a sequence
2. When do ROW_NUMBER, RANK, AND DENSE_RANK produce the same results?
   ROW_NUMBER, RANK, and DENSE_RANK produce the same results when the column being sorted by does not contain any duplicate values within the result set

4.3.3 Lesson summary

- ROW_NUMBER is used to number rows sequentially in a result set but might not produce identical results if there are ties in the column(s) used for sorting
- RANK numbers a tie with identical values but can produce gaps in a sequence
- DENSE_RANK numbers ties with identical values but does not produce gaps in the sequence
- NTILE allows you to divide a result set into approximately equal-sized groups

4.4 Chapter summary

- Recursive CTEs can be used to solve a variety of problems that require traversal of a hierarchy more efficiently than cursor-based approaches
- Subqueries allow you to embed one query within another query. A noncorrelated subquery is independent of the outer query, whereas a correlated subquery references columns in the outer query
- Ranking functions can be used to solve a variety of problems that require ordering of a result set, such as pagination and finding gaps within a sequence

5 Chapter 5 Programming Microsoft SQL Server with T-SQL User-Defined Stored Procedures, Functions, Triggers, and Views

5.1 Lesson 1 Stored procedures

5.1.1 Creating stored procedures

Almost any command within the T-SQL language can be included in a stored procedure.

The only commands that cannot be used in a stored procedure are the following:

Do NOT use in store procedures
USE <database name>
SET SHOWPLAN_TEXT
SET SHOWPLAN_ALL
SET PARSEONLY
SET SHOWPLAN_XML
CREATE AGGREGATE
CREATE RULE
CREATE DEFAULT
CREATE SCHEMA
CREATE FUNCTION or ALTER FUNCTION
CREATE TRIGGER or ALTER TRIGGER
CREATE PROCEDURE or ALTER PROCEDURE
CREATE VIEW or ALTER VIEW

Syntax

```
CREATE {PROC | PROCEDURE } [schema_name.]procedure_name [; number]
      [{@parameter [type_schema_name.]data_type }
            [VARYING][= default] [OUT |OUTPUT][READONLY]
      ][,...n]
[WITH <procedure_option>[,...n]]
[FOR REPLICATION]
AS {<sql_statement> [;][...n] | <method_specifier>}[;]
<procedure_option> ::=
      [ENCRYPTION] [RECOMPILE] [EXECUTE AS CLAUSE]
```

5.1.2 Variables, Parameters, and Return Codes

- Objects, that are designed to pass values within your code

Variables
- Way to manipulate, store, and pass data within a stored procedure / between stored procedures and functions
- Two types of variables: local and global
    - local variable is designated by a single at sign (@)
    - global variable is designated by a double at sign (@@)

Global variables

Global variable	Definition
@@ERROR	Error code from the last statement executed
@@IDENTITY	Value of the last identity value inserted within the connection
@@ROWCOUNT	The number of rows affected by the last statement
@@TRANCOUNT	The number of open transactions within the connection
@@VERSION	The version of SQL Server

Parameters
- Local variables that are used to pass values into a stored procedure
- Two types of parameters: input and output

5.1.3 Controls flow constructs

Stored procedures have several control flow constructs that can be used

Control flow constructs
RETURN
IF…ELSE
BEGIN…END
WHILE
BREAK/CONTINUE
WAITFOR
GOTO

5.1.4 Error Messages

- Three components
    - Error number (number between 1 and 49999)
    - Severity level (number between 0 and 25)
    - Error Message(up to 255 Unicode characters long)
        - Own messages (number higher than 50001)

5.1.5 Error Handling

The way a TRY…CATCH is implemented
- If an error with a severity less than 20 is encountered within the TRY block, control passes to the corresponding CATCH block
- If an error is encountered in the CATCH block, the transaction is aborted and the error is returned to the calling application unless the CATCH block is nested within another TRY block
- The CATCH block must immediately follow the TRY block
- Within the CATCH block, you can commit or roll back the current transaction unless the transaction is in an uncommitable state
- A RAISERROR executed in the TRY block immediately passes control to the CATCH block without returning an error message to the application
- A RAISERROR executed in the CATCH block closes the transaction and returns control to the calling application with the specified error message
- If a RAISEERROR is not executed within the CATCH block, the calling application never receives an error message

Within the CATCH block, you have access to the following functions

Function	Description
ERROR_NUMBER()	The error number of the error thrown
ERROR_MESSAGE()	The text of the error message
ERROR_SEVERITY()	The severity level of the error message
ERROR_STATE()	The state of the error
ERROR_PROCEDURE()	The function, trigger, or procedure name that was executing when the error occurred
ERROR_LINE()	The line of code within the function, trigger, or procedure that caused the error

5.1.6 Cursors

- Used when you need to process data one row at a time
- Allows scrolling forward as well as backward through the result set

Cursor components

Component	Description
DECLARE	Used to define the SELECT statement that is the basis for the rows in the cursor
OPEN	Causes the SELECT statement to be executed and load the rows into a memory structure
FETCH	Used to retrieve one row at a time from the cursor
CLOSE	Used to close the processing on the cursor
DEALLOCATE	Used to remove the cursor and release the memory structures containing the cursor result set (It is not necessary to close and deallocate the cursors in a procedure)

Syntax

```
DECLARE cursor_name CURSOR [LOCAL | GLOBAL]
      [FORWARD_ONLY | SCROLL]
      [STATIC | KEYSET |DYNAMIC | FAST_FORWARD]
      [READ_ONLY | SCROLL_LOCKS |OPTIMISTIC]
      [TYPE_WARNING]
      FOR select_statement
      [FOR UPDATE [OF column_name [,...n]]]
```

SET-BASED PROCESSING

- Multilevel cursors in stored procedures -> probably replace the cursors with a set-based process

Types of cursors

Type	Description
FAST_FORWARD	- Fastest performing - Only move forward one row at a time - Scrolling not supported - Same as FORWARD_ONLY, READ_ONLY - Default option for cursors
STATIC	- Retrieved and stored in a temp table - Fetches go against the temp table and modifications to the underlying tables for the cursor are not visible - Supports scrolling - Modifications are not allowed
KEYSET	- Set of keys that uniquely identify each row in the cursor result set is stored in a temp table - Non-key columns are retrieved from underlying tables - Modifications to rows are reflected as the cursor is scrolled - Inserts into underlying table are not accessible for the cursor

	- If you attempt to access a row that has been deleted, @@FETCH_STATUS returns -2
DYNAMIC	- Most expensive cursor to use - Reflects all changes made to the underlying result set - Position and order of rows can change each time a fetch is made - FETCH ABSOLUTE is not available for dynamic cursor

If a cursor is declared using the SCROLL option, the FETCH statement has several options

Fetch options	Description
FETCH FIRST	Fetches the first row in the result set
FETCH LAST	Fetches the last row in the result set
FETCH NEXT	Fetches the next row in the result set
FETCH PRIOR	Fetches the row in the result set just before the current position of the cursor pointer
FETCH ABSOLUTE n	Fetches the *n*th row from the beginning of the result set
FETCH RELATIVE n	Fetches the *n*th row forward in the cursor result set from the current position of the cursor pointer
READ_ONLY	Cursor cannot be updated
SCROLLS_LOCKS	A lock is acquired as each row is read into the cursor, guaranteeing that any transaction executed against the cursor succeeds
OPTIMISTIC	Lock is not acquired. SQL Server uses either a timestamp or a calculated checksum in the event that a timestamp column does not exist to detect if the data has changed since being read into the cursor. If the data has changed, the modification fails

5.1.7 Quick Check

1. What are the four types of cursors that you can create?
   The four types of cursers that can be created are FAST_FORWARD, STATIC, KEYSET, and DYNAMIC
2. How does XACT_ABORT behave within a TRY block?
   If XACT_ABORT is set within a TRY block, when an error is thrown, control passes to the CATCH block. However, the transaction is doomed and cannot be committed within the CATCH block.

