MS SQL Server

By
Srikanth Pragada

What is MS SQL Server?

- It is a relational database management system from Microsoft.
- Runs on Windows platform, Linux and Docker.
- Uses TSQL (Transact SQL) for data processing.
- Typically used as the database engine for web applications developed using ASP.NET.
- MS SQL Server Express Edition is a cut-short edition of full-blown MS SQL Server.

Authentication Type

Windows Authentication

- Windows 2000+ users and groups are mapped into SQL Server Logins in their Windows user profile. When they attempt to log into SQL Server, they are validated through the Windows domain and mapped to roles according to the Login. These roles identify what the user is allowed to do.
- The best part of this model is that you have only one password. (If you change it in the Windows domain, then it's changed for your SQL Server logins too.) The downside is that mapping this process can get complex and, to administer the Windows user side of things, you must be a domain administrator.

SQL Server Authentication

- The security does not care at all about what the user's rights to the network are, but rather what you explicitly set up in SQL Server. The authentication process doesn't take into account the current network login at all; instead, the user provides a SQL Server-specific login and password.
- This can be nice because the administrator for a given SQL Server doesn't need to be a domain administrator to give rights to users on the SQL Server. The process also tends to be somewhat simpler than under Windows authentication. Finally, it means that one user can have multiple logins that give different rights to different things.

Transact-SQL (T-SQL)

- Transact-SQL is central to the use of SQL Server.
- All applications that communicate with SQL Server do so by sending Transact-SQL statements to the server, regardless of an application's user interface.

