

Database Session Notes

Introduction to T-SQL Querying

About T-SQL

- Structured Query Language (SQL)
 - Developed by IBM in the 1970s
 - Adopted by ANSI and ISO standards bodies
 - Widely used in the industry
 - PL/SQL (Oracle), SQL Procedural Language (IBM), Transact-SQL (Microsoft)
- Transact-SQL is commonly referred to as T-SQL
 - The querying language of SQL Server 2016
- SQL is declarative
 - Describe what you want, not the individual steps

Database Session Notes

T-SQL is declarative

For example, if you want to retrieve a list of customers who are located in Portland, the pseudocode would be:

Procedural Approach

```
Loop through each row in the data.  
  If the city is Portland,  
    return the row;  
  otherwise  
    do nothing.  
  Move to next row.  
End loop.
```

Declarative Approach

```
Return all the customers whose city is Portland
```

Categories of T-SQL Statements

Data Manipulation Language (DML)

- Used to query and manipulate data
- SELECT
- INSERT
- UPDATE
- DELETE

Data Definition Language (DDL)

- Used to define database objects
- CREATE
- ALTER
- DROP

Data Control Language (DCL)

- Used to manage security permissions
- GRANT
- REVOKE
- DENY

*DML with SELECT is the focus of this course

Database Session Notes

SQL Server Data Types

SQL Server Data Types

- SQL Server associates columns, expressions, variables and parameters with data types
- Data types determine the kind of data that can be held in a column or variable
 - Integers, characters, dates, decimals, binary strings, and so on
- SQL Server supplies built-in data types
- Developers can also define custom data types

SQL Server Data Type Categories

Exact numeric	Unicode character strings
Approximate numeric	Binary strings
Date and time	Other (eg: xml, uniqueidentifier)
Character strings	

Database Session Notes

Numeric Data Types

- Exact Numeric Data Types

Data Type	Range	Storage (bytes)
tinyint	0 to 255	1
smallint	-32,768 to 32,767	2
int	2^{31} (-2,147,483,648) to $2^{31}-1$ (2,147,483,647)	4
bigint	-2^{63} - $2^{63}-1$ (+/- 9 quintillion)	8
bit	1, 0 or NULL	1
decimal/numeric	$-10^{38} + 1$ through $10^{38} - 1$ when maximum precision is used	5-17
money	-922,337,203,685,477.5808 to 922,337,203,685,477.5807	8
smallmoney	-214,748.3648 to 214,748.3647	4

Character Data Types

- SQL Server supports two kinds of character data as fixed-width or variable-width data:
 - Single-byte: **char** and **varchar**
 - One byte stored per character
 - Only 256 possible characters—limits language support
 - Multibyte: **nchar** and **nvarchar**
 - Multiple bytes stored per character (usually two bytes, but sometimes up to four)
 - More than 65,000 characters represented—multiple language support
 - Precede character string literals with N (National)
 - text** and **ntext** data types are deprecated, but may still be used in older systems
 - In new development, use **varchar(max)** and **nvarchar(max)** instead

Database Session Notes

Character Data Types

Data Type	Range	Storage
char(n)	1 – 8000 characters	<i>n</i> bytes padded
nchar(n)	1 – 4000 characters	2 * <i>n</i> bytes padded
varchar(n)	1 – 8000 characters	Actual length + 2 bytes
nvarchar(n)	1 – 4000 characters	Actual length + 2 bytes
varchar(MAX)	up to 2 GB	Actual length + 2 bytes
nvarchar(MAX)	up to 2 GB	Actual length + 2 bytes

- Single-byte character data is indicated with single quotation marks alone.
- Multi-byte character data is indicated by single quotation marks with the prefix N (for National). The N prefix is always required.

Character String Functions

- Common functions that modify character strings

Function	Syntax	Remarks
SUBSTRING	SUBSTRING (expression , start , length)	Returns part of an expression.
LEFT, RIGHT	LEFT (expression , integer_value) RIGHT (expression , integer_value)	LEFT returns left part of string up to integer_value. RIGHT returns right part of string up to integer value.
LEN, DATALENGTH	LEN (string_expression) DATALENGTH (expression)	LEN returns the number of characters in string_expression, excluding trailing spaces. DATALENGTH returns the number of bytes used.
CHARINDEX	CHARINDEX (expressionToFind, expressionToSearch)	Searches expressionToSearch for expressionToFind and returns its start position if found.
REPLACE	REPLACE (string_expression , string_pattern , string_replacement)	Replaces all occurrences of string_pattern in string_expression with string_replacement.
UPPER, LOWER	UPPER (character_expression) LOWER (character_expression)	UPPER converts all characters in a string to uppercase. LOWER converts all characters in a string to lowercase.

Database Session Notes

The LIKE Predicate

- The LIKE predicate can be used to check a character string for a match with a pattern
- Patterns are expressed with symbols
 - % (Percent) represents a string of any length
 - _ (Underscore) represents a single character
 - [<List of characters>] represents a single character within the supplied list
 - [<Character> - <character>] represents a single character within the specified range
 - [^<Character list or range>] represents a single character not in the specified list or range
 - ESCAPE Character allows you to search for characters that would otherwise be treated as part of a pattern - %, _, [, and])

```
SELECT categoryid, categoryname, description
FROM Production.Categories
WHERE description LIKE 'Sweet%';
```

Date and Time Data Types

- Older versions of SQL Server support only **datetime** and **smalldatetime** data types
- SQL Server 2008 introduced **date**, **time**, **datetime2** and **datetimeoffset** data types
- SQL Server 2012 added further functionality for working with date and time data types

Data Type	Storage (bytes)	Date Range (Gregorian Calendar)	Accuracy	Recommended Entry Format
datetime	8	January 1, 1753 to December 31, 9999	Rounded to increments of .000, .003, or .007 seconds	YYYYMMDD hh:mm:ss[.mmm]
smalldatetime	4	January 1, 1900 to June 6, 2079	1 minute	YYYYMMDD hh:mm:ss[.mmm]
datetime2 (ANSI)	6 to 8	January 1, 0001 to December 31, 9999	100 nanoseconds	YYYYMMDD hh:mm:ss[.nnnnnnn]
date (ANSI)	3	January 1, 0001 to December 31, 9999	1 day	YYYY-MM-DD
time (ANSI)	3 to 5	n/a – time only	100 nanoseconds	hh:mm:ss[.nnnnnnn]
datetimeoffset (ANSI)	8 to 10	January 1, 0001 to December 31, 9999	100 nanoseconds	YYYY-MM-DDThh:mm:ss[.nnnnnnn][{+ -}hh:mm]

Database Session Notes

Entering Date and Time Data Types Using Strings

- SQL Server doesn't offer a means to enter a date or time value as a literal value
 - Dates and times are entered as character literals and converted explicitly or implicitly
 - For example, **char** converted to **datetime** due to precedence
 - Formats are language-dependent, and can cause confusion
- Best practices:
 - Use character strings to express date and time values
 - Use language-neutral formats

```
SELECT orderid, custid, empid, orderdate
FROM Sales.Orders
WHERE orderdate = '20070825';
```

Working Separately with Date and Time

- **datetime**, **smalldatetime**, **datetime2**, and **datetimeoffset** include both date and time data
- If only date is specified, time set to midnight (all zeros)

```
DECLARE @DateOnly AS datetime2 = '20160112';
SELECT @DateOnly AS Result;
```

- If only time is specified, date set to base date (January 1, 1900)

```
DECLARE @time AS time = '12:34:56';
SELECT CAST(@time AS datetime2) AS Result;
```

Database Session Notes

Binary String Data Types

- Binary string data types

Data Type	Range	Storage (bytes)
binary(n)	1 to 8000 bytes	n bytes
varbinary(n)	1 to 8000 bytes	n bytes + 2
varbinary(max)	1 to 2.1 billion (approx.) bytes	n bytes + 2

- The **image** data type is also a binary string type but is marked for removal in a future version of SQL Server; **varbinary(max)** should be used instead

Querying Date and Time Values

- Date values converted from character literals often omit time
 - Queries written with equality operator for date will match midnight

```
SELECT orderid, custid, empid, orderdate
FROM Sales.Orders
WHERE orderdate = '20070825';
```

- If time values are stored, queries need to account for time past midnight on a date
 - Use range filters instead of equality

```
SELECT orderid, custid, empid, orderdate
FROM Sales.Orders
WHERE orderdate >= '20070825' AND orderdate < '20070826';
... WHERE orderdate BETWEEN '20070825' AND '20070826'
```


Database Session Notes

Other Data Types

Data Type	Range	Storage (bytes)	Remarks
xml	0-2 GB	0-2 GB	Stores XML in native hierarchical structure
uniqueidentifier	Auto-generated	16	Globally unique identifier (GUID)
hierarchyid	n/a	Depends on content	Represents position in a hierarchy
rowversion	Auto-generated	8	Previously called timestamp
geometry	0-2 GB	0-2 GB	Shape definitions in Euclidian geometry
geography	0-2 GB	0-2 GB	Shape definitions in round-earth geometry
sql_variant	0-8000 bytes	Depends on content	Can store data of various other data types in the same column
cursor	n/a	n/a	Not a storage datatype—used for cursor operations
table	n/a	n/a	Not a storage data type—used for query operations

Data Type Precedence

- Data type precedence determines which data type will be chosen when expressions of different types are combined
- By default, the data type with the lower precedence is converted to the data type with the higher precedence
- It is important to understand implicit conversions
 - Conversion to a data type of lower precedence must be made explicitly (using CAST or CONVERT functions)
- Example precedence (low to high)
 - CHAR -> VARCHAR -> NVARCHAR -> TINYINT -> INT -> DECIMAL -> TIME -> DATE -> DATETIME2 -> XML
- Not all combinations of data type have a conversion (implicit or explicit)

Database Session Notes

When are Data Types Converted?

- Data type conversion scenarios
 - When data is moved, compared to or combined with other data
 - During variable assignment
- Implicit conversion
 - When comparing data of one data type to another
 - Transparent to the user
- Explicit conversion
 - Uses CAST or CONVERT functions

```
WHERE <column of smallint type> = <value of int type>
```

```
CAST(unitprice AS INT)
```

T-SQL Language Elements

Database Session Notes

Set Theory and SQL Server

Characteristics of a Set	Example
Elements of a set called Members	Customer as a member of set called Customers
Elements of a set are described by attributes	First name, Last name, Age
Elements must be unique	Customer ID

Set theory does not specify the order of its members

Predicates and Operators

Elements:	Predicates and Operators:
Predicates	ALL, ANY, BETWEEN, IN, LIKE, OR, SOME
Comparison Operators	=, >, <, >=, <=, <>, !=, !>, !<
Logical Operators	AND, OR, NOT
Arithmetic Operators	*, /, %, +, -,
Concatenation	+

Database Session Notes

Variables

- Local variables in T-SQL temporarily store a value of a specific data type
- Name begins with single @ sign
 - @@ reserved for system functions
- Assigned a data type
- Must be declared and used within the same batch
- In SQL Server 2016, you can declare and initialize a variable in the same statement

```
DECLARE @search varchar(30) = 'Match%';
```

Comments

- Two methods for marking text as comments
 - A block comment, surround text with /* and */

```
/*  
    All the text in this paragraph will be treated as  
    comments by SQL Server.  
*/
```

- An inline comment, precede text with --

```
-- This is an inline comment
```

- Many T-SQL editors will color comments as above

Database Session Notes

Expressions

- Combination of identifiers, values, and operators evaluated to obtain a single result
- Can be used in SELECT statements
 - SELECT clause
 - WHERE clause
- Can be single constant, single-valued function, or variable
- Can be combined if expressions have the same data type

```
SELECT YEAR(orderdate) + 1 ...  
SELECT qty * unitprice ...
```

Functions

String Functions	Date and Time Functions	Aggregate Functions
<ul style="list-style-type: none">• SUBSTRING• LEFT, RIGHT• LEN, DATALENGTH• REPLACE• REPLICATE• UPPER, LOWER• LTRIM, RTRIM• STUFF• SOUNDEX	<ul style="list-style-type: none">• GETDATE• SYSDATETIME• GETUTCDATE• DATEADD• DATEDIFF• YEAR• MONTH• DAY• DATENAME• DATEPART• ISDATE	<ul style="list-style-type: none">• SUM• MIN• MAX• AVG• COUNT• COUNT_BIG• STDEV,• STDEVP• VAR