5.1.8 Lesson summary

- A stored procedure is a batch of T-SQL code that is given a name and is stored within a database
- You can pass parameters to a stored procedure either by name or by position. You can also return data from a stored procedure using output parameters
- You can use the EXECUTE AS clause to cause a stored procedure to execute under a specific security context
- Cursors allow you to process data on a row by row basis; however, if you are making the same modification to every row within a cursor, a set-oriented approach is more efficient
- A TRY…CATCH block delivers structured error handling to your procedure

5.2 Lesson 2: User-Defined Functions

- Programmable objects, used to perform calculations
- Can access data and return results
- Cannot make any modifications

5.2.1 System functions

Function category	Description
Aggregate	Combine multiple values (SUM, AVG, COUNT)
Configuration	Return system configuration information (@@VERSION, @@LANGUAGE, @@SERVERNAME)
Cryptographic	Support encryption and decryption
Cursor	Return state information about a cursor (@@FETCH_STATUS, @@CURSOR_ROWS)
Date and time	Return portions of a date/time or calculate dates and times (DATEADD, DATEPART, DATEDIFF, GETDATE)
Management	Return information to manage portions of SQL Server (sys.dm_db_index_physical_stats)
Mathematical	Perform mathematical operations (SIN, COS, TAN, LOG, PI, ROUND)
Metadata	Return information about database objects (OBJECT_NAME, OBJECT_ID, DATABASEPROBERTYEX, DB_NAME)
Ranking	Return values used in ranking result sets
Rowset	Return a result set that can be joined to other tables (CONTAINS, FREETEXT)
Security	Return security information about users and roles (SUSER_SNAME, Has_perms_by_name, USER_NAME)
String	Manipulate CHAR and VARCHAR data (POS, CHARINDEX, SOUNDEX, REPLACE, STUFF, RTRIM)
System	Return information about a variety of system, database, and object settings as well as data (DATALENGTH, HOST_NAME, ISDATE, ISNULL, SCOPE_IDENTITY, CAST, CONVERT)
System statistics	Return operational information about a SQL Server instance (fn_virtualfilestats, @@CONNECTIONS)
Text and image	Manipulate text and image data (TEXTPTR, TEXTVALID) *(Text and image data types have been deprecated and you should not use either of these functions in applications)*

5.2.2 User-defined functions

- Perform an action that changes the state of an instance or database

- Modify data in a table
- Call a function that has an external effect, such as the RAND function
- Create or access temporary tables
- Execute code dynamically

Syntax

```
CREATE FUNCTION [schema_name.]function_name
([{@parameter_name [AS] [type_schema_name.]parameter_data_type
      [= default][READONLY]} [,...n]])
RETURNS return_data_type
      [WITH <function_option> [,...n]]
      [AS]
      BEGIN
            <function_body>
            RETURN scalar_expression
      END
```

5.2.3 Schemabinding

The SCHEMABINDING option is applied to ensure that you can't drop dependent objects. For example, if you were to create a function that performed a SELECT against the Sales.SalesOrderHeader table did not exist. To prevent objects that a programmable object relies on from being dropped or altered, you specify the SCHEMABINDING option. If you attempt to drop or modify the dependent object, SQL Server prevents the change. To drop or alter a dependent object, you first have to drop the programmable object that depends on the object you want to drop or alter.

5.2.4 Retrieving data from a function

- Retrieve data from a function by using a SELECT statement
- Functions can be used in
    - A SELECT list
    - A WHERE clause
    - An expression
    - A CHECK or DEFAULT constraint
    - A FROM clause with the CROSS/OUTER APPLY function

Best Practices: Avoid using Functions in the WHERE clause

5.2.5 Quick Check

1. What are the three types of functions that you can create?
   You can create a scalar function, which returns a single value, an inline table-valued function, which contains a single SELECT statement and is treated the same as a view, and a multi-statement table-valued function, which returns a table
2. What are the required elements of a function?
   Every function ends with a RETURN statement. Scalar functions include the value to be returned immediately following the RETURN statement. Inline table-valued functions include the SELECT statement for the result set to return immediately following the RETURN statement. Multi-statement table-valued functions just terminate with a RETURN. With the exception of inline table-valued functions, the entire function body is required to be enclosed in a BEGIN…END block

5.2.6 Lesson summary

- You can create scalar functions, inline table-valued functions, and multi-statement table-valued functions
- With the exception of inline table-valued functions, the function body must be enclosed within a BEGIN…END block
- All functions must terminate with a RETURN statement
- Functions are not allowed to change the state of a database or of a SQL Server instance

5.3 Lesson 3: Triggers

- Special type of stored procedure
- Automatically execute when a DML or DDL statement is executed

5.3.1 DML Triggers

- Created on a table or a view
- Defined for a specific event (INSERT, UPDATE, or DELETE)
- Have access to the *inserted* and *deleted* tables

Syntax

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

- Regardless of the number of rows that are affected, a trigger fires only once for an action

5.3.2 DDL Triggers

- Executes either when a DDL statement is executed or when the user logs on to the SQL Server instance
- Has access to the *EVENTDATA* function

Syntax

```
CREATE TRIGGER trigger_name
ON {ALL SERVER | DATABASE }
[WITH <ddl_trigger_option> [,...n]]
{FOR | AFTER} {event_type | event_group} [,...n]
AS {sql_statement [;] [,...n] | EXTERNAL NAME <method specifier>[;]}

<ddl_trigger_option> ::=
      [ENCRYPTION] [EXECUTE AS Clause]

<method_specifier> ::=
      assembly_name.class_name.method_name
```

DDL Trigger Event Types

DDL Command	Event Type
CREATE DATABASE	CREATE_DATABASE
DROP LOGIN	DROP_LOGIN
UPDATE STATISTICS	UPDATE_STATISTICS
DROP TRIGGER	DROP_TRIGGER
ALTER TABLE	ALTER_TABLE

EVENTDATA Function
- Returns the following XML document

```
<EVENT_INSTANCE>
    <EventType>type</EventType>
    <PostTime>date-time</PostTime>
    <SPID>spid</SPID>
    <ServerName>name</ServerName>
    <LoginName>name</LoginName>
    <UserName>name</UserName>
    <DatabaseName>name</DatabaseName>
    <SchemaName>name</SchemaName>
    <ObjectName>name</ObjectName>
    <ObjectType>type</ObjectType>
    <TSQLCommand>command</TSQLCommand>
</EVENT_INSTANCE>
```

- Can be queried by using the *value()* method

Example

```
SELECT EVENTDATA().value
        ('(/EVENT_INSTANCE/DatabaseName)[1]','nvarchar(max)'),
       EVENTDATA().value
        ('(/EVENT_INSTANCE/SchemaName)[1]','nvarchar(max)'),
       EVENTDATA().value
        ('(/EVENT_INSTANCE/ObjectName)[1]','nvarchar(max)'),
       EVENTDATA().value
        ('(/EVENT_INSTANCE/TSQLCommand)[1]','nvarchar(max)')
```

5.3.3 Logon Triggers

- Fire at logon to the SQL Server instance
    - After authentication succeeds
    - Before the user session is actually established
- It is not possible to return any messages to a user
- Used to audit and restrict access

Syntax

```
CREATE TRIGGER trigger_name
ON ALL SERVER
[WITH <logon_trigger_option>[,...n]]
{FOR | AFTER} LOGON
AS {sql_statement [;] [,...n] | EXTERNAL NAME <method specifier> [;]}
<logon_trigger_option> ::= [ENCRYPTION] [EXECUTE AS Clause]
```

Quick Check

1. What are the three types of triggers that can be created?
   You can create DML, DDL, and logon triggers
2. You query the XML document returned by the EVENTDATA function within DDL and logon triggers to retrieve information about the event that caused the trigger to fire. Each event has a different XML schema

