Table of Contents

Background: What is SQL? Why do we need it? 3

Identifying Data Types 3

Managing Tables 5

Manipulating Data 8

Retrieving Attributes 10

JOINS 14

Subqueries 17

Using Functions to Customize ResultSet 19

GROUPING DATA 23

Determine if object exists 25

add Check Constraint 25

T-SQL Processing Order 25

Window Function 25

Casting 25

Case Expressions 25

String Functions 25

Wildcards 26

Date Functions 26

Metadata Queries 26

CROSS JOIN 27

Derived Tables 27

Common Table Expression's / CTE's 27

Recursive CTE 28

Correlated Queries 29

EXISTS 29

Views 29

Apply Operator 30

SET Operators 30

PIVOT 30

UNPIVOT 31

Select from VALUES 31

Grouping Sets 31

Group by cube 32
Insert from Select 32
Insert from Sproc 32
Select into a new table 32
Bulk Insert 33
Last Identity 33
DELETE 33
UPDATE based on join 34
MERGE 34
OUTPUT clause 34
Transactions 35
Detailed Transaction Template 35
Locking 36
Variables 36
Flow Control 37
Cursors 37
Min Key approach 38
Temporary Tables 38
Table Variables 38
Table Types 38
Dynamic SQL 39
User Defined Functions 39
Stored Procedures 40
Triggers 40
Try / Catch 41
Template for basic table + relationships 41
Identity Insert 41

Background: What is SQL? Why do we need it?

SQL is a database language used to query and manipulate the data in the database.

Main objectives:

- To provide an efficient and convenient environment
- Manage information about users who interact with the DBMS

The SQL statements can be categorized as

Data Definition Language(DDL) Commands:

- CREATE: creates a new database object, such as a table.
- ALTER: used to modify the database object
- DROP: used to delete the objects.

Data Manipulation Language(DML) Commands:

- INSERT: used to insert a new data row record in a table.
- UPDATE: used to modify an existing record in a table.
- DELETE: used delete a record from the table.

Data Control Language(DCL) Commands:

- GRANT: used to assign permission to users to access database objects.
- REVOKE: used to deny permission to users to access database objects.

Data Query Language(DQL) Commands:

- SELECT: it is the DQL command to select data from the database.

Data Transfer Language(DTL) Commands:

- COMMIT: used to save any transaction into the database permanently.
- ROLLBACK: restores the database to the last committed state.

Identifying Data Types

Data types specify the type of data that an object can contain, such as integer data or character data. We need to specify the data type according to the data to be stored.

Following are some of the essential data types:

Data Type	Used to Store
int	Integer data
smallint	Integer data
tinyint	Integer data
bigint	Integer data
decimal	Numeric data type with a fixed precision and scale.
numeric	numeric data type with a fixed precision and scale.
float	floating precision data
money	monetary data
datetime	data and time data
char(n)	fixed length character data
varchar(n)	variable length character data
text	character string
bit	integer data with 0 or 1
image	variable length binary data to store images
real	floating precision number
binary	fixed length binary data
cursor	cursor reference
sql_variant	different data types

timestamp	unique number in the database that is updated every time in a row that contains timestamp is inserted or updated.
table	temporary set of rows returned as a result set of a table-valued function.
xml	store and return xml values

Managing Tables

Create Table

Table can be created using the CREATE TABLE statement. The syntax is as follows:

```
CREATE TABLE table_name

( col_name1 datatype,

col_name2 datatype,

col_name3 datatype,

...

);
```

Example: Create a table named EmployeeLeave in Human Resource schema with the following attributes:

Columns	Data Type	Checks
EmployeeID	int	NOT NULL
LeaveStartDate	date	NOT NULL
LeaveEndDate	date	NOT NULL
LeaveReason	varchar(100)	NOT NULL

LeaveType	char(2)	NOT NULL

```
CREATE TABLE HumanResources.EmployeeLeave
(
EmployeeID int NOT NULL,
LeaveStartDate datetime NOT NULL,
LeaveEndDate datetime NOT NULL,
LeaveReason varchar(100),
LeaveType char(2) NOT NULL );
```

Constraints in SQL

Constraints define rules that must be followed to maintain consistency and correctness of data. A constraint can be created by using either of the following statements:

```
CREATE TABLE statement
ALTER TABLE statement
CREATE TABLE table_name
(
column_name CONSTRAINT constraint_name constraint_type
)
```

Types of Constraints:

Constraint	Description	Syntax
Primary key	Columns or columns that uniquely identify all rows in the table.	CREATE TABLE table_name (col_name [CONSTRAINT constraint_name PRIMARY KEY] (col_name(s)))

Unique key	Enforces uniqueness on non primary key columns.	**CREATE TABLE** table_name (col_name [**CONSTRAINT** constraint_name **UNIQUE KEY**] (col_name(s)))
Foreign key	Is used to remove the inconsistency in two tables when the data depends on other tables.	**CREATE TABLE** table_name (col_name [**CONSTRAINT** constraint_name **FOREIGN KEY**] (col_name) **REFERENCES** table_name (col_name))
Check	Enforce domain integrity by restricting the values to be inserted in the column.	**CREATE TABLE** table_name (col_name [**CONSTRAINT** constraint_name] **CHECK** (expression) (col_name(s))) expression: **IN**, **LIKE**, **BETWEEN**

3.2 Modifying Tables

Modify table using ALTER TABLE statement when:

1. Adding column
2. Altering data type
3. Adding or removing constraints

Syntax of ALTER TABLE:

```
ALTER TABLE table_name

ADD column_name;

ALTER TABLE table_name
```

```
DROP COLUMN column_name;

ALTER TABLE table_name

ALTER COLUMN column_name data_type;
```

Renaming a Table

A table can be renamed whenever required using RENAME TABLE statement:

RENAME TABLE old_table_name TO new_table_name;

Dropping a Table versus Truncate Table

A table can be dropped or deleted when no longer required using DROP TABLE statement:

```
DROP TABLE table_name;
```

The contents of the table can be deleted when no longer required without deleting the table itself using TRUNCATE TABLE statement:

```
TRUNCATE TABLE table_name;
```

Manipulating Data

Storing Data in a Table

Syntax:

```
INSERT INTO table_name (col_name1, col_name2, col_name3…)

VALUES (value1, value2, value3…);
```

Example: Inserting data into Student table.

```
INSERT INTO Student (StudentID, FirstName,LastName,Marks)

VALUES ('101','John','Ray','78');
```

Example: Inserting multiple data into Student table.

```
INSERT INTO Student
```

```
VALUES (101,'John','Ray',78),
 (102,'Steve','Jobs',89),
 (103,'Ben','Matt',77),
 (104,'Ron','Neil',65),
 (105,'Andy','Clifton',65),
 (106,'Park','Jin',90);
```

Copying Data from one table to another:

```
INSERT INTO table_name2
SELECT * FROM table_name1
WHERE [condition]
```

Updating Data in a Table

Data can be updated in the table using UPDATE DML statement:

```
SELECT table_name
SET col_name1 = value1 , col_name2 = value2...
WHERE condition
```

Example update marks of Andy to 85

```
SELECT table_name
SET Marks = 85
WHERE FirstName = 'Andy'
```

Deleting Data from a Table

A row can be deleted when no longer required using DELETE DML statement.

Syntax:

```
DELETE FROM table_name
```

```
WHERE condition

DELETE FROM Student

WHERE StudentID = '103'
```

Deleting all records from a table:

```
DELETE table_name
```

Retrieving Attributes

One or more column can be displayed while retrieving data from the table.

One may want to view all the details of the Employee table or might want to view few columns.

Required data can be retrieved data from the database tables by using the SELECT statement.