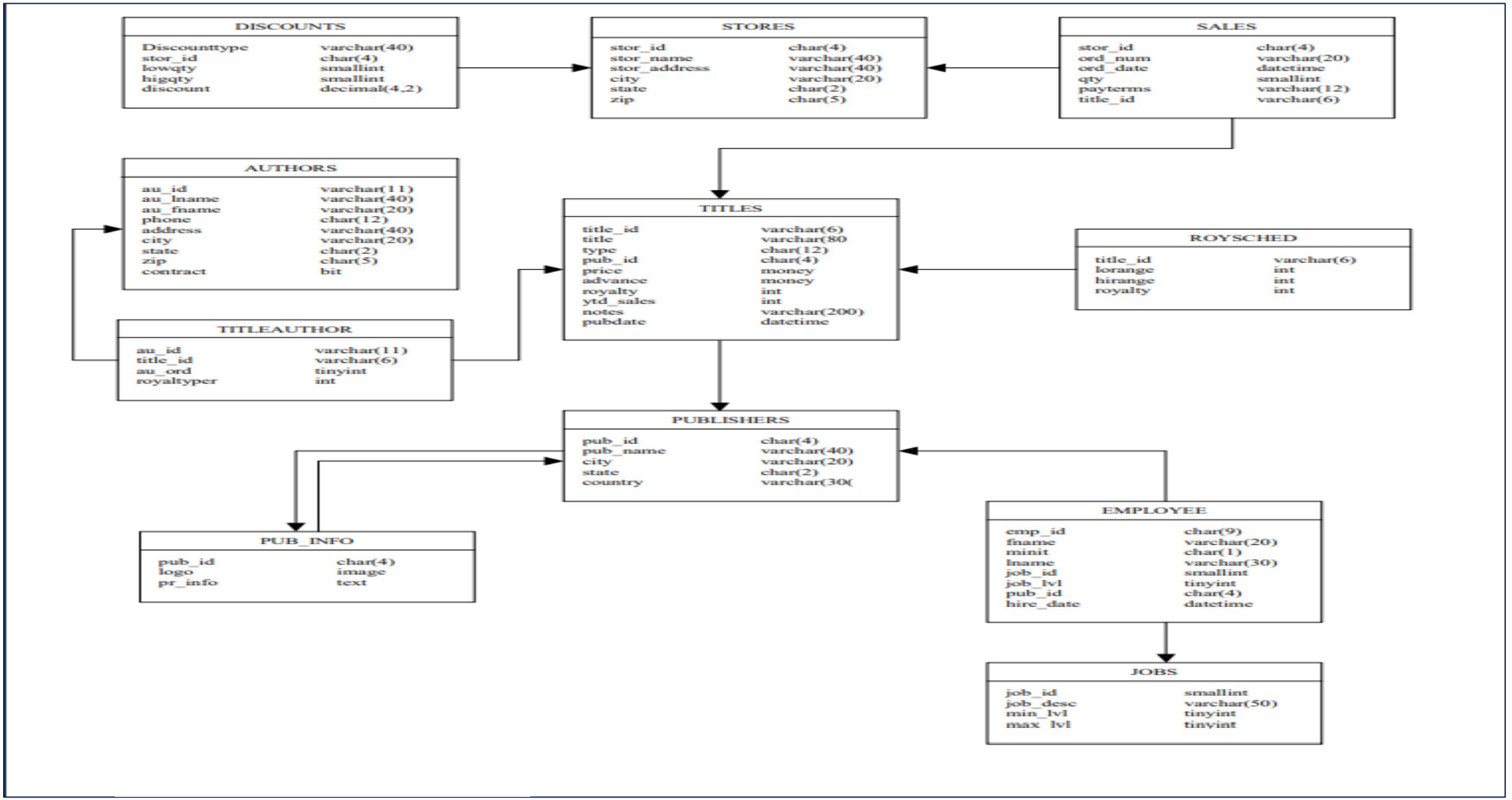
Rules for Identifiers

- The first character must be either a letter or underscore (_), at sign (@), or number sign (#). Although the number sign or double number sign characters can be used to begin the names of other types of objects, it is NOT recommended.
- Some Transact-SQL functions have names that start with double at signs (@@). To avoid confusion with these functions, you should not use names that start with @@.
- Subsequent characters can include letters, decimal numbers, @, $, #, or underscore.
- The identifier must not be a Transact-SQL reserved word. SQL Server reserves both the uppercase and lowercase versions of reserved words.

Data Types

Data Type	Meaning
bigint	Integer data from -2^63 through 2^63-1.
int	Integer data from -2^31 through 2^31 – 1.
smallint	Integer data from -2^15 through 2^15 – 1.
tinyint	Integer data from 0 through 255.
decimal	Fixed precision and scale numeric data from -10^38 +1 through 10^38 –1.
money	Monetary data values from -2^63 through 2^63 - 1 with accuracy to a ten-thousandth of a monetary unit.
datetime	Date and time data from January 1, 1753, through December 31, 9999, with an accuracy of three-hundredths of a second, or 3.33 milliseconds.
char	Fixed-length non-Unicode character data with a maximum length of 8,000 characters. NCHAR is Unicode version.
varchar	Variable-length non-Unicode data with a maximum of 8,000 characters. NVARCHAR is Unicode version.
text	Variable-length non-Unicode data with a maximum length of 2^31 - 1 characters. NTEXT is Unicode version.
binary	Fixed-length binary data with a maximum length of 8,000 bytes.
varbinary	Variable-length binary data with a maximum of 8,000 bytes.
image	Variable-length binary data with a maximum of 2^31-1 bytes.

Pubs Sample Database



Create Table command

```
CREATE TABLE table_name
({ <column_definition > | < table_constraint > } [ ,...n ] )
   <column_definition > ::=
   {column_name data_type }
   [{DEFAULT constant_expression | [ IDENTITY [ ( seed , increment ) ] ] }]
   [<column_constraint > [ ...n ] ]

<column_constraint> ::=
   [ CONSTRAINT constraint_name ]
   { [ NULL | NOT NULL ] | [ PRIMARY KEY | UNIQUE ]
      | REFERENCES ref_table [ ( ref_column ) ]
      [ ON DELETE { CASCADE | NO ACTION } ] [ ON UPDATE { CASCADE | NO ACTION } ]
   }
<table_constraint> ::=
   [ CONSTRAINT constraint_name ]
   { [ { PRIMARY KEY | UNIQUE } { ( column [ ,...n ] ) } ]
   | FOREIGN KEY
     ( column [ ,...n ] )
      REFERENCES ref_table [ ( ref_column [ ,...n ] ) ]
      [ ON DELETE { CASCADE | NO ACTION } ] [ ON UPDATE { CASCADE | NO ACTION } ]
 }
```

Create Table command

```
create table categories
( CatCode varchar(10) constraint categories_pk primary key,
  CatDesc varchar(50) constraint categories_catdesc_nn  not null )

create table products
( ProdId   int  identity (100,1)   constraint products_pk primary key,
  ProdName varchar(30)  constraint products_prodname_nn not null,
  Price    money  constraint products_price_chk check (price>=0),
  Qoh      int  default 0,
  Remarks  varchar(100),
  CatCode  varchar(10) constraint products_catcode_fk
                          references categories(catcode) on delete cascade)

create table sales
( Invno int identity constraint sales_invno_pk primary key,
  Prodid int constraint sales_prodid_fk references products(prodid),
  Transdate datetime,
  Qty  int constraint sales_qty_chk check(qty > 0 ),
  Amount money
);
```

Create Table command

```
insert into categories values ('hd','Hard Disk');
insert into categories values ('pr','Printer');
insert into categories values ('ud','USB Device');
insert into categories values ('mo','Monitor');

insert into products values('Seagate Hard Disk 200 MB',3000,10,'5 years warranty', 'hd');
insert into products values('Transcend Pen Drive 2 GB',500,3,'3 years warranty', 'ud');
insert into products values('HP LaserJet 1020',4500,5,'Low printing cost', 'pr');
insert into products values('HP All in one printer 5445',15000,10,
                            'Print, Scan and Copy', 'pr');


insert into sales values (100,'5/1/22',1,2800);
insert into sales values (101,'5/5/22',1,550);
insert into sales values (101,'5/5/22',2,900);
insert into sales values (102,'5/8/22',1,4500);
insert into sales values (100,'5/15/22',3,8000);
```

SELECT Statement

Retrieves rows from the database and enables the selection of one or many rows or columns from one or many tables.

```
SELECT [ ALL | DISTINCT ]
   [TOP expression [PERCENT]]
   < select_list >
   [ INTO new_table ] [ FROM { <table_source> } [ ,...n ] ]
   [ WHERE <search_condition> ]
   [ GROUP BY [ALL ] group_by_expression [ ,...n ]
   [ HAVING < search_condition > ]
```

```
select prodname, price from products;
select prodname, qty, price, amount =  qty * price from products;
select * from sales where amount > 1000 and qty < 2 order by amount desc;
```

Numeric Functions

Function	Meaning
Abs(number)	Returns absolute value of the number.
Round(number,precision)	Rounds the given number to the specified precision. Negative precision rounds number on the left of decimal point.
Power(base,power)	Raises base to the given power.
Sqrt(number)	Returns square root of the number.
Sign(number)	Returns the positive (+1), zero (0), or negative (-1) sign of the given expression.

Example	Output
Round(10.345,2)	10.350
Power(2,3)	8
Sign(10)	1

String Functions

Function	Meaning
Ascii(char)	Returns ascii code of the given character.
Char(integer)	Returns the character for the given ascii code.
Lower(char)	Converts the given string to lowercase.
Upper(char)	Converts the given string to uppercase.
Len(char)	Returns the length of the string.
Left(char, len)	Returns the len leftmost characters.
Ltrim(char)	Removes leading spaces.
Rtrim(char)	Removes trailing spaces.
Charindex(word, string, start)	Returns index of the word in the string.
Replicate(char, integer)	Replicates the given char for the specified number of times.
Replace(char, source, target)	Replaces every occurrence of source to target.
Reverse(char)	Reverses the string.
Right(char, int)	Returns specified number of characters from right of the string.
Str(number [,len [,decimal]])	Converts number to string.
Stuff(source, spos, length, target)	Replaces a part of source string with target string.
Substring(char, spos, length)	Returns a part of string.

String Functions - Examples

Example	Output
ascii('abc')	97
left('asp.net',3)	asp
charindex('.','asp.net 4.5')	4
charindex('.','asp.net 4.5',5)	10
replicate('*',5)	*****
replace('c#.net','c#','visual basic')	visual basic.net
str(1234.567,7,2)	1234.57
stuff('c#.net',1,2,'vb')	vb.net
substring('asp.net 4.0',4,4)	.net

DateTime functions and Examples

Function	Meaning
DATENAME(datepart, date)	Returns string form of a part of the date.
DATEPART(datepart, date)	Returns value of a part of the date.
GETDATE()	Returns system date.
DATEADD(datepart,units,date)	Adds the specified part to date.
DATEDIFF(datepart, d1, d2)	Returns the difference between two dates in the specified form.
MONTH(date)	Returns month.
DAY(date)	Returns day of the given date.
YEAR(date)	Returns year of the date.

Example	Output
select getdate()	2017-04-10 00:04:24.937
select datename(month,getdate())	April
select datepart(mm,getdate())	4
select dateadd(dd,10,getdate())	2017-04-20 00:06:42.983
select datediff(dd,'1/1/17',getdate())	99

Date Formats

Following are the available data parts that can be used with DATEPART, DATENAME, DATEADD and DATEDIFF functions.

Date Part	Abbreviation
Year	yy
Quarter	qq
Month	mm
Day of year	dy
Day	dd
Week	wk
WeekDay	dw
Hour	hh
Minute	mi
Second	ss
Millisecond	ms

System Formats

System functions return information about the system.

Function	Meaning
HOST_NAME()	Returns name of the host on which SQL Server runs.
USER_NAME()	Returns the name of the current user.
ISNULL(expression, value)	Returns the value when expression is null.
DB_NAME()	Returns the current database name.
@@RowCount	Returns the number of rows affected by the last statement.
@@IDENTITY	Is a system function that returns the last-inserted identity value.
@@ERROR	Returns the error number for the last Transact-SQL statement executed.
@@TRANCOUNT	Returns the number of active transactions for the current connection.

Example	Output
select host_name()	classroom
select db_name()	msdb
select isnull(max(id),0) + 1 from ids	1
select user_name()	dbo

Convert Function

Converts value from one type to another.

```
CONVERT (datatype, expression [, style])
```

Possible values for **style** parameter.

Note: 1, 2, 3 for the same but without century.

Style	Display
101	mm/dd/yyyy
102	yyyy.mm.dd
103	dd/mm/yyyy

Example	Output
select convert(char(10),getdate(),103)	10/04/2017
select convert(char(10),getdate(),3)	10/04/17

Aggregate Functions

The following are the aggregate functions used to get aggregates.

Function	Result
SUM	Total of the values in the numeric expression
AVG	Average of the values in the numeric expression
COUNT	Number of values in the expression
MAX	Highest value in the expression
MIN	Lowest value in the expression
STDDEV	Standard Deviation
VAR	Variance

Grouping Rows

```
select prodid, sum(qty)
from sales
group by prodid
```

```
select prodid, sum(qty)
from sales
where amount > 1000
group by prodid
```

```
select prodid, avg(qty)
from sales
group by prodid
having avg(amount) > 1000
```

Joining Tables

- By using joins, you can retrieve data from two or more tables based on logical relationships between the tables. Joins indicate how SQL Server should use data from one table to select the rows in another table.
- A typical join condition specifies a foreign key from one table and its associated key in the other table.
- Joins are expressed logically using the following Transact-SQL syntax:

- ✓ INNER JOIN
- ✓ LEFT [OUTER] JOIN
- ✓ RIGHT [OUTER] JOIN
- ✓ FULL [OUTER] JOIN
- ✓ CROSS JOIN

```
FROM first_table < join_type > second_table [ ON ( join_condition ) ]
```

Joins Examples

```
select catdesc, prodname from categories c inner join products p
on (c.catcode = p.catcode)
```

```
select catdesc, prodname, price
from categories c inner join products p
on (c.catcode = p.catcode)
where price > 2000
```

```
select prodname, amount from products p left outer join sales s
on (p.prodid = s.prodid)
```

Subquery

- A subquery is a query that is nested inside a SELECT, INSERT, UPDATE, or DELETE statement, or inside another subquery.
- A subquery can be used anywhere an expression is allowed.
- A subquery is also called an inner query or inner select, while the statement containing a subquery is also called an outer query or outer select.
- Statements that include a subquery usually take one of these formats:
  - ✓ WHERE expression [NOT] IN (subquery)
  - ✓ WHERE expression comparison_operator [ANY | ALL] (subquery)
  - ✓ WHERE [NOT] EXISTS (subquery)

Subquery Examples

```
select * from products
where prodid in (select prodid from sales where qty > 2)
```

```
select * from products
where price >
   (select avg(price) from products)
```

```
select * from products as p
where catcode = 'hd'
and prodid not in
   (select prodid from sales)
```

Views

- A view is a virtual table whose contents are defined by a query.
- Like a table, a view consists of a set of named columns and rows of data.
- The rows and columns of data come from tables referenced in the query defining the view and are produced dynamically when the view is referenced.
- A view acts as a filter on the underlying tables referenced in the view.
- The query that defines the view can be from one or more tables or from other views.

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

View Examples

```
create view costlyproducts
as
select * from products
where price > 1000
```

```
select * from costlyproducts
```

```
create view bigsales(product, saledate, amount)
as
select prodname, transdate, amount
from products p join sales s
on p.prodid = s.prodid
where amount > 5000
```

```
select product, amount from bigsales
```

Indexing

- An index is an on-disk structure associated with a table or view that speeds retrieval of rows from the table or view.
- An index contains keys built from one or more columns in the table or view.
- These keys are stored in a structure (B-tree) that enables SQL Server to find the row or rows associated with the key values quickly and efficiently.
- Indexes can be unique, which means no two rows can have the same value for the index key.
- Indexes are automatically maintained for a table or view whenever the table data is modified.
- When this query is executed, the query optimizer evaluates each available method for retrieving the data and selects the most efficient method. The method may be a table scan, or may be scanning one or more indexes if they exist.

```
CREATE [ UNIQUE ] INDEX index_name
    ON <object> (column [ ASC | DESC ] [ ,...n ] )
```

```
CREATE INDEX prodname_index
    ON products (prodname)
```

IF statement

```
IF expression
  Statement
ELSE
  IF expression
     Statement
  ELSE
     . . .
```

IF statement can execute only one statement for either IF or ELSE. However, if you need to execute multiple statements then use BEGIN and END to enclose multiple statements.

```
if @price > 100
 begin
   select @price = @price * 0.90
   select @discount = 'y'
 end
else
  select @discount  = 'n'
```

IF Exists

Returns true if select command returns one or more rows, otherwise false.

```
IF exists(query)
    statement
[ELSE
    statement ]
```

```
if exists(select * from products where catcode = 'hd')
      print "We have some products in category HD"
else
      print "We have no products in category HD"
```

Print Statement

Print statement prints the value of a variable or a global variable or the given text.

```
print {@variable | @@global_variable | text}
```

```
print getdate()
print 'Current Date :' +  convert(char(10),getdate(),3)
```

CASE Function

- Evaluates a list of conditions and returns one of multiple possible result expressions.
- Returns a value depending upon the value of an expression. If none of the conditions is true then it returns value given after ELSE.
- The simple CASE function compares an expression to a set of simple expressions to determine the result.
- The searched CASE function evaluates a set of Boolean expressions to determine the result.
- Both formats support an optional ELSE argument.

```
CASE input_expression
    WHEN when_expression THEN
result_expression
    [ ...n ]
    [ ELSE else_result_expression ]
END
```

```
CASE
    WHEN Boolean_expression THEN
result_expression  [ ...n ]
    [ ELSE else_result_expression ]
END
```

```
select @disrate =
   case @code
       when 1 then 10
       when 2 then 20
       when 3 then 25
       else 5
   end
```

Transaction Management

A transaction is a collection of DML statements that either must be completely executed or must not be executed at all. A transaction represents a single task such as deposit of amount into account, issue of book in library, cancellation of an order etc.

BEGIN TRANSACTION

- Marks the starting point of an explicit, local transaction. BEGIN TRANSACTION increments @@TRANCOUNT by 1.
- If errors are encountered, all data modifications made after the BEGIN TRANSACTION can be rolled back to return the data to this known state of consistency.

```
BEGIN {TRAN | TRANSACTION}
```

Transaction Management

ROLLBACK TRANSACTION

Rolls back an explicit or implicit transaction to the beginning of the transaction, or to a savepoint inside the transaction.

```
ROLLBACK { TRAN | TRANSACTION }
```

COMMIT TRANSACTION

Marks the end of a successful implicit or explicit transaction. Issuing a COMMIT TRANSACTION when @@TRANCOUNT is 0 results in an error; there is no corresponding BEGIN TRANSACTION.

```
COMMIT { TRAN | TRANSACTION }
```