5.3.4 Lesson summary

- Triggers are specialized stored procedures that automatically execute in response to a DDL or DML event
- You can create three types of trigger: DML, DDL, and logon triggers
- A DML trigger executes when an INSERT, UPDATE, or DELETE statement for which the trigger is coded occurs
- A DDL trigger executes when a DDL statement for which the trigger is coded occurs
- A logon trigger executes when there is a logon attempt
- You can access the inserted and deleted tables within a DML trigger
- You can access the XML document provided by the EVENTDATA function within a DDL or logon trigger

5.4 Lesson 4 Views

5.4.1 Creating a view

- SELECT statement that has been given a name and is stored in a database

Syntax

```
CREATE VIEW [schema_name.]view_name [(column [,...n])]
[WITH <view_attribute> [,...]]
AS select_statement
[WITH CHECK OPTION][;]
```

- Select statement in a view cannot do any of the following
  - Contain the COMPUTE or COMPUTE BY clause
  - Create a permanent or temporary table by using the INTO keyword
  - Use an OPTION clause
  - Reference a temporary table
  - Reference any type of variable
  - Contain an ORDER BY clause unless a TOP operator is also specified

5.4.2 Modifying data through a view

- Following requirements have to be met
  - The data modification must reference exactly one table
  - Columns in the view must reference columns in a table directly
  - The column cannot be derived from an aggregate
  - The column cannot be computed as the result of a UNION / UNION ALL, CROSSJOIN, EXCEPT, or INTERSECT
  - The column being modified cannot be affected by the DISTINCT, GROUP BY, or HAVING clause
  - The TOP operator is not used

5.4.3 Partitioned Views

- Split large tables across multiple storage structures
- Bring all the data back together using a view
- Implements a UNION ALL of all member tables with the same structure

Conditions

- All columns of the member tables should be contained in the select list of the view
- Columns in the same ordinal position of each SELECT statement need to be of exactly the same data type and collation
- At least one column that corresponds to a CHECK constraint, unique to each member table, should be in the same ordinal position of each SELECT statement
- The constraints must form unique, non-overlapping data sets in each member table
- The same column cannot be used multiple times in the select list
- The partitioning column, defined by the CHECK constraint, must be part of the primary key
- The partitioning column cannot be computed, be a timestamp data type, have a DEFAULT constraint, or be an identity column
- The same member table cannot appear twice within the view definition
- Member tables cannot have indexes on computed columns
- The primary key of each table must have the same number of columns for each member table
- All member tables must have the same ANSI_PADDING setting

Split member tables of a partitioned view across SQL Server instances -> distributed partitioned view

5.4.4 Creating an Indexed View

- Can improve performance
- Data is already materialized
- Data does not have to be calculated on the fly

Requirements

- The SELECT statement cannot reference other views
- All functions must be deterministic
- AVG, MIN, MAX, STDEV, STDEVP, VAR, and VARP are not allowed
- The index created must be both clustered and unique
- ANSI_NULLS must have been set to ON when the view and any tables referenced by the view were created
- The view must be created with the SCHEMABINDING option
- The SELECT statement must not contain subqueries, outer joins, EXCEPT, INTERSECT, TOP, UNION, ORDER BY, DISTINCT, COMPUTE/COMPUTE BY, CROSS/OUTER APPLY, PIVOR, or UNPIVOT

5.4.5 Determinism

- Determinism
    - Function that returns the same value every time it is called, given the same input parameters
    - SUBSTRING

- Nondeterministic
    - Function that could return a different value each time it is called, given the same input params
    - RAND, GETDATE

5.4.6 Query substitution

- When a nonmaterialized view is referenced, SQL Server replaces the name of the view with the actual SELECT statement defined by the view, rewrites the query as if you had not reference the view at all, and then submits the rewritten query to the optimizer
- When an index is created against a view, the data is materialized
- Queries that reference the indexed view do not substitute the definition of the view but instead return the results directly from the indexed view

5.4.7 Quick check

1. What types of views can be created?
   You can create a regular view that is just a stored SELECT statement. You can also create a partitioned view that uses the UNION AL keywords to combine multiple member tables
2. What types of indexes can be created on a view?
   You can index a view by creating a unique, clustered index

5.4.8 Lesson summary

- A view is a name for a SELECT statement stored within a database
- A view has to return a single result set and cannot reference variables or temporary tables
- You can update data through a view so long as the data modification can be resolved to a specific set of rows in an underlying table
- If a view does not meet the requirements for allowing data modifications, you can create an INSTEAD OF trigger to process the data modification instead
- You can combine multiple tables that have been physically partitioned using a UNION ALL statement to create a partitioned view
- A distributed partitioned view uses linked servers to combine multiple member tables across SQL Server instances
- You can create a unique, clustered index on a view to materialize the result set for improved query performance

5.5 Chapter Summary

- SQL Server allows you to create four programmable objects: functions, stored procedures, triggers, and views
- Functions can return a scalar value or a result set but are not allowed to change the state of a database or SQL Server instance
- Stored procedures provide a programming API that abstracts the database structure from applications
- A stored procedure can contain almost any command within the T-SQL language
- Triggers are created for tables and views and automatically execute in response to an INSERT, UPDATE, or DELETE
- Views allow you to assign a name to a SELECT statement that produces a single result set and that is stored within a database

6 Chapter 6 Techniques to Improve Query Performance

6.1 Lesson 1 Tuning Queries

6.1.1 Evaluating Query Performance

- Three main metrics to consider
    - Query cost
    - Page reads
    - Query execution time

Query Cost
- Takes into account CPU and input/output (I/O) resources

Page reads
- Number of 8-kilobyte data pages accessed by the SQL Server storage engine while executing a query
- SET STATISTICS IO ON
- Logical reads: Number of pages read from memory
- Scan count: Number of passes though an index or heap it took to respond to the query
- LOB: How many page reads were used to retrieve Large Object (LOB) data

Query execution time
- Affected by blocking (locks) as well as resource contention on the server
- SET STATISTICS TIME ON

Examining the Theoretical Query Execution Order

Theoretical Execution Order – Excluding the UNION Clause

Clauses	Results
1. FROM, JOIN, APPLY, and ON	The join is executed and the first query filter (the ON clause) is applied
2. WHERE	The second query filter is applied
3. GROUP BY and aggregate functions (such as SUM, AVG, and so on) that are included in the query	Grouping and aggregation calculations are performed
4. HAVING	The third query filter (filtering of the results of aggregate functions) is applied
5. SELECT	Columns that should be returned by the query are selected
6. ORDER BY	Results are sorted
7. TOP	The fourth (and last) query filter is applied; this causes the query to return only the first X rows from the results thus far
8. FOR XML	The tabular result returned by the SELECT statement is converted to Extensible Markup Language (XML)

Theoretical Execution Order – Including the UNION Clause

Clauses	Results
1. FROM, JOIN, APPLY, and ON	The join is executed and the first query filter (the ON clause) is applied
2. WHERE	The second query filter is applied
3. GROUP BY and aggregate functions (such as SUM, AVG, and so on) that are included in the query	Grouping and aggregation calculations are performed
4. HAVING	The third query filter (filtering of the results of aggregate functions) is applied
5. TOP	The fourth (and last) query filter is applied; this causes the query to return only the first X rows from the results thus far
6. UNION and SELECT	The results of each SELECT statement included in the query are concatenated; columns that should be returned by the query are selected
7. ORDER BY	Results are sorted
8. FOR XML	The tabular result returned by the SELECT with UNION statement is converted to XML

6.1.2 Tuning Query Performance

- Rewriting the query
- De-normalizing tables
- Adding indexes
- Removing indexes

6.1.3 The Graphical execution Plan

ITEM TO WATCH FOR	POSSIBLE IMPLICATIONS
Thick arrows 	A thick arrow represents a large number of rows moving from one operation in the execution plan to another. The greater the number of rows transferred from one operation to another, the thicker the arrow.
Hash operations  Hash Match (Inner Join) Cost: 73 %	If a hash operation is used to handle clauses such as *GROUP BY* and *JOIN*, it often means that an appropriate index did not exist to optimize the query.
Sorts  Sort Cost: 69 %	A sort isn't necessarily bad, but if it is a high percentage of the query cost, you should consider whether an index can be built to remove the need for the sort operation.
Large plans	The plan with fewer operations is typically the better optimized plan.
Table or clustered index scans  Table Scan [Customers] [c1] Cost: 100 %  Clustered Index Scan (Clustered) [Customers].[Index1] [c1] Cost: 100 %	A clustered index scan and a table scan indicate that no appropriate index can be used to optimize the query.

6.1.4 Using Search Arguments

- Search argument (SARG)
    - Filter expression that is used to limit the number of rows returned by a query and that can use an index seek operation that substantially improves the performance of the query

6.1.5 Lesson Summary

- Understanding how queries are logically constructed is important to knowing that they correctly return the intended result
- Understanding how queries are logically constructed helps you understand what physical constructs (like indexes) help the query execute faster
- Make sure you understand your metrics when you measure performance

6.2 Lesson 2: Creating Indexes

Two types of indexes
- clustered indexes
  - Is the actual table; that is, the bottom level of a clustered index contains the actual rows, including all columns, of the table
  - Always cover queries
- nonclustered indexes
  - Contains only the columns included in the index's key, plus a pointer pointing to the actual data row

Table without clustered index -> heap (unsorted)
Table with clustered index -> sorted

6.2.1 Improving performance with covered indexes

- A covered index always performs better than a noncovered index
- For queries that return a very limited number of rows, a noncovered index also performs very well
- For the somewhat-selective query, the noncovered index reads more than 34 times more pages than the covered index. In this case, a query was considered selective by the optimizer when it matched less than roughly 0.77 percent of the table

6.2.2 Using included columns and reducing index depth

- Included columns cannot be used for tasks such as filtering or sorting
- Sole benefit is reducing page reads through covering queries by avoiding table lookups

- Only columns that are used for filtering, grouping, or sorting should be part of the index key
- All other columns -> included columns

- Retrieve information from the sys.dm_db_index_physical_stats

```
SELECT *
FROM sys.dm_db_index_physical_stats(
      DB_ID(),
      OBJECT_ID('Test.IncludedColumnsTest'),
      NULL,
      NULL,
      'DETAILED')
AS a
```

6.2.3 Using clustered indexes

- Reading from the clustered index never results in lookups
- Clustered index
  - Generally be defined on columns that are often queried and return a lot of data
  - Avoids the problem of lookups and fetching a large number of rows

6.2.4 Read performance vs. Write performance

- Index only helps boost the read performance
- Write performance is typically degraded
- 5 nonclustered indexes on a table -> 6 inserts (1 table, 5 indexes)
    - Same with delete statements
    - Not the same with update statements (only indexes that contain the columns that are updated)

6.2.5 Using indexed views

- Create indexes on a view
- Extensive requirements to create indexes on views
- View with indexes -> View is materialized -> still a view but it stores data found in the view
- Can greatly improve the read performance of queries

6.2.6 Partitioning

Partition Functions

- CREATE PARTITION FUNCTION
- LEFT / RIGHT

Example LEFT

```
CREATE PARTITION FUNCTION PF(INT)
AS RANGE LEFT
FOR VALUES (10, 20, 30)
```

Partitions available with the PF partition function

Partition number	Partition range
1	<= 10
2	> 10 AND <= 20
3	> 20 AND <= 30
4	> 30

Example RIGHT

```
CREATE PARTITION FUNCTION PF(INT)
AS RANGE RIGHT
FOR VALUES (10, 20, 30)
```

Partitions available with the PF partition function

Partition number	Partition range
1	< 10
2	>= 10 AND < 20
3	>= 20 AND < 30
4	>= 30

Partition Schemes
- CREATE PARTITION SCHEME
- Map between partitions for a particular partition and file groups
- Store different parts of a table on different types of storage devices