Database Session Notes

Control of Flow, Errors, and Transactions

Control of Flow

- IF ... ELSE
- WHILE
- BREAK
- CONTINUE
- BEGIN ... END
- WAITFOR

Error Handling

- TRY
- CATCH
- THROW

Transaction Control

- BEGIN TRANSACTION
- ROLLBACK TRANSACTION
- COMMIT TRANSACTION
- ROLLBACK WORK
- SAVE TRANSACTION

The above are used in programmatic code objects

Batch Separators

- Batches are sets of commands sent to SQL Server as a unit
- Batches determine variable scope, name resolution
- To separate statements into batches, use a separator:
 - SQL Server tools use the GO keyword
 - GO is not an SQL Server T-SQL command
 - GO [count] executes the preceding batch [count] times

Database Session Notes

SELECT Queries

Elements of the SELECT Statement

Clause	Expression
SELECT	<select list>
FROM	<table or view>
WHERE	<search condition>
GROUP BY	<group by list>
ORDER BY	<order by list>

Database Session Notes

Retrieving Columns from a Table or View

- Use SELECT with column list to show columns
- Use FROM to specify the source table or view
 - Specify both schema and object names
- Delimit names if necessary
- End all statements with a semicolon

Keyword	Expression
SELECT	<select list>
FROM	<table or view>

```
SELECT companyname, country  
FROM Sales.Customers;
```

Displaying Columns

- Displaying all columns
 - This is not best practice in production code!

```
SELECT *  
FROM Sales.Customers;
```

- Displaying only specified columns

```
SELECT companyname, country  
FROM Sales.Customers;
```


Database Session Notes

Using Calculations in the SELECT Clause

- Calculations are scalar, returning one value per row

Operator	Description
+	Add or concatenate
-	Subtract
*	Multiply
/	Divide
%	Modulo

- Using scalar expressions in the SELECT clause

```
SELECT unitprice, qty, (qty * unitprice)
FROM Sales.OrderDetails;
```

SQL Sets and Duplicate Rows

- SQL query results are not truly relational:
 - Rows are not guaranteed to be unique
 - No guaranteed order
- Even unique rows in a source table can return duplicate values for some columns

```
SELECT country
FROM Sales.Customers;
```

```
country
-----
Argentina
Argentina
Belgium
Austria
Austria
```

Database Session Notes

Understanding DISTINCT

- DISTINCT specifies that only unique rows can appear in the result set
- Removes duplicates based on column list results, not source table
- Provides uniqueness across set of selected columns
- Removes rows already operated on by WHERE, HAVING, and GROUP BY clauses
- Some queries may improve performance by filtering out duplicates before execution of SELECT clause

SELECT DISTINCT Syntax

```
SELECT DISTINCT <column list>  
FROM <table or view>
```

```
SELECT DISTINCT companyname, country  
FROM Sales.Customers;
```

companyname	country
-----	-----
Customer AHPOP	UK
Customer AHXHT	Mexico
Customer AZJED	Germany
Customer BSVAR	France
Customer CCFIZ	Poland

Database Session Notes

Use Aliases to Refer to Columns

- Column aliases using AS

```
SELECT orderid, unitprice, qty AS quantity  
FROM Sales.OrderDetails;
```

- Column aliases using =

```
SELECT orderid, unitprice, quantity = qty  
FROM Sales.OrderDetails;
```

```
SELECT orderid, unitprice, quantity, cost = quantity * unitprice  
FROM Sales.OrderDetails;
```

- Accidental column aliases

```
SELECT orderid, unitprice quantity  
FROM Sales.OrderDetails;
```

Use Aliases to Refer to Tables

- Create table aliases in the FROM clause
- Create table aliases with AS

```
SELECT custid, orderdate  
FROM SalesOrders AS SO;
```

- Create table aliases without AS

```
SELECT custid, orderdate  
FROM SalesOrders SO;
```

- Using table aliases in the SELECT clause

```
SELECT SO.custid, SO.orderdate  
FROM SalesOrders AS SO
```

Database Session Notes

Using CASE Expressions in SELECT Clauses

- T-SQL CASE expressions return a single (scalar) value
- CASE expressions may be used in:
 - SELECT column list
 - WHERE or HAVING clauses
 - ORDER BY clause
- CASE returns result of expression
 - Not a control-of-flow mechanism
- In SELECT clause, CASE behaves as calculated column requiring an alias

Forms of CASE Expressions

- Two forms of T-SQL CASE expressions:
- Simple CASE
 - Compares one value to a list of possible values
 - Returns first match
 - If no match, returns value found in optional ELSE clause
 - If no match and no ELSE, returns NULL
- Searched CASE
 - Evaluates a set of predicates, or logical expressions
 - Returns value found in THEN clause matching first expression that evaluates to TRUE

Database Session Notes

Question

You have the following SELECT query:

```
SELECT FirstName, LastName, Gender
FROM hr.Employees;
```

This returns:

FirstName	LastName	Gender
Maya	Steele	1
Adam	Brookes	0
Naomi	Sharp	1
Pedro	Fielder	0
Zachary	Parsons	0

How could you make these results clearer?

Question

Answer:

Use the following query:

```
SELECT FirstName, LastName, Gender =
CASE Gender
  WHEN 1 THEN 'Female'
  WHEN 0 THEN 'Male'
  ELSE 'Unspecified'
END
FROM hr.Employees;
```

Database Session Notes

Subqueries

Working with Subqueries

- Subqueries are nested queries: queries within queries
- Results of inner query passed to outer query
 - Inner query acts like an expression from perspective of outer query
- Subqueries can be self-contained or correlated
 - Self-contained subqueries have no dependency on outer query
 - Correlated subqueries depend on values from outer query
- Subqueries can be scalar, multi-valued, or table-valued

Database Session Notes

Comparing Self-Contained and Correlated Subqueries

Self-Contained Subquery:

Outer Query:

```
SELECT orderid, productid,  
unitprice, qty  
FROM Sales.OrderDetails  
WHERE orderid = ( )
```

Inner Query:

```
SELECT MAX(orderid)  
AS lastorder  
FROM Sales.Orders
```



Correlated Subquery:

Outer Query:

```
SELECT orderid, empid,  
orderdate  
FROM Sales.Orders AS O1  
WHERE  
orderdate = ( )
```

Inner Query:

```
SELECT MAX(orderdate)  
FROM Sales.Orders AS O2  
WHERE O2.empid =  
O1.empid
```



Writing Scalar Subqueries

- Scalar subquery returns single value to outer query
- Can be used anywhere single-valued expression is used: SELECT, WHERE, and so on

```
SELECT orderid, productid, unitprice, qty  
FROM Sales.OrderDetails  
WHERE orderid =  
  (SELECT MAX(orderid) AS lastorder  
   FROM Sales.Orders);
```

- If inner query returns an empty set, result is converted to NULL
- Construction of outer query determines whether inner query must return a single value

Database Session Notes

Writing Multi-Valued Subqueries

- Multi-valued subquery returns multiple values as a single column set to the outer query
- Used with IN predicate
- If any value in the subquery result matches IN predicate expression, the predicate returns TRUE

```
SELECT custid, orderid
FROM Sales.orders
WHERE custid IN (
    SELECT custid
    FROM Sales.Customers
    WHERE country = N'Mexico');
```

```
SELECT c.custid, o.orderid
FROM Sales.Customers AS c
JOIN Sales.Orders AS o
    ON c.custid = o.custid
WHERE c.country = 'Mexico';
```

- May also be expressed as a JOIN (test both for performance)

Writing Correlated Subqueries

- Write inner query to accept input value from outer query
- Write outer query to accept appropriate return result (scalar or multi-valued)
- Correlate queries by passing value from outer query to match argument in inner query

```
SELECT custid, orderid, orderdate
FROM Sales.Orders AS outerorders
WHERE orderdate =
    (SELECT MAX(orderdate)
     FROM Sales.Orders AS innerorders
     WHERE innerorders.custid = outerorders.custid)
ORDER BY custid;
```


Database Session Notes

Writing Queries Using EXISTS with Subqueries

- The keyword EXISTS does not follow a column name or other expression
- The SELECT list of a subquery introduced by EXISTS typically only uses an asterisk (*)

```
SELECT custid, companyname  
FROM Sales.Customers AS c  
WHERE EXISTS (  
    SELECT *  
    FROM Sales.Orders AS o  
    WHERE c.custid=o.custid);
```

```
SELECT custid, companyname  
FROM Sales.Customers AS c  
WHERE NOT EXISTS (  
    SELECT *  
    FROM Sales.Orders AS o  
    WHERE c.custid=o.custid);
```

Sorting Data

Database Session Notes

Using the ORDER BY Clause

- ORDER BY sorts rows in results for presentation purposes
 - No guaranteed order of rows without ORDER BY
 - Use of ORDER BY guarantees the sort order of the result
 - Last clause to be logically processed
 - Sorts all NULLs together
- ORDER BY can refer to:
 - Columns by name, alias or ordinal position (not recommended)
 - Columns not part of SELECT list
 - Unless DISTINCT specified
- Declare sort order with ASC or DESC

ORDER BY Clause Syntax

Writing ORDER BY using column names:

```
SELECT <select list>  
FROM <table source>  
ORDER BY <column1_name>, <column2_name>;
```

Writing ORDER BY using column aliases:

```
SELECT <column> AS <alias>  
FROM <table source>  
ORDER BY <alias1>, <alias2>;
```

- Specifying sort order in the ORDER BY clause:

```
SELECT <column> AS <alias>  
FROM <table source>  
ORDER BY <column_name|alias> ASC|DESC;
```

Database Session Notes

ORDER BY Clause Examples

- ORDER BY with column names:

```
SELECT orderid, custid, orderdate  
FROM Sales.Orders  
ORDER BY orderdate;
```

- ORDER BY with column alias:

```
SELECT orderid, custid, YEAR(orderdate) AS orderyear  
FROM Sales.Orders  
ORDER BY orderyear;
```

- ORDER BY with descending order:

```
SELECT orderid, custid, orderdate  
FROM Sales.Orders  
ORDER BY orderdate DESC;
```

Filtering Data

Database Session Notes

Filtering Data in the WHERE Clause with Predicates

- WHERE clauses use predicates
 - Must be expressed as logical conditions
 - Only rows for which predicate evaluates to TRUE are accepted
 - Values of FALSE or UNKNOWN filtered out
- WHERE clause follows FROM, precedes other clauses
 - Can't see aliases declared in SELECT clause
- Can be optimized by SQL Server to use indexes
- Data filtered server-side
 - Can reduce network traffic and client memory usage

WHERE Clause Syntax

- Filter rows for customers from Spain

```
SELECT contactname, country
FROM Sales.Customers
WHERE country = N'Spain';
```

- Filter rows for orders after July 1, 2007

```
SELECT orderid, orderdate
FROM Sales.Orders
WHERE orderdate > '20070701';
```

- Filter orders within a range of dates

```
SELECT orderid, custid, orderdate
FROM Sales.Orders
WHERE orderdate >= '20070101' AND orderdate < '20080101';
```

Database Session Notes

WHERE Clause Syntax

Predicates and Operators	Description
IN	Determines whether a specified value matches any value in a subquery or a list.
BETWEEN	Specifies an inclusive range to test.
LIKE	Determines whether a specific character string matches a specified pattern.
AND	Combines two Boolean expressions and returns TRUE only when both are TRUE.
OR	Combines two Boolean expressions and returns TRUE if either is TRUE.
NOT	Reverses the result of a search condition.

Question

You have a table named Employees that includes a column named StartDate. You want to find who started in any year other than 2014. What query would you use?

Answer

```
SELECT FirstName, LastName
FROM Employees
WHERE YEAR( Employees.StartDate ) <> 2014
```

```
SELECT FirstName, LastName
FROM Employees
WHERE Employees.StartDate < '20140101'
      OR Employees.StartDate >= '20150101'
```

```
SELECT FirstName, LastName
FROM Employees
WHERE Employees.StartDate
      NOT( BETWEEN '20140101' AND '20141231' )
```

Database Session Notes

Filtering in the SELECT Clause Using the TOP Option

- TOP allows you to limit the number or percentage of rows returned by a query
- Works with ORDER BY clause to limit rows by sort order:
 - If ORDER BY list is not unique, results are not deterministic (no single correct result set)
 - Modify ORDER BY list to ensure uniqueness, or use TOP WITH TIES
- Added to SELECT clause:
 - SELECT TOP (N) | TOP (N) Percent
 - With percent, number of rows rounded up (nondeterministic)
 - SELECT TOP (N) WITH TIES
 - Retrieve duplicates where applicable (deterministic)
- **TOP is proprietary to Microsoft SQL Server**

TOP without TIES

- Without the WITH TIES Option:

```
SELECT TOP (5) orderid, custid, orderdate
FROM Sales.Orders
ORDER BY orderdate DESC;
```

orderid	custid	orderdate
11077	65	2008-05-06 00:00:00.000
11076	9	2008-05-06 00:00:00.000
11075	68	2008-05-06 00:00:00.000
11074	73	2008-05-06 00:00:00.000
11073	58	2008-05-05 00:00:00.000

(5 row(s) affected)

Database Session Notes

TOP with TIES

- With the WITH TIES Option:

```
SELECT TOP (5) WITH TIES orderid, custid, orderdate
FROM Sales.Orders
ORDER BY orderdate DESC;
```

orderid	custid	orderdate
11077	65	2008-05-06 00:00:00.000
11076	9	2008-05-06 00:00:00.000
11075	68	2008-05-06 00:00:00.000
11074	73	2008-05-06 00:00:00.000
11073	58	2008-05-05 00:00:00.000
11072	20	2008-05-05 00:00:00.000
11071	46	2008-05-05 00:00:00.000
11070	44	2008-05-05 00:00:00.000

(8 row(s) affected)

Filtering in the ORDER BY Clause Using OFFSET-FETCH

OFFSET-FETCH is an extension to the ORDER BY clause:

- Allows filtering a requested range of rows
 - Dependent on ORDER BY clause
- Provides a mechanism for paging through results
- Specify number of rows to skip, number of rows to retrieve:

```
ORDER BY <order_by_list>
OFFSET <offset_value> ROW(S)
FETCH FIRST|NEXT <fetch_value> ROW(S) ONLY
```

- Available in SQL Server 2012, 2014, and 2016
 - Provides more compatibility than TOP
- While TOP may be used without ORDER BY (with unpredictable results), OFFSET/FETCH, without an ORDER BY, will return an error.

Database Session Notes

OFFSET-FETCH Syntax

- OFFSET value must be supplied
 - May be zero if no skipping is required
- The optional FETCH clause allows all rows following the OFFSET value to be returned
- Natural Language approach to code:
 - ROW and ROWS interchangeable
 - FIRST and NEXT interchangeable
 - ONLY optional—makes meaning clearer to human reader
- OFFSET value and FETCH value may be constants or expressions, including variables and parameters

```
OFFSET <offset_value> ROW|ROWS  
FETCH FIRST|NEXT <fetch_value> ROW|ROWS [ONLY]
```

Understanding NULLs

Database Session Notes

Three-Valued Logic

- SQL Server uses NULLs to mark missing values
 - NULL can be "missing but applicable" or "missing but inapplicable"
 - Customer middle name: Not supplied, or doesn't have one?
 - Missing but applicable:
License plate number of person who owns an automobile.
 - Missing but inapplicable:
License plate number of person who is a pedestrian.
- With no missing values, predicate outputs are TRUE or FALSE only ($5 > 2$, $1=1$)
- With missing values, outputs can be TRUE, FALSE or UNKNOWN ($\text{NULL} > 99$, $\text{NULL} = \text{NULL}$)
- Predicates return UNKNOWN when comparing missing value to another value, including another missing value

Handling NULL in Queries

- Different components of SQL Server handle NULL differently
 - Query filters (ON, WHERE, HAVING) filter out UNKNOWNs (meaning treat NULL like FALSE)
 - CHECK constraints accept UNKNOWNs
 - ORDER BY, DISTINCT treat NULLs as equals
- Testing for NULL
 - Use IS NULL or IS NOT NULL rather than $= \text{NULL}$ or $<> \text{NULL}$

Database Session Notes

Handling NULL in Queries

Sample Data:

```
-- ascending sort order explicitly included for clarity.  
SELECT empid, lastname, region  
FROM HR.Employees  
ORDER BY region ASC;
```

empid	lastname	region
-----	-----	-----
5	Buck	NULL
6	Suurs	NULL
7	King	NULL
9	Dolgopyatova	NULL
8	Cameron	WA
1	Davis	WA
2	Funk	WA
3	Lew	WA
4	Peled	WA

ORDER BY clause
sorts the NULLs
together and first
— a behavior you
cannot override.

Handling NULL in Queries

Incorrectly testing for NULLs

```
SELECT empid, lastname, region  
FROM HR.Employees  
WHERE region = NULL;
```

empid	lastname	region
-----	-----	-----
(0 rows affected)		

This returns
unexpected results

Database Session Notes

Handling NULL in Queries

Correctly testing for NULLs

```
SELECT empid, lastname, region
FROM HR.Employees
WHERE region IS NULL;
```

empid	lastname	region
5	Buck	NULL
6	Suurs	NULL
7	King	NULL
9	Dolgopyatova	NULL

(4 row(s) affected)

Querying Multiple Tables
using JOINS

Database Session Notes

The FROM Clause and Virtual Tables

- FROM clause determines source tables to be used in SELECT statement
- FROM clause can contain tables and operators
- Result set of FROM clause is a **virtual** table
 - Subsequent logical operations in SELECT statement consume this virtual table
- FROM clause can establish table aliases for use by subsequent phases of the query

Join Terminology: Cartesian Product

- Characteristics of a Cartesian product
 - Output or intermediate result of FROM clause
 - Combine all possible combinations of two sets
 - In T-SQL queries, usually not desired
 - Special case: table of numbers

Name		Product		Name	Product
Davis		Alice Mutton		Davis	Alice Mutton
Funk		Crab Meat		Davis	Crab Meat
King		Ipoh Coffee		Davis	Ipoh Coffee
				Funk	Alice Mutton
				Funk	Crab Meat
				Funk	Ipoh Coffee
				King	Alice Mutton
				King	Crab Meat
				King	Ipoh Coffee

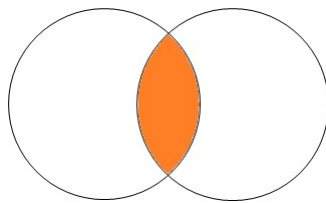
Database Session Notes

Join Types

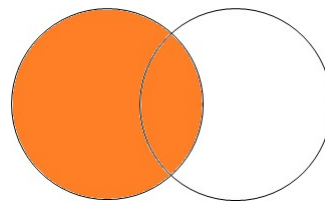
- Join types in FROM clauses specify the operations performed on the virtual table:

Join Type	Description
CROSS JOIN	Combines all rows in both tables (creates Cartesian product)
INNER JOIN / JOIN	Starts with Cartesian product; applies filter to match rows between tables based on predicate
LEFT OUTER JOIN RIGHT OUTER JOIN FULL OUTER JOIN	Starts with Cartesian product; all rows from designated table preserved, matching rows from other table retrieved. Additional NULLs inserted as placeholders

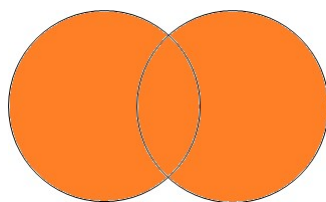
JOIN Types



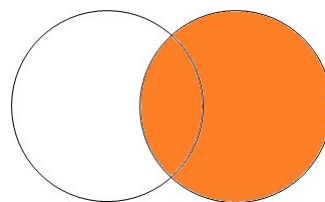
INNER JOIN / JOIN



LEFT JOIN



OUTER JOIN



RIGHT JOIN

Database Session Notes

T-SQL Syntax Choices

- ANSI SQL-92
 - Tables joined by JOIN operator in FROM Clause

```
SELECT ...  
FROM Table1  
      JOIN Table2 ON <on_predicate>
```

- ANSI SQL-89
 - Tables joined by commas in FROM Clause
 - Not recommended: accidental Cartesian products!

```
SELECT ...  
FROM Table1, Table2  
WHERE <where_predicate>
```

Inner Join Syntax

- List tables in FROM Clause separated by JOIN operator
- Table aliases preferred
- Table order does not matter

```
FROM t1 JOIN t2  
      ON t1.column = t2.column
```

```
SELECT o.orderid,  
       o.orderdate,  
       od.productid,  
       od.unitprice,  
       od.qty  
FROM Sales.Orders AS o  
      JOIN Sales.OrderDetails AS od  
      ON o.orderid = od.orderid;
```

Database Session Notes

Inner Join Examples

- Join based on single matching attribute

```
SELECT ...  
FROM Production.Categories AS C  
JOIN Production.Products AS P  
ON C.categoryid = P.categoryid;
```

- Join based on multiple matching attributes (composite join)

```
-- List cities and countries where both  
-- customers and employees live  
SELECT DISTINCT e.city, e.country  
FROM Sales.Customers AS c  
JOIN HR.Employees AS e  
ON c.city = e.city AND c.country = e.country;
```

Understanding Outer Joins

- Returns all rows from one table and any matching rows from second table
- One table's rows are "preserved"
 - Designated with LEFT, RIGHT, FULL keyword
 - All rows from preserved table output to result set
- Matches from other table retrieved
- Additional rows added to results for nonmatched rows
 - NULLs added in places where attributes do not match
- Example: return all customers and, for those who have placed orders, return order information; customers without matching orders will display NULL for order details

Database Session Notes

Outer Join Syntax

- Return all rows from first table, only matches from second:

```
FROM t1 LEFT OUTER JOIN t2 ON  
t1.col = t2.col
```

- Return all rows from second table, only matches from first:

```
FROM t1 RIGHT OUTER JOIN t2 ON  
t1.col = t2.col
```

- Return only rows from first table, with no match in second:

```
FROM t1 LEFT OUTER JOIN t2 ON  
t1.col = t2.col  
WHERE t2.col IS NULL
```

Outer Join Examples

- All customers with order details if present:

```
SELECT c.custid, c.contactname, o.orderid,  
o.orderdate  
FROM Sales.Customers AS C  
LEFT OUTER JOIN Sales.Orders AS O  
ON c.custid = o.custid;
```

- Customers who did not place orders:

```
SELECT c.custid, c.contactname, o.orderid,  
o.orderdate  
FROM Sales.Customers AS C  
LEFT OUTER JOIN Sales.Orders AS O  
ON c.custid = o.custid  
WHERE o.orderid IS NULL;
```


Database Session Notes

Understanding Cross Joins

- Combine each row from first table with each row from second table
- All possible combinations output
- Logical foundation for inner and outer joins
 - Inner join starts with Cartesian product, adds filter
 - Outer join takes Cartesian output, filtered, adds back nonmatching rows (with NULL placeholders)
- Due to Cartesian product output, not typically a desired form of join
- Some useful exceptions:
 - Table of numbers, generating data for testing

Cross Join Syntax

- No matching performed, no ON clause used
- Return all rows from left table combined with each row from right table (ANSI SQL-92 syntax):

```
SELECT ...  
FROM t1 CROSS JOIN t2
```

- Return all rows from left table combined with each row from right table (ANSI SQL-89 syntax):

```
SELECT ...  
FROM t1, t2
```

Database Session Notes


Cross Join Examples

- Create test data by returning all combinations of two inputs:

```
SELECT e1.firstname, e2.lastname  
FROM HR.Employees AS e1  
CROSS JOIN HR.Employees AS e2;
```

Understanding Self Joins

- Why use self joins?
 - Compare rows in same table to each other
- Create two instances of same table in FROM clause
 - At least one alias required
- Example: Return all employees and the name of the employee's manager



empid
lastname
firstname
title
titleofcourtesy
birthdate
hiredate
address
city
region
postalcode
country
phone
mgrid

Database Session Notes

Self Join Examples

- Return all employees with ID of employee's manager when a manager exists (inner join):

```
SELECT e.empid, e.lastname,  
       e.title, e.mgrid, m.lastname  
FROM   HR.Employees AS e  
JOIN   HR.Employees AS m  
ON     e.mgrid=m.empid;
```

- Return all employees with ID of manager (outer join). This will return NULL for the CEO:

```
SELECT e.empid, e.lastname,  
       e.title, m.mgrid  
FROM   HR.Employees AS e  
LEFT OUTER JOIN HR.Employees AS m  
ON     e.mgrid=m.empid;
```

Best Practices

- Table aliases should always be defined when joining tables.
- Joins should be expressed using JOIN and ON keywords.

Database Session Notes

Set Theory and SQL Server

Characteristics of a Set	Example
Elements of a set called Members	Customer as a member of set called Customers
Elements of a set are described by attributes	First name, Last name, Age
Elements must be unique	Customer ID

Set theory does not specify the order of its members

Elements of a SELECT Statement

Element	Expression	Role
SELECT	<select list>	Defines which columns to return
FROM	<table source>	Defines table(s) to query
WHERE	<search condition>	Filters returned data using a predicate
GROUP BY	<group by list>	Arranges rows by groups
HAVING	<search condition>	Filters groups by a predicate
ORDER BY	<order by list>	Sorts the results

Database Session Notes

Logical Query Processing

- | | | |
|----|----------|--------------------|
| 5. | SELECT | <select list> |
| 1. | FROM | <table source> |
| 2. | WHERE | <search condition> |
| 3. | GROUP BY | <group by list> |
| 4. | HAVING | <search condition> |
| 6. | ORDER BY | <order by list> |

The order in which a query is written is not the order in which it is evaluated by SQL Server

Logical Vs. Processing Order

- | | | |
|--------------|--------------|--------------------|
| (1) FROM | (5) SELECT | <select list> |
| (2) WHERE | (1) FROM | <table source> |
| (3) GROUP BY | (2) WHERE | <search condition> |
| (4) HAVING | (3) GROUP BY | <group by list> |
| (5) SELECT | (4) HAVING | <search condition> |
| (6) ORDER BY | (6) ORDER BY | <order by list> |

Processing Order

Logical Order

Database Session Notes

Applying the Logical Order of Operations to Writing SELECT Statements

```
USE TSQL;  
  
SELECT EmployeeId, YEAR(OrderDate) AS OrderYear  
FROM Sales.Orders  
WHERE CustomerId = 71  
GROUP BY EmployeeId, YEAR(OrderDate)  
HAVING COUNT(*) > 1  
ORDER BY EmployeeId, OrderYear;
```

Using DML to Modify Data
The INSERT Statement

Database Session Notes

Using INSERT to Add Data

- The INSERT ... VALUES statement inserts a new row

```
INSERT INTO Sales.OrderDetails
(orderid, productid, unitprice, qty, discount)
VALUES (10255, 39, 18, 2, 0.05);
```

- Table and row constructors add multirow capability to INSERT ... VALUES

```
INSERT INTO Sales.OrderDetails
(orderid, productid, unitprice, qty, discount)
VALUES
(10256, 39, 18, 2, 0.05),
(10258, 39, 18, 5, 0.10);
```

Using INSERT with Data Providers

- INSERT ... SELECT to insert rows from another table:

```
INSERT Sales.OrderDetails
(orderid, productid, unitprice, qty, discount)
SELECT * FROM NewOrderDetails
```

Database Session Notes

Using SELECT INTO

SELECT -> INTO is similar to INSERT <- SELECT

- It also creates a table for the output, fashioned on the output itself
- The new table is based on query column structure
 - Uses column names, data types, and null settings
 - Does not copy constraints or indexes

```
SELECT *  
INTO NewProducts  
FROM PRODUCTION.PRODUCTS WHERE ProductID >= 70
```

- To add further rows to an existing table you use INSERT INTO or INSERT SELECT.
For SELECT INTO, the table should not exist.

Using DML to Modify Data
The UPDATE Statement

Database Session Notes

Using UPDATE to Modify Data

- UPDATE changes all rows in a table or view
- Unless rows are filtered with a WHERE clause or constrained with a JOIN clause
- Column values are changed with the SET clause

```
UPDATE Production.Products
SET    unitprice = (unitprice * 1.04)
WHERE  categoryid = 1 AND discontinued = 0;
```

```
-- Using compound assignment operators
UPDATE Production.Products
SET    unitprice *= 1.04
WHERE  categoryid = 1 AND discontinued = 0;
```

Updating Data in One Table Based on a Join to Another

```
-- Notice use of Alias to make reading better
UPDATE Reason
SET Name += '?'
FROM Production.ScrapReason AS Reason
INNER JOIN Production.WorkOrder AS WorkOrder
ON Reason.ScrapReasonID = WorkOrder.ScrapReasonID
AND WorkOrder.ScrappedQty > 300;
```

Database Session Notes

Using MERGE to Modify Data

MERGE modifies data based on a condition

- When the source matches the target
- When the source has no match in the target
- When the target has no match in the source

```
MERGE TOP (10)
INTO   Store          AS Destination
USING   StoreBackup    AS StagingTable
        ON(Destination.Key = StagingTable.Key)
WHEN NOT MATCHED THEN
        INSERT (C1,..)
        VALUES (Source.C1,..)
WHEN MATCHED THEN
        UPDATE SET Destination.C1 = StagingTable.C1,..;
```

Using IDENTITY

The IDENTITY property generates column values automatically

- Optional seed and increment values can be provided

```
CREATE TABLE Production.Products
(PID int IDENTITY(1,1) NOT NULL, Name VARCHAR(15),...)
```

- Only one column in a table may have IDENTITY defined
- IDENTITY column must be omitted in a normal INSERT statement

```
INSERT INTO Production.Products (Name, ...)
VALUES ('MOC 2072 Manual', ...)
```

- Functions are provided to return last generated values
 - SELECT @@IDENTITY: default scope is session
 - SELECT SCOPE_IDENTITY(): scope is object containing the call
 - SELECT IDENT_CURRENT('tablename'): in this case, scope is defined by tablename
- There is a setting to allow identity columns to be changed manually ON or automatic OFF
 - SET IDENTITY_INSERT <TableName> [ON|OFF]

Database Session Notes

Using DML to Modify Data
The DELETE Statement

DELETE Vs. TRUNCATE

The DELETE Statement

```
DELETE FROM Production.Products  
WHERE Id = 4
```

```
DELETE FROM Production.Products
```

The TRUNCATE Statement

```
TRUNCATE TABLE Production.Products
```

Database Session Notes

Programmability
Variables and Control Flows

Introducing T-SQL Variables

- Variables are objects that allow storage of a value for use later in the same batch
- Variables are defined with the DECLARE keyword
 - In SQL Server 2008 and later, variables can be declared and initialized in the same statement
- Variables are always local to the batch in which they're declared and go out of scope when the batch ends

```
--Declare and initialize variables
DECLARE @numrows INT = 3, @catid INT = 2;
--Use variables to pass parameters to procedure
EXEC Production.ProdsByCategory
    @numrows = @numrows, @catid = @catid;
GO
```

Database Session Notes

Working with Variables

- Initialize a variable using the DECLARE statement

```
DECLARE @i INT = 0;
```

- Assign a single (scalar) value using the SET statement

```
SET @i = 1;
```

- Assign a value to a variable using a SELECT statement

- Be sure that the SELECT statement returns exactly one row

```
SELECT @i = COUNT(*) FROM Sales.SalesOrderHeader;
```

Working with IF...ELSE

IF...ELSE uses a predicate to determine the flow of the code

- The code in the IF block is executed if the predicate evaluates to TRUE
- The code in the ELSE block is executed if the predicate evaluates to FALSE or UNKNOWN
- Very useful when combined with the EXISTS operator

```
IF OBJECT_ID('dbo.t1') IS NULL
    PRINT 'Object does not exist';
ELSE
    DROP TABLE dbo.t1;
GO
```

Database Session Notes

Working with WHILE

- WHILE enables code to execute in a loop
- Statements in the WHILE block repeat as the predicate evaluates to TRUE
- The loop ends when the predicate evaluates to FALSE or UNKNOWN
- Execution can be altered by BREAK or CONTINUE

```
DECLARE @empid AS INT = 1, @lname AS NVARCHAR(20);
WHILE @empid <= 5
BEGIN
    SELECT @lname = lastname FROM HR.Employees
    WHERE empid = @empid;
    PRINT @lname;
    SET @empid += 1;
END;
```

Programmability
VIEWS

Database Session Notes

Writing Queries That Return Results from Views

- Views may be referenced in a SELECT statement just like a table
- Views are named table expressions with definitions stored in a database
- Like derived tables and CTEs, queries that use views can provide encapsulation and simplification
- From an administrative perspective, views can provide a security layer to a database

```
SELECT      <select_list>  
FROM        <view_name>  
ORDER BY    <sort_list>;
```

Creating Simple Views

- Views are saved queries created in a database by administrators and developers
- Views are defined with a single SELECT statement
- ORDER BY is not permitted in a view definition without the use of TOP, OFFSET/FETCH, or FOR XML
- To sort the output, use ORDER BY in the outer query
- View creation supports additional options beyond the scope of this class

```
CREATE VIEW HR.EmpPhoneList  
AS  
SELECT empid, lastname, firstname, phone  
FROM HR.Employees;
```

Database Session Notes

Programmability FUNCTIONs

Writing Queries That Use Inline TVFs

- TVFs are named table expressions with definitions stored in a database
- TVFs return a virtual table to the calling query
- SQL Server provides two types of TVFs:
 - Inline, based on a single SELECT statement
 - Multi-statement, which creates and loads a table variable
- Unlike views, TVFs support input parameters
- Inline TVFs may be thought of as parameterized views

Database Session Notes

Creating Simple Inline TVFs

- TVFs are created by administrators and developers
- Create and name function and optional parameters with CREATE FUNCTION
- Declare return type as TABLE
- Define inline SELECT statement following RETURN

```
CREATE FUNCTION Sales.fn_LineTotal (@orderid INT)
RETURNS TABLE
AS
RETURN
    SELECT orderid,
           CAST((qty * unitprice * (1 - discount)) AS
              DECIMAL(8, 2)) AS line_total
    FROM Sales.OrderDetails
    WHERE orderid = @orderid ;
```

Programmability
STORED PROCEDURES

Database Session Notes

Creating Procedures

- Input parameters passed to procedure logically behave like local variables within procedure code
- Assign name with @ prefix, data type in procedure header
- Refer to parameter in body of procedure

```
CREATE PROCEDURE Production.ProdsByCategory  
(@numrows AS int, @catid AS int)  
AS  
    SELECT TOP(@numrows) productid,  
        productname, unitprice  
    FROM Production.Products  
    WHERE categoryid = @catid;
```

Creating Procedures That Accept Parameters

- Input parameters passed to procedure logically behave like local variables within procedure code
- Assign name with @ prefix, data type in procedure header
- Refer to parameter in body of procedure

```
CREATE PROCEDURE Production.ProdsByCategory  
(@numrows AS int, @catid AS int)  
AS  
    SELECT TOP(@numrows) productid,  
        productname, unitprice  
    FROM Production.Products  
    WHERE categoryid = @catid;
```

Database Session Notes

Examining Stored Procedures

- Stored procedures are collections of T-SQL statements stored in a database
- Procedures can return results, manipulate data, and perform administrative actions on the server
- With other objects, procedures can provide a trusted application programming interface to a database, insulating applications from database structure changes
 - Use views, functions, and procedures to return data
 - Use procedures to modify and add new data
 - Alter procedure definition in one place, rather than update application code

Executing Stored Procedures

- Invoke a stored procedure using EXECUTE or EXEC
- Call procedure with two-part name
- Pass parameters in @name=value form, using appropriate data type

```
EXEC Production.ProductsbySuppliers  
    @supplierid = 1;
```

```
EXEC Production.ProductsbySuppliers  
    @supplierid = 1, @numrows = 2;
```

Database Session Notes

Writing Queries with Dynamic SQL

- Using `sp_executesql`
 - Accepts string as code to be run
 - Supports input, output parameters for query
 - Allows parameterized code while minimizing risk of SQL injection
 - Can perform better than EXEC due to query plan reuse

```
DECLARE @sqlcode AS NVARCHAR(256) =  
    N'<code_to_run>';  
EXEC sys.sp_executesql @statement = @sqlcode;
```

```
DECLARE @sqlcode AS NVARCHAR(256) =  
    N'SELECT GETDATE() AS dt';  
EXEC sys.sp_executesql @statement = @sqlcode;
```