Try… Catch

- Implements error handling similar to exception handling in C#.
- A group of Transact-SQL statements can be enclosed in a TRY block.
- If an error occurs in the TRY block, control is passed to another group of statements that is enclosed in a CATCH block.
- A TRY block must be immediately followed by an associated CATCH block.
- TRY…CATCH constructs can be nested. Either a TRY block or a CATCH block can contain nested TRY…CATCH constructs. For example, a CATCH block can contain an embedded TRY…CATCH construct to handle errors encountered by the CATCH code.
- The TRY…CATCH construct cannot be used in a user-defined function.

```
BEGIN TRY
     { sql_statement | statement_block }
END TRY
BEGIN CATCH
     { sql_statement | statement_block }
END CATCH
[ ; ]
```

System Functions in CATCH Block

In the scope of a CATCH block, the following system functions can be used to obtain information about the error that caused the CATCH block to be executed:

- ERROR_NUMBER() returns the number of the error.
- ERROR_SEVERITY() returns the severity.
- ERROR_STATE() returns the error state number.
- ERROR_PROCEDURE() returns the name of the stored procedure or trigger where the error occurred.
- ERROR_LINE() returns the line number inside the routine that caused the error.
- ERROR_MESSAGE() returns the complete text of the error message. The text includes the values supplied for any substitutable parameters, such as lengths, object names, or times.

Try.. Catch Example

```
BEGIN TRANSACTION;
BEGIN TRY
    -- Generate a constraint violation error.
    DELETE FROM Products WHERE ProdID = 100;
END TRY
BEGIN CATCH
    SELECT
        ERROR_NUMBER() AS ErrorNumber,
        ERROR_SEVERITY() AS ErrorSeverity,
        ERROR_STATE() as ErrorState,
        ERROR_MESSAGE() as ErrorMessage;

    IF @@TRANCOUNT > 0
        ROLLBACK TRANSACTION;
END CATCH;

IF @@TRANCOUNT > 0
    COMMIT TRANSACTION;
```

RaisError Statement

Used to raise a user-defined error. Terminates the program and returns an error to caller.

```
RAISERROR ({Errornumber|Text} [,Severity][,State])
```

Error Number	Is an integer in the range 50000 to 2147483647.
Text	Error message up to 8000 characters.
Severity	Any number in the range 0 through 18. The higher the severity number, the more severe the error is. Severity 19 to 25 indicates more severe system errors.
State	An arbitrary integer in the range 1 through 127 is used to indicate invocation state of the error.

WHILE Statement

- Repeats execution of statement as long as the condition is true.
- **Break** is used to terminate the loop and **Continue** is used to skip remaining portion of the loop and starts next iteration.

```
WHILE condition
      statement
      [BREAK]  [CONTINUE]
```

```
declare @i int
select @i = 1
while @i < 10
  begin
     print @i
     select @i = @i + 1
  end
```

CURSORS

- Cursors are used to take a set of rows and provide the ability to interact with a single row at a time.
- Complete set of rows returned by the SELECT is known as the result set.
- Applications need a mechanism to work with one row or a small block of rows at a time.
- Cursors allow positioning at specific rows of the result set.

Steps in cursor processing

DECLARE Statement

Defines the attributes of a Transact-SQL server cursor, such as its scrolling behavior and the query used to build the result set on which the cursor operates.

```
DECLARE cursor_name [ INSENSITIVE ] [ SCROLL ]
CURSOR
     FOR select_statement
```

OPEN Statement

Opens a Transact-SQL server cursor and populates the cursor by executing the Transact-SQL statement specified on the DECLARE CURSOR.

```
OPEN { { [ GLOBAL ] cursor_name } |
cursor_variable_name }
```

Steps in cursor processing

Fetch Statement

Retrieves a specific row from the specified cursor.

```
FETCH   [ [ NEXT | PRIOR | FIRST | LAST | ABSOLUTE { n | @nvar }
                    | RELATIVE { n | @nvar } ]
FROM  cursor_name  [ INTO @variable_name [ ,...n ] ]
```

CLOSE statement

Closes an open cursor by releasing the current result set and freeing any cursor locks held on the rows on which the cursor is positioned.

```
CLOSE { { [ GLOBAL ] cursor_name } | cursor_variable_name }
```

DEALLOCATE statement

Removes a cursor reference. When the last cursor reference is deallocated, the data structures comprising the cursor are released by Microsoft SQL Server.

```
DEALLOCATE { { [ GLOBAL ] cursor_name } | cursor_variable_name }
```

Cursor Example

```
declare products_cursor cursor for select prodname from  dbo.products;
declare @prodname varchar(50);

open products_cursor;
fetch next from products_cursor into @prodname;

while @@fetch_status = 0
 begin
   print @prodname;
   fetch from products_cursor into @prodname;
 end;

close products_cursor;
deallocate products_cursor;
```

Stored Procedures

- Stored procedures are procedures that are stored in database.
- They are invoked from client by just giving the name.
- A procedure may or may not take parameters.
- If it takes parameters, they may be in, out or in out parameters. IN parameters provide values to procedure, OUT parameters are used by procedure to provide values to caller.
- IN OUT parameters allow both.
- Procedure can optionally return exit code using return statement or a collection of rows.
- Create Procedure statement creates a procedure. A stored procedure is a saved collection of Transact-SQL statements.
- It is possible to call a stored procedure using EXECUTE statement.

```
CREATE PROCEDURE  [schema_name.] procedure_name
    [ { @parameter [ type_schema_name. ] data_type }
      [ VARYING ] [ = default ] [ OUT | OUTPUT ]
    ] [ ,...n ]
AS
sql_statements
```

```
EXECUTE procedure
```

Stored Procedures Examples

```
CREATE PROCEDURE dbo.ProductDetails
AS
    select prodid, prodname,price, qoh, catdesc
    from  products p inner join categories c on ( p.catcode = c.catcode)
    order by c.catcode, prodid
```

```
CREATE PROCEDURE dbo.ChangePrice(@prodid int, @newprice money)
AS
 if exists( select * from products where prodid = @prodid)
  begin
    update products set price = @newprice where prodid = @prodid
    return 0 /* success */
  end
else
  return 1  /* prodid not found */
```

```
CREATE PROCEDURE dbo.ChangePrice2(@prodid int, @newprice money)
AS
    update products set price = @newprice where prodid = @prodid
    if @@rowcount = 0
       raiserror('Invalid Product ID',16,1)
```

OUT Parameters Example

The following example shows how to create a stored procedure that returns highest and lowest prices for products of the given category.

```
create procedure GetHighLowPrices(@catcode varchar(10), @highprice money output,
                                  @lowprice money output) as
 select @highprice =  max(price), @lowprice = min(price)
 from products where catcode = @catcode;
```

The following code calls the procedure and prints values returned by procedure through out parameters.

```
declare @highprice money, @lowprice money
execute GetHighLowPrices 'pr', @highprice output, @lowprice output;
print @highprice
print @lowprice
```

Stored Functions

A function is similar to procedure but it must return a value.

```
create function  function_name
   ([@parameter_name as scalar_parameter_data_type ,...n ])  returns data_type  [as]
begin
    function_body
    return scalar_expression
end
```

The following function returns total sales amount for the given product.

```
create function TotalSales(@prodid int) returns money as
begin
 declare @total as money
 select @total =  sum(amount) from sales where prodid = @prodid;
 return @total;
end;
```

At the time of calling a function you must use name of the owner of the function followed by function name.

```
select prodname, dbo.TotalSales(prodid) from products
```

Database Trigger

- A trigger is a T-SQL program that is automatically executed on an event.
- A trigger is a special kind of stored procedure that automatically executes when an event occurs in the database server.
- DML triggers execute when a user tries to modify data through a data manipulation language (DML) event. DML events are INSERT, UPDATE, or DELETE statements on a table or view.
- Triggers are called as post-filters as they are executed after data is written.
- Triggers are used to enforce business rules that cannot be enforced using constraints such as Primary key, Check etc.
- DML triggers use the **deleted** and **inserted** logical (conceptual) tables. They are structurally similar to the table on which the trigger is defined, that is, the table on which the user action is tried. The **deleted** and **inserted** tables hold the old values and new values of the rows that may be changed by the user action.

```
CREATE TRIGGER [schema_name.]trigger_name
ON {table | view }
{FOR | AFTER | INSTEAD OF }
{[ INSERT ] [ , ] [ UPDATE ] [ , ] [ DELETE ] }
AS { sql_statement }
```

Database Trigger - Example

```
create trigger trg_sales_qty_check
on sales
after update
as
declare @oldqty int, @newqty int

select @oldqty = qty from deleted
select @newqty = qty from inserted
if @oldqty != @newqty
 begin
   rollback transaction
   raiserror('Cannot change quantity of a sale transaction',16,1);
end
```