Example

```
CREATE PARTITION SCHEME PS
AS PARTITION PF TO (FG1, FG2, FG1, FG2)
```

Example partitioning

```
USE AdventureWorks;

CREATE PARTITION FUNCTION PFCustomerID (INT)
AS RANGE LEFT
FOR VALUES (5000, 10000, 15000);

CREATE PARTITION SCHEME PSCustomerID
AS PARTITION PFCustomerID ALL TO ([PRIMARY]);

CREATE TABLE Test.CustomersPartitioned (
    CustomerID INT IDENTITY NOT NULL
    ,AccountNumber VARCHAR(50) NOT NULL
    ,ModifiedDate DATETIME2 NOT NULL
) ON PSCustomerID (CustomerID);

CREATE NONCLUSTERED INDEX AccountNumberIdx
    ON Test.CustomersPartitioned (AccountNumber);

INSERT Test.CustomersPartitioned (AccountNumber, ModifiedDate)
    SELECT AccountNumber, ModifiedDate FROM Sales.Customer;

SELECT index_id, partition_number, rows FROM sys.partitions
    WHERE object_id = OBJECT_ID('Test.CustomersPartitioned')
    ORDER BY index_id, partition_number;
```

Here are the results of the query against the *sys.partitions* catalog view:

```
index_id    partition_number rows
----------- ---------------- --------------------
0           1                5000
0           2                5000
0           3                5000
0           4                4185
2           1                5000
2           2                5000
2           3                5000
2           4                4185
```

6.2.7 Tuning indexes automatically

- Graphical execution plan -> note, which lets you retrieve the script needed to create the missing index
- sys.dm_db_missing_index_details
- sys.dm_db_missing_index_groups
- sys.dm_db_missing_index_group_stats

6.2.8 Lesson summary

- Indexes typically help read performance but can hurt write performance
- Indexed views can increase performance even more than indexes, but they are restrictive and typically cannot be created for the entire query
- Deciding which columns to put in the index key and which should be implemented as included columns is important
- Analyze which indexes are actually being used and drop the ones that aren't. This saves storage space and minimizes the resources used to maintain indexes for write operations

6.3 Chapter summary

- Always evaluate different ways of implementing costly queries
- Because indexes take up space (if not disabled) and are maintained for write operations, try to drop unused indexes
- When measuring query performance, always include the query execution time as a metric; don't just rely on cost and page reads
- Create covered indexes for the most frequently executed queries
- Evaluate creating indexed views to cover entire queries or parts of queries

7 Chapter 7 Extending Microsoft SQL Server Functionality with XML, SQLCLR, and Filestream

7.1 Lesson 1 Working with XML

- Retrieving relational data as XML
  Instead of retrieving a tabular result set from the database, you retrieve an XML document
- Passing data as XML to the database
  Instead of passing scalar values to the database by issuing multiple data manipulation language (DML) statements or running a stored procedure multiple times, an XML document or fragment can be passed directly to the database
- Storing and querying an actual XML document or fragment in the database
  This is one of the more controversial topics. Why would you store XML directly in a table?

7.2 Retrieving tabular data as XML

- At the end of the SELECT statement: FOR XML <mode>
- Modes: RAW, AURO, EXPLICIT, and PATH
- Recommended and most powerful mode: PATH

7.3 FOR XML RAW

Example

```
SELECT c.CustomerID,
            c.AccountNumber
FROM Sales.Customer AS c
WHERE c.CustomerID IN (1, 2)
FOR XML RAW
```

```
SELECT c.CustomerID,
            c.AccountNumber
FROM Sales.Customer AS c
WHERE c.CustomerID IN (1, 2)
FOR XML RAW('Customer'), ROOT('Customers')
```

```
<row CustomerID="1" AccountNumber="AW000000001" />
<row CustomerID="2" AccountNumber="AW000000002" />
```

```
<Customers>
      <Customer Id="1" AccountNumber="AW000000001" />
      <Customer Id="2" AccountNumber="AW000000002" />
</Customers>
```