The syntax of SELECT statement is:

```
SELECT [ALL | DISTINCT] select_column_list

[INTO [new_table_name]]

[FROM [table_name | view_name]]

[WHERE search condition]
```

Consider the following Student table:

StudentID	FirstName	LastName	Marks
101	John	Ray	78
102	Steve	Jobs	89
103	Ben	Matt	77
104	Ron	Neil	65

105	Andy	Clifton	65
106	Park	Jin	90

Retrieving Selected Rows

To retrieve selected rows from a table use WHERE clause in the SELECT statement.

```
SELECT *
FROM Student
WHERE StudentID = 104;
```

HAVING Clause is used instead of WHERE for aggregate functions.

Comparison Operators

Comparison operators test for the similarity between two expressions.

Syntax:

```
SELECT column_list
FROM table_name
WHERE expression1 comparison_operatore expression2
```

Example of some comparison operators:

```
SELECT StudentID,Marks
FROM Student
WHERE Marks = 90;
SELECT StudentID,Marks
FROM Student
WHERE StudentID > 101;
SELECT StudentID,Marks
FROM Student
```

```
WHERE Marks != 89;

SELECT StudentID,Marks

FROM Student

WHERE Marks >= 50;
```

Logical Operators

Logical operators are used to SELECT statement to retrieve records based on one or more conditions. More than one logical operator can be combined to apply multiple search conditions.

Syntax:

```
SELECT column_list

FROM table_name

WHERE conditional_expression1 [NOT]

 conditional_expression2
```

Types of Logical Operators:

OR Operator

```
SELECT StudentID,Marks,

FROM Student

WHERE Marks= 40 OR Marks=56 OR Marks = 65;
```

AND Operator

```
SELECT StudentID,Marks,

FROM Student

WHERE Marks= 89 AND Marks=56 AND Marks = 65;
```

NOT Operator

```
SELECT StudentID,Marks,
```

```
FROM Student

WHERE NOT LastName = “Jobs”;
```

Range Operator

Range operator retrieves data based on range.

Syntax:

```
SELECT column_name1, col_name2….

FROM table_name

WHERE expression1 range_operator expression2 AND expression3
```

Types of Range operators:

BETWEEN

```
SELECT StudentID,Marks

FROM Student

WHERE Marks BETWEEN 40 AND 70;
```

NOT BETWEEN

```
SELECT FirstName,Marks,

FROM Student

WHERE Marks NOT BETWEEN 40 AND 50;
```

Retrieve Records That Match a Pattern

Data from the table can be retrieved that match a specific pattern.

The LIKE keyword matches the given character string with a specific pattern.

```
SELECT *

FROM Student

WHERE FirstName LIKE ‘Ro%’
```

```
SELECT *

FROM Student

WHERE FirstName LIKE '_e%'
```

Displaying in a Sequence

Use ORDER BY clause to display the data retrieved in a specific order.

```
SELECT StudentID, LastName,

FROM Student

ORDER BY Marks DESC;
```

Displaying without Duplication

The DISTINCT keyword is used to eliminate rows with duplicate values in a column.

Syntax:

```
SELECT [ALL | DISTINCT] col_names

FROM table_name

WHERE search_condition

SELECT DISTINCT Marks

FROM Student

WHERE LastName LIKE 'o%';
```

JOINS

Joins are used to retrieve data from more than one table together as a part of a single result set. Two or more tables can be joined based on a common attribute.

Types of JOINS:

Consider two tables Employees and EmployeeSalary

EmployeeID (PK)	FirstName	LastName	Title

1001	Ron	Brent	Developer
1002	Alex	Matt	Manager
1003	Ray	Maxi	Tester
1004	August	Berg	Quality

EmployeeID (FK)	Department	Salary
1001	Application	65000
1002	Digital Marketing	75000
1003	Web	45000
1004	Software Tools	68000

INNER JOIN

An inner join retrieves records from multiple tables by using a comparison operator on a common column.

Syntax:

```
SELECT column_name1,colomn_name2, ...
FROM table1 INNER JOIN table2
ON table1.column_name = table2.column_name
```

Example:

```
SELECT e.LastName, e.Title, es.salary,
FROM e.Employees INNER JOIN es.EmployeeSalary
ON e.EmployeeID = es.EmployeeID
```

OUTER JOIN

An outer join displays the resulting set containing all the rows from one table and the matching rows from another table.

An outer join displays NULL for the column of the related table where it does not find matching records.

Syntax:

```
SELECT column_name1,colomn_name2, ...

FROM table1 [LEFT|RIGHT|FULL]OUTER JOIN table2

ON table1.column_name = table2.column_name
```

Types of Outer Join

LEFT OUTER JOIN: In left outer join all rows from the table on the left side of the LEFT OUTER JOIN keyword is returned, and the matching rows from the table specified on the right side are returned the result set.

Example:

```
SELECT e.LastName, e.Title, es.salary,

FROM e.Employees LEFT OUTER JOIN es.EmployeeSalary

ON e.EmployeeID = es.EmployeeID
```

RIGHT OUTER JOIN: In right outer join all rows from the table on the right side of the RIGHT OUTER JOIN keyword are returned, and the matching rows from the table specified on the left side are returned is the result set.

Example:

```
SELECT e.LastName, e.Title, es.salary,

FROM e.Employees LEFT OUTER JOIN es.EmployeeSalary

ON e.EmployeeID = es.EmployeeID
```

FULL OUTER JOIN: It is a combination of left outer join and right outer join. This outer join returns all the matching and non-matching rows from both tables. Whilst, the matching records are displayed only once.

Example:

```
SELECT e.LastName, e.Title, es.salary,

FROM e.Employees FULL OUTER JOIN es.EmployeeSalary

ON e.EmployeeID = es.EmployeeID
```

CROSS JOIN

Also known as the Cartesian Product between two tables joins each row from one table with each row of another table. The rows in the result set is the count of rows in the first table times the count of rows in the second table.

Syntax:

```
SELECT column_name1,colomn_name2,column_name1 + column_name2 AS new_column_name

FROM table1 CROSS JOIN table2
```

EQUI JOIN

An Equi join is the same as inner join and joins tables with the help of foreign key except this join is used to display all columns from both tables.

SELF JOIN

In self join, a table is joined with itself. As a result, one row is in a table correlates with other rows in the same table. In this join, a table name is mentioned twice in the query. Hence, to differentiate the two instances of a single table, the table is given two aliases. Syntax:

```
SELECT t1.c1,t1.c2 AS column1,t2.c3,t2.c4 AS column2

FROM table1 t1 JOIN table2 t2

WHERE condition
```

Subqueries

An SQL statement that is used inside another SQL statement is termed as a subquery.

They are nested inside WHERE or HAVING clause of SELECT, INSERT, UPDATE and DELETE statements.

- Outer Query: Query that represents the parent query.
- Inner Query: Query that represents the subquery.

Using IN Keyword

If a subquery returns more than one value, we might execute the outer query if the values within the columns specified in the condition match any value in the result set of the subquery.

Syntax:

```
SELECT column, column

FROM table_name

WHERE column [NOT] IN

(SELECT column

 FROM table_name [WHERE conditional_expression] )
```

Using EXISTS Keyword

EXISTS clause is used with subquery to check if a set of records exists.

TRUE value is returned by the subquery in case if the subquery returns any row.

Syntax:

```
SELECT column, column

FROM table_name

WHERE EXISTS

(SELECT column_name FROM table_name WHERE condition)
```

Using Nested Subqueries

A subquery can contain more than one subqueries. Subqueries are used when the condition of a query is dependent on the result of another query, which is, in turn, is dependent on the result of another subquery.

Syntax:

```
SELECT column, column

FROM table_name

WHERE column_name expression_operator
```

```
(SELECT column_list FROM table_name

 WHERE column_name expression_operator

 (SELECT column_list FROM table_name

 WHERE [condition] ) )
```

Correlated Subquery

A correlated subquery can be defined as a query that depends on the outer query for its evaluation.

Using Functions to Customize ResultSet

Various in-built functions can be used to customize the result set.

Syntax:

```
SELECT function_name (parameters)
```

Using String Functions

String values in the result set can be manipulated by using string functions.

They are used with char and varchar data types.

Following are the commonly used string functions are:

Function Name	Example
left	`SELECT left ('RICHARD' ,4)`
len	`SELECT len ('RICHARD')`

lower	SELECT lower ('RICHARD')
reverse	SELECT reverse ('ACTION')
right	SELECT right ('RICHARD' ,4)
space	SELECT 'RICHARD' + space(2) + 'HILL'
str	SELECT str (123.45,6,2)
substring	SELECT substring ('Weather' ,2,2)
upper	SELECT upper ('RICHARD')

Using Date Functions

Date functions are used to manipulate date time values or to parse the date values.

Date parsing includes extracting components, such as day, month, and year from a date value.

Some of the commonly used date functions are:

Function Name	Parameters	Description
dateadd	(date part, number, date)	Adds the number of date parts to the date.
datediff	(date part, date1, date2)	Calculates the number of date parts between two dates.
Datename	(date part, date)	Returns date part from the listed as a character value.
datepart	(date part, date)	Returns date part from the listed as an integer.
getdate	0	Returns current date and time
day	(date)	Returns an integer, which represents the day.
month	(date)	Returns an integer, which represents the month.
year	(date)	Returns an integer, which represents the year.

Using Mathematical Functions

Numeric values in a result set can be manipulated in using mathematical functions.

The following table lists the mathematical functions:

Function Name	Parameters	Description
abs	(numeric_expression)	Returns an absolute value
acts,asin,atan	(float_expression)	Returns an angle in radians
cos, sin,	(float_expression)	Returns the cosine, sine, cotangent, or tangent of the angle in

cot,tan		radians.
degrees	(numeric_expression)	Returns the smallest integer greater than or equal to specifies value.
exp	(float_expression)	Returns the exponential value of the specified value.
floor	(numeric_expression)	Returns the largest integer less than or equal to the specified value.
log	(float_expression)	Returns the natural logarithm of the specified value.
pi	0	Returns the constant value of 3.141592653589793
power	(numeric_expression,y)	Returns the value of numeric expression to the value of y
radians	(numeric_expression)	Converts from degrees to radians.
rand	([seed])	Returns a random float number between 0 and 1.
round	(numeric_expression,length)	Returns a numeric expression rounded off to the length specified as an integer expression.
sign	(numeric_expression)	Returns positive, negative or zero.
sqrt	(float_expression)	Returns the square root of the specified value.

Using Ranking Functions

Ranking functions are used to generate sequential numbers for each row to give a rank based on specific criteria.

Ranking functions return a ranking value for each row. Following functions are used to rank the records:

- row_number Function: This function returns the sequential numbers, starting at 1, for the rows in a result set based on a column.
- rank Function: This function returns the rank of each row in a result set based on specified criteria.
- dense_rank Function: The dense_rank() function is used where consecutive ranking values need to be given based on specified criteria.

These functions use the OVER clause that determines the ascending or descending sequence in which rows are assigned a rank.

Using Aggregate Functions

The aggregate functions, on execution, summarize the values for a column or group of columns and produce a single value.

Syntax:

```
SELECT aggrgate_function ([ALL | DISTINCT] expression)

FROM table_name
```

Following are the aggregate functions:

Function Name	Description
avg	returns the average of values in a numeric expression, either all or distinct.
count	returns the number of values in an expression, either all or distinct.
min	returns the lowest value in an expression.
max	returns the highest value in an expression.
sum	returns the total of values in an expression, either all or distinct.

GROUPING DATA

Grouping data means to view data that match a specific criteria to be displayed together in the result set.

Data can be grouped by using GROUP BY, COMPUTE,COMPUTE BY and PIVOT clause in the SELECT statement.

GROUP BY Clause

Summarizes the result set into groups as defined in the query by using aggregate functions.

Syntax:

```
SELECT column_list
```

```
FROM table_name

WHERE condition

[GROUP BY [ALