CS108: Advanced Database

- Database Programming

Lecture 02:

The Foundation Statements of T-SQL

Overview

- The fundamental Transact-SQL (T-SQL) statements
  - SELECT
  - INSERT
  - UPDATE
  - DELETE

Fundamental T-SQL statements

- SELECT, INSERT, UPDATE, and DELETE are the bread and butter of T-SQL
- These statements make up the basis of T-SQL's Data Manipulation Language (DML)
- SQL also provides many operators and keywords that help refine the queries

Basic SELECT Statement

- The SELECT statement and the structures used are basic commands will perform with SQL Server
- The basic syntax rules for a SELECT statement:

```
SELECT [ALL|DISTINCT] [TOP (<expression>) [PERCENT] [WITH TIES]]
       <column list>
[FROM  <source table(s)/view(s)>]
[WHERE <restrictive condition>]
[GROUP BY <column name or expression using a column in the SELECT list>]
[HAVING <restrictive condition based on the GROUP BY results>]
[ORDER BY <column list>]
[[FOR XML {RAW|AUTO|EXPLICIT|PATH [(<element>)]}[, XMLDATA]
                                  [, ELEMENTS][, BINARY base 64]]
[;]
```

Elements of the SELECT Statement

- The clauses are logically processed in the following order:

1. FROM - Queries the rows from …
2. WHERE - Filters only record where …
3. GROUP BY - Groups the record by …
4. HAVING - Filters only groups having …
5. SELECT - Selects (returns) for each group the …
6. ORDER BY - Orders (sorts) the rows in the output by …

Basic Query

- A SELECT indicates that we are merely reading information, as opposed to modifying it.
- What we are selecting is identified by an expression or column list immediately following the SELECT.
- The FROM statement specifies the name of the table or tables from which we want to get our data.
- With these, we have enough to create a basic SELECT statement.

Example: Query Metadata

- Most modern relational databases store metadata in system tables
- In SQL Server, we can query metadata through system views or stored procedures
- INFORMATION_SCHEMA
  - An information schema view is a common method for obtaining metadata.
  - This schema is contained in each database. Each information schema view contains metadata for all data objects stored in that particular database.

Example: INFORMATION_SCHEMA

- Ways to obtain metadata
  - Standard SQL SELECT
  - System stored procedures
- Commonly used views
  - TABLES
  - COLUMNS
  - TABLE_CONSTRAINTS
  - VIEWS
  - KEY_COLUMN_USAGE

Results | Messages

	TABLE_CATALOG	TABLE_SCHEMA	TABLE_NAME	TABLE_
1	AdventureWorks2014	Sales	vStoreWithContacts	VIEW
2	AdventureWorks2014	Production	ScrapReason	BASE T
3	AdventureWorks2014	Sales	vStoreWithAddresses	VIEW
4	AdventureWorks2014	Purchasing	vVendorWithContacts	VIEW
5	AdventureWorks2014	HumanResources	Shift	BASE T
6	AdventureWorks2014	Purchasing	vVendorWithAddresses	VIEW
7	AdventureWorks2014	Production	ProductCategory	BASE T
8	AdventureWorks2014	Purchasing	ShipMethod	BASE T
9	AdventureWorks2014	Production	ProductCostHistory	BASE T
10	AdventureWorks2014	Production	ProductDescription	BASE T
11	AdventureWorks2014	Sales	ShoppingCartItem	BASE T
12	AdventureWorks2014	Production	ProductDocument	BASE T
13	AdventureWorks2014	dbo	DatabaseLog	BASE T
14	AdventureWorks2014	Production	ProductInventory	BASE T
15	AdventureWorks2014	Sales	SpecialOffer	BASE T
16	AdventureWorks2014	dbo	ErrorLog	BASE T
17	AdventureWorks2014	Production	ProductListPriceHistory	BASE T

```
SELECT * FROM INFORMATION_SCHEMA.TABLES;
```

More Examples

```
SELECT * FROM Sales.Customer;
```

	CustomerID	PersonID	StoreID	TerritoryID	AccountNumber	rowguid
1	1	NULL	934	1	AW00000001	3F5AE95E-B87D-4AED-95B4-C3797AFCB74F
2	2	NULL	1028	1	AW00000002	E552F657-A9AF-4A7D-A645-C429D6E02491
3	3	NULL	642	4	AW00000003	130774B1-DB21-4EF3-98C8-C104BCD6ED6D
4	4	NULL	932	4	AW00000004	FF862851-1DAA-4044-BE7C-3E85583C054D
5	5	NULL	1026	4	AW00000005	83905BDC-6F5E-4F71-B162-C98DA069F38A
6	6	NULL	644	4	AW00000006	1A92DF88-BFA2-467D-BD54-FCB9E647FDD7
7	7	NULL	930	1	AW00000007	03E9273E-B193-448E-9823-FE0C44AEED78
8	8	NULL	1024	5	AW00000008	801368B1-4323-4BFA-8BEA-5B5B1E4BD4A0
9	9	NULL	620	5	AW00000009	B900BB7F-23C3-481D-80DA-C49A5BD6F772
10	10	NULL	928	6	AW00000010	CDB6698D-2FF1-4FBA-8F22-60AD1D11DABD

(19820 row(s) affected)

```
SELECT LastName FROM Person.Person;
```

	LastName
1	Abbas
2	Abel
3	Abercrombie
4	Abercrombie
5	Abercrombie
6	Abolrous
7	Abolrous
8	Acevedo
9	Achong
10	Ackerman
11	Ackerman
12	Adams
13	Adams
14	Adams

The FROM Clause

- The FROM clause is the very first query clause that is logically processed
- We specify the names of the tables that we want to query and table operators that operate on those tables

Note: If an identifier is irregular - for example, if it has embedded spaces or special characters, starts with a digit, or is a reserved Keyword - we have to delimit it

We can delimit identifiers in SQL Server in a couple of ways:

- double quotes - for example, “Order Details”
- square brackets - for example, [Order Details]

The WHERE Clause

- In the WHERE clause, we specify a predicate or logical expression to filter the rows returned by the FROM phase
- Only rows for which the logical expression evaluates to TRUE are returned by the WHERE phase to the subsequent logical query processing phase
- The WHERE clause has significance when it comes to query performance
  - Based on the filter expression, SQL Server evaluates the use of indexes to access the required data - get the required data with much less work compared to applying full table scans

The WHERE Clause (Cont.)

- The logical expression evaluates to TRUE are returned by the WHERE phase, actually, the query filters is "*accept TRUE*"
  - WHERE phase filters out both FALSE and UNKNOWN
  - Conversely, CHECK constraints is "*reject FALSE*"
- Example:

```
SELECT ProductID, Color
FROM   Production.Product
WHERE  Color <> 'Black'
```

	ProductID	Color
1	320	Silver
2	321	Silver
3	332	Silver
4	351	Silver
5	352	Silver
6	461	Silver
7	679	Silver
8	706	Red

```
SELECT ProductID, Color
FROM   Production.Product
WHERE  Color <> 'Black' OR Color IS NULL
```

	ProductID	Color
1	1	NULL
2	2	NULL
3	3	NULL
4	4	NULL
5	316	NULL
6	320	Silver
7	321	Silver
8	323	NULL

Three-valued Predicate Logic in T-SQL

- T-SQL uses three-valued predicate logic, where logical expressions can evaluate to TRUE, FALSE, or UNKNOWN.
- A logical expression involving only existing or present values evaluates to either TRUE or FALSE
- When the logical expression involves a missing value, it evaluates to UNKNOWN
- One of the tricky aspects of UNKNOWN is that when we negate it, we still get UNKNOWN
- An expression comparing two NULL marks (NULL = NULL) evaluates to UNKNOWN

The GROUP BY Clause

- The GROUP BY clause is used to aggregate information.
- For example, we want to know how many parts were ordered in a given set of orders

```
SELECT SalesOrderID, OrderQty
FROM   Sales.SalesOrderDetail
WHERE  SalesOrderID IN (43660, 43670, 43672);
```

```
SalesOrderID OrderQty
------------ --------
43660        1
43660        1
43670        1
43670        2
43670        2
43670        1
43672        6
43672        2
43672        1

(9 row(s) affected)
```

- We can make use of the GROUP BY clause with an aggregator

```
SELECT SalesOrderID, SUM(OrderQty)
FROM   Sales.SalesOrderDetail
WHERE  SalesOrderID IN (43660, 43670, 43672)
GROUP BY SalesOrderID;
```

```
SalesOrderID SUM(OrderQty)
------------ --------
43660        2
43670        6
43672        9

(3 row(s) affected)
```

The GROUP BY Clause (Cont.)

- When using a GROUP BY clause, all the columns in the SELECT list must either be aggregates (SUM, MIN/MAX, AVG, and so on) or columns included in the GROUP BY clause
- Likewise, if using an aggregate in the SELECT list, the SELECT list must only contain aggregates, or there must be a GROUP BY clause
- We can also group based on multiple columns. To do this we just add a comma and the next column name

The GROUP BY Clause (Cont.)

- All phases subsequent to the GROUP BY phase must operate on groups as opposed to operating on individual rows
- Each group is ultimately represented by a single row in the final result of the query
- All expressions processed in phases subsequent to the GROUP BY phase are required to guarantee returning a scalar per group

The GROUP BY Clause (Cont.)

- Expressions based on elements that participate in the GROUP BY list meet the requirement
  - By definition each *group* has only one *unique* occurrence of each GROUP BY element
- For example:

```
SELECT Color, YEAR(SellStartDate)
FROM   Production.Product
WHERE  Color <> 'Black'
GROUP BY Color, YEAR(SellStartDate)
```

Color	YEAR(...)
Blue	2011
Blue	2013
Grey	2012
Multi	2011
Multi	2012
Red	2008
Red	2011
Red	2012

Example: Aggregating Data

```
SELECT CustomerID, SalesPersonID, COUNT(*)
FROM   Sales.SalesOrderHeader
WHERE  CustomerID <= 11010
GROUP BY CustomerID, SalesPersonID
```

```
CustomerID  SalesPersonID
----------- ------------- -----------
11000       NULL          3
11001       NULL          3
11002       NULL          3
11003       NULL          3
11004       NULL          3
11005       NULL          3
11006       NULL          3
11007       NULL          3
11008       NULL          3
11009       NULL          3
11010       NULL          3
(11 row(s) affected)
```

Aggregates (Cont.)

- Elements that do not participate in the GROUP BY list are allowed only as inputs to an aggregate function
  - AVG: This one is for computing averages
  - MIN/MAX: These grab the minimum and maximum amounts for each grouping for a selected column
  - COUNT(Expression | *): The COUNT(*) function is about counting the rows in a query

Aggregates (Cont.)

- If an attribute that does not participate in the GROUP BY list and not as an input to an aggregate function, we get an error
  - No guarantee that the expression will return a single value per group
- Example:

```
SELECT    Color, YEAR(SellStartDate), ProductID
FROM      Production.Product
WHERE     Color <> 'Black'
GROUP BY Color, YEAR(SellStartDate)
```

```
Column 'Production.Product.ProductID' is invalid in the select list because it
is not contained in either an aggregate function or the GROUP BY clause.
```

Aggregates (Cont.)

- Note that all aggregate functions *ignore NULL* marks with one exception - COUNT(*)
- For example, consider a group of five rows called *QTY*
  - *COUNT(*) would return 5*
  - *COUNT(QTY) would return 4*
- DISTINCT keyword to handle only distinct occurrences of known values
  - *COUNT(DISTINCT QTY) would return 2*
  - *SUM(QTY) would return 60, and SUM(DISTINCT QTY) would return 40*

```
QTY
------
30
10
NULL
10
10
```

Groups with the HAVING Clause

- The HAVING clause is used only when there is also a GROUP BY in the query
  - Whereas the WHERE clause is applied to each row before it even has a chance to become part of a group
  - The HAVING clause is applied to the aggregated value for that group

WHERE: All of the conditions have been against specific *rows* before grouping.

HAVING: Place conditions on the *groups* themselves. In other words, apply the condition after the *groups* are fully accumulated.

The HAVING Clause

- With the HAVING clause, we can specify a predicate to filter groups as opposed to filtering individual rows
- Only groups for which the logical expression in the HAVING clause evaluates to TRUE are returned by the HAVING phase. (FALSE or UNKNOWN are filtered out)
- Because the HAVING clause is processed after the rows have been grouped, we can refer to *aggregate functions* in the logical expression. For example, HAVING COUNT(*) > 1

Example

```
CREATE TABLE HumanResources.Employee2 (
  EmployeeID INT NOT NULL,
  ManagerID  INT NULL REFERENCES HumanResources.Employee2(EmployeeID),
  JobTitle   NVARCHAR(50) NOT NULL,
  LastName   NVARCHAR(50) NOT NULL,
  FirstName  NVARCHAR(50) NOT NULL,
  CONSTRAINT PK_EmployeeID PRIMARY KEY CLUSTERED ( EmployeeID ASC )
);

INSERT INTO
HumanResources.Employee2(EmployeeID,ManagerID,JobTitle,LastName,
    FirstName) VALUES
(1, NULL, 'CEO','Smith', 'Hunter'),
(2, 1, 'CFO', 'Jones', 'Drew'),
(3, 1, 'COO','Lenzy','Sheila'),
(4, 1, 'CTO', 'Huntington', 'Karla'),
(5, 4, 'VP of Engineering', 'Gutierrez', 'Ron'),
(8, 5, 'VP of Engineering', 'Gutierrez', 'Ron'),
(9, 5, 'Software Engineer', 'Bray', 'Marky'),
(10 ,5, 'Data Architect', 'Cheechov', 'Robert'),
(11 ,5, 'Software Engineer', 'Gale', 'Sue'),
(6, 4, 'VP of Professional Services', 'Cross', 'Gary'),
(7, 4, 'VP of Security','Lebowski','Jeff');
```

Example: The HAVING Clause

```
SELECT ManagerID, COUNT(*)
FROM   HumanResources.Employee2
GROUP BY ManagerID;
```

ManagerID	COUNT(*)
NULL	1
1	3
4	3
5	4

```
SELECT ManagerID, COUNT(*)
FROM   HumanResources.Employee2
WHERE  EmployeeID != 5
GROUP BY ManagerID;
```

ManagerID	COUNT(*)
NULL	1
1	3
4	2
5	4

```
SELECT ManagerID, COUNT(*)
FROM   HumanResources.Employee2
WHERE  EmployeeID != 5
GROUP BY ManagerID
HAVING COUNT(*) > 3;
```

ManagerID	COUNT(*)
5	4

The SELECT Clause

- The SELECT clause is where we specify the attributes that we want to return in the result table of the query
- T-SQL supports a couple of other forms with which we can alias expressions
  - <expression> AS <alias>,
    - AVG(UnitPrice) AS Price
  - <expression> <alias>,
    - AVG(UnitPrice) Price
  - <alias> = <expression>,
    - Price = AVG(UnitPrice)

Alias

- Column alias: representing derived and constant columns

```
SELECT    CategoryID, AVG(UnitPrice) Price
FROM      Products
GROUP BY CategoryID
ORDER BY Price;
```

Column alias can be used in ORDER BY

```
SELECT ProductName, UnitPrice * 0.9 Discount
FROM   Products
WHERE  UnitPrice * 0.9 > 20;
```

Column alias can NOT be used in WHERE or HAVING clause (SQL Server)

- Table alias: commonly used in table joins and sub-queries

```
SELECT ProductName, CategoryName
FROM   Products AS p, Categories c
Where  p.CategoryID = c.CategoryID
```

"AS" is optional

If an alias is assigned, it must be used instead of the original table name

Alias Symbol

- Column alias: use [ ] or ' '

```
SELECT   ProductName, UnitPrice * 0.9 AS 'Discount Price'
FROM     Products
ORDER BY 'Discount Price';
```

```
SELECT   ProductName, UnitPrice * 0.9 AS [Discount Price]
FROM     Products
ORDER BY [Discount Price];
```

- Table alias: use [ ]

```
SELECT ProductName, CategoryName
FROM   Products AS [table p], Categories c
WHERE  [table p].CategoryID = c.CategoryID
```

The SELECT Clause (Cont.)

- The SELECT clause is processed after the FROM, WHERE, GROUP BY, and HAVING clauses
  - Aliases in the SELECT clause do not exist as far as clauses that are processed before the SELECT clause

```
SELECT Color, YEAR(SellStartDate) AS Year
FROM   Production.Product
WHERE  Year > 2007
```

```
Msg 207, Level 16, State 1, Line 3
Invalid column name 'Year'.
```

- Use of the same expression YEAR(…) in the query

```
SELECT Color, YEAR(SellStartDate) AS Year
FROM   Production.Product
WHERE  YEAR(SellStartDate) > 2007
```

The SELECT Clause - Result Set

- In the relational model, operations on relations are based on relational algebra and result in a relation (a set).
- In SQL, a SELECT query is not guaranteed to return a true set - namely, unique rows with no guaranteed order.
  - without a key, uniqueness of rows is not guaranteed, which case the table isn't a set; it's a multiset or a bag
- The term "result set" is often used to describe the output of a SELECT query
  - but a result set doesn't necessarily qualify as a set in the mathematical sense

The DISTINCT and ALL Predicates

- SQL provides the means to guarantee uniqueness in the result of a SELECT statement in the form of a DISTINCT clause that removes duplicate rows

```
SELECT DISTINCT Color, YEAR(SellStartDate) AS Year
FROM            Production.Product
WHERE           YEAR(SellStartDate) > 2007
```

Color	Year
NULL	2008
NULL	2012
NULL	2013
...	

- ALL predicate says to include every row. ALL is the default for any SELECT statement.

```
SELECT ALL      Color, YEAR(SellStartDate) AS Year
FROM            Production.Product
WHERE           YEAR(SellStartDate) > 2007
```

Color	Year
NULL	2008
NULL	2008
NULL	2008
...	

The SELECT Clause (Cont.)

- SQL supports the use of an asterisk (*) in the SELECT list to request all attributes from the queried tables
- Such use of an asterisk is a bad programming practice in most cases
  - Any schema changes applied to the table - such as adding or removing columns - might result in failures in the client application
- It is still *not allowed* to refer to a column alias that was created in the *same* SELECT clause

```
SELECT Color,
       YEAR(SellStartDate) AS OrderYear,
       OrderYear + 1       AS NextYear
FROM   Production.Product
```

The ORDER BY Clause

- The ORDER BY clause allows us to sort the rows in the output for presentation purposes
- In terms of logical query processing, ORDER BY is the very last clause to be processed
- We can use any combination of columns in our ORDER BY clause
- The DESC (ASC) keyword tells SQL Server that the ORDER BY should work in descending (ascending) order

Example: ORDER BY

```
SELECT Name, SalesPersonID
FROM   Sales.Store
WHERE  Name BETWEEN 'g' AND 'j'
       AND SalesPersonID > 283
ORDER BY SalesPersonID, Name DESC;
```

```
Name                                               SalesPersonID
-------------------------------------------------- -------------
Inexpensive Parts Shop                             286
Ideal Components                                   286
Helpful Sales and Repair Service                   286
Helmets and Cycles                                 286
Global Sports Outlet                               286
Gears and Parts Company                            286
Irregulars Outlet                                  288
Hometown Riding Supplies                           288
Good Bicycle Store                                 288
Global Bike Retailers                              288
Instruments and Parts Company                      289
Instant Cycle Store                                290
Impervious Paint Company                           290
Hiatus Bike Tours                                  290
Getaway Inn                                        290
(15 row(s) affected)
```

The ORDER BY Clause (Cont.)

- A table has no guaranteed order, it is a set or multiset, a set has no order
- A table without specifying an ORDER BY clause, the query returns a table result, and SQL Server is free to return the rows in the output in any order
- The only way to guarantee that the rows in the result are sorted is to explicitly specify an ORDER BY clause
- If we do specify an ORDER BY clause, the result cannot qualify as a table - what standard SQL calls a *cursor* - a nonrelational result with order guaranteed among rows

The ORDER BY Clause (Cont.)

- T-SQL allows us to specify ordinal positions of columns in the ORDER BY clause, based on the order in which the columns appear in the SELECT list

```
SELECT Color, YEAR(SellStartDate) AS OrderYear,
FROM   Production.Product
ORDER BY COLOR, YEAR(SellStartDate)
```

```
FROM   Production ...
ORDER BY 1, 2
```

- T-SQL allows us to specify elements in the ORDER BY clause that do not appear in the SELECT clause
- However, when DISTINCT is specified, we are restricted in the ORDER BY list only to elements that appear in the SELECT list

The ORDER BY Clause (Cont.)

- T-SQL allows us to specify elements in the ORDER BY clause that do not appear in the SELECT clause

```
SELECT   Color, YEAR(SellStartDate) AS OrderYear
FROM     Production.Product
ORDER BY ProductID
```

- However, when DISTINCT is specified, we are restricted in the ORDER BY list only to elements that appear in the SELECT list.

```
SELECT DISTINCT Color, YEAR(SellStartDate) AS OrderYear
FROM            Production.Product
ORDER BY        ProductID
```

```
ORDER BY Color | ORDER BY OrderYear | ORDER BY Color, OrderYear
```

The TOP Filter

- The TOP option allows us to limit the number or percentage of rows that the query returns.

Color	Year
NULL	2008
NULL	2008
NULL	2008
NULL	2008
NULL	2008

```
SELECT TOP(5) Color, YEAR(SellStartDate) AS OrderYear
FROM          Production.Product
ORDER BY      ProductID
```

- If DISTINCT is specified in the SELECT clause, the TOP filter is evaluated after duplicate rows have been removed
- The PERCENT keyword calculates the number of rows to return based on a percentage of the number of qualifying rows
- If we include WITH TIES, additional rows will be included if their values match or tie, the values of the loase row

The OFFSET-FETCH Filter

- Standard SQL defines a TOP-like filter called OFFSET-FETCH that does support skipping capabilities
- The OFFSET-FETCH filter is considered part of the ORDER BY clause, which normally serves a presentation ordering purpose
- By using the OFFSET clause, we can indicate how many rows to skip, and by using the FETCH clause, we can indicate how many rows to filter after the skipped rows

The OFFSET-FETCH Filter

- By using the OFFSET clause, we can indicate how many rows to skip, and by using the FETCH clause, we can indicate how many rows to filter after the skipped rows.

```
[ORDER BY <column list> [ASC|DESC]
[OFFSET <count expression> {ROW|ROWS}
  [FETCH {FIRST|NEXT} <count expression> {ROW|ROWS} ONLY]]
```

```
SELECT   Color, YEAR(SellStartDate) AS OrderYear
FROM     Production.Product
ORDER BY ProductID
OFFSET 10 ROWS FETCH NEXT 5 ROWS ONLY;
```

Color	Year
Black	2008
NULL	2008
NULL	2008
NULL	2008
NULL	2008

Window Functions

- A window function is a function that, for each row in the underlying query, operates on a window of rows and computes a scalar result value
- The window of rows is defined by using an OVER clause
- Window functions are very profound and allow us to address a wide variety of needs
- The syntax of window function and OVER Clause:

```
WindowFunction (<expression>) OVER (
   [ PARTITION BY <expression list> ]
   [ ORDER BY <order list> ]
)
```

Window Functions

- A window function operates on a set of rows exposed to it by a clause called OVER
- For each row in the underlying query, the OVER clause exposes to the function a subset of the rows from the underlying query's result set
- The OVER clause can restrict the rows in the window
  - by using the PARTITION BY subclause, and
  - by using the ORDER BY subclause to define ordering for the calculation

Window Functions

- The ROW_NUMBER window function assigns unique, sequential, incrementing integers to the rows in the result within the respective partition, based on the indicated ordering
- For example,

```
SELECT   Color, YEAR(SellStartDate) AS OrderYear, ListPrice,
         ROW_NUMBER() OVER (PARTITION BY YEAR(SellStartDate)
                            ORDER BY ListPrice) AS NUMBER
FROM     Production.Product
WHERE    ListPrice > 0
ORDER BY YEAR(SellStartDate)
```

  - Partitions the window by YEAR(SellStartDate)
  - Order the records in each window by ListPrice
  - Assign the row number to each record in each window

Window Functions

```
Color           OrderYear   ListPrice             Rownum
--------------- ----------- --------------------- -----------
NULL            2008        133.34                1
NULL            2008        133.34                2
NULL            2008        133.34                3
...
NULL            2008        196.92                9
Black           2008        1431.50               10
Red             2008        1431.50               11
Multi           2011        8.99                  1
White           2011        9.50                  2
White           2011        9.50                  3
Blue            2011        34.99                 4
...
Red             2011        3578.27               71
Red             2011        3578.27               72
NULL            2012        13.99                 1
NULL            2012        19.99                 2
...
(304 row(s) affected)
```

Summary

- To put it all together, the following list presents the logical order in which all clauses discussed so far are processed:
  1. FROM
  2. WHERE
  3. GROUP BY
  4. SELECT
     - Expressions
     - DISTINCT
  5. ORDER BY
     - TOP/OFFSET-FETCH

Exercises

1. Write a query against the Production.Product table that returns ProductID, Color and SellStartDate that SellStartDate in 2007.
2. Write a query against the Sales.SalesOrderDetail table that returns orders with total value (OrderQty* UnitPrice) greater than 10,000, sorted by total value.

	SalesOrderID	SalesOrderDetailID	CarrierTrackingNumber	OrderQty	ProductID	SpecialOfferID	UnitPrice	UnitPriceDiscount	LineTotal
1	43659	1	4911-403C-98	1	776	1	2024.994	0.00	2024.994000
2	43659	2	4911-403C-98	3	777	1	2024.994	0.00	6074.982000
3	43659	3	4911-403C-98	1	778	1	2024.994	0.00	2024.994000
4	43659	4	4911-403C-98	1	771	1	2039.994	0.00	2039.994000
5	43659	5	4911-403C-98	1	772	1	2039.994	0.00	2039.994000
6	43659	6	4911-403C-98	2	773	1	2039.994	0.00	4079.988000
7	43659	7	4911-403C-98	1	774	1	2039.994	0.00	2039.994000
8	43659	8	4911-403C-98	3	714	1	2		
9	43659	9	4911-403C-98	1	716	1	2		
10	43659	10	4911-403C-98	6	709	1	5.70	0.00	34.200000
11	43659	11	4911-403C-98	2	712	1	5.1865	0.00	10.373000
12	43659	12	4911-403C-98	4	711	1	20.1865	0.00	80.746000

SalesOrderDetail

Exercises

3. Write a query against the Sales.SalesOrderHeader table that returns the three TerritoryID with the highest average freight in 2012 (based on OrderDate).

	SalesOrderID	OrderDate	SalesOrderNumber	AccountNumber	CustomerID	SalesPersonID	TerritoryID	SubTotal	Freight
1	43659	2011-05-31 00:00:00.000	SO43659	10-4020-000676	29825	279	5	20565.6206	616.0984
2	43660	2011-05-31 00:00:00.000	SO43660	10-4020-000117	29672	279	5	1294.2529	38.8276
3	43661	2011-05-31 00:00:00.000	SO43661	10-4020-000442	29734	282	6	32726.4786	985.553
4	43662	2011-05-31 00:00:00.000	SO43662	10-4020-000227	29994	282	6	28832.5289	867.2389
5	43663	2011-05-31 00:00:00.000	SO43663	10-4020-000510	29565	276	4	419.4589	12.5838
6	43664	2011-05-31 00:00:00.000	SO43664	10-4020-000397	29898	280	1	24432.6088	732.81
7	43665	2011-05-31 00:00:00.000	SO43665	10-4020-000146	29580	283	1	14352.7713	429.9821
8	43666	2011-05-31 00:00:00.000	SO43666	10-4020-000511	30052	276	4	5056.4896	151.9921
9	43667	2011-05-31 00:00:00.000	SO43667	10-4020-000646	29974	277	3	6107.082	183.1626
10	43668	2011-05-31 00:00:00.000	SO43668	10-4020-000514	29614	282	6	35944.1562	1081.8...
11	43669	2011-05-31 00:00:00.000	SO43669	10-4020-000578	29747	283	1	714.7043	22.0367
12	43670	2011-05-31 00:00:00.000	SO43670	10-4020-000504	29566	275	3	6122.082	183.6126
13	43671	2011-05-31 00:00:00.000	SO43671	10-4020-000200	29890	283	1	8128.7876	244.0042
14	43672	2011-05-31 00:00:00.000	SO43672					6124.182	183.6257
15	43673	2011-05-31 00:00:00.000	SO43673					3746.2015	111.8629
16	43674	2011-05-31 00:00:00.000	SO43674	10-4020-000083	29596	282	6	2624.382	78.7315

SalesOrderHeader

ProductID	int	Not null
Name	Name (user-defined type) nvarchar(50)	Not null
ProductNumber	nvarchar(25)	Not null
MakeFlag	Flag (user-defined type) bit	Not null
FinishedGoodsFlag	Flag (user-defined type) bit	Not null
Color	nvarchar(15)	null
SafetyStockLevel	smallint	Not null
ReorderPoint	smallint	Not null
StandardCost	money	Not null
ListPrice	money	Not null
Size	nvarchar(5)	null
SizeUnitMeasureCode	nchar(3)	null
WeightUnitMeasureCode	nchar(3)	null
Weight	decimal (8,2)	null
DaysToManufacture	int	Not null
ProductLine	nchar(2)	null
Class	nchar(2)	null
Style	nchar(2)	null
ProductSubcategoryID	smallint	null
ProductModelID	int	null
SellStartDate	datetime	Not null
SellEndDate	datetime	null
DiscontinuedDate	datetime	null
rowguid	uniqueidentifier ROWGUIDCOL	Not null
ModifiedDate	datetime	Not null

SalesOrderID	int	Not null	Primary key. Foreign key to SalesOrderHeader.SalesOrderID.
SalesOrderDetailID	int	Not null	Primary key. A sequential number used to ensure data uniqueness..
CarrierTrackingNumber	nvarchar(25)	Null	Shipment tracking number supplied by the shipper.
OrderQty	smallint	Not null	Quantity ordered per product.
ProductID	int	Not null	Product sold to customer. Foreign key to Product.ProductID.
SpecialOfferID	int	Not null	Promotional code. Foreign key to SpecialOffer.SpecialOfferID.
UnitPrice	money	Not null	Selling price of a single product.
UnitPriceDiscount	money	Not null	Discount amount.
LineTotal	Computed as OrderQty * UnitPrice	Not null	Per product subtotal.
rowguid	uniqueidentifier ROWGUIDCOL	Not null	ROWGUIDCOL number uniquely identifying the row. Used to support a merge replication sample.
ModifiedDate	datetime	Not null	Date and time the row was last updated.

SalesOrderID	int	Not null
RevisionNumber	tinyint	Not null
OrderDate	datetime	Not null
DueDate	datetime	Not null
ShipDate	datetime	Null
Status	tinyint	Not null
OnlineOrderFlag	Flag (user-defined type) bit	Not null
SalesOrderNumber	nvarchar(25)	Not null
PurchaseOrderNumber	OrderNumber (user-defined type) nvarchar(25)	Null
AccountNumber	AccountNumber (user-defined type) nvarchar(15)	Null
CustomerID	int	Not null
ContactID	int	Not null
SalesPersonID	int	Null
TerritoryID	int	Null
BillToAddressID	int	Not null
ShipToAddressID	int	Not null
ShipMethodID	int	Not null
CreditCardID	int	Null
CreditCardApprovalCode	varchar(15)	Null
CurrencyRateID	int	Null
SubTotal	money	Not null
TaxAmt	money	Not null
Freight	money	Not null
TotalDue	Computed as SubTotal + TaxAmt + Freight	Not null
Comment	nvarchar(128)	Null
rowguid	uniqueidentifier ROWGUIDCOL	Not null
ModifiedDate	datetime	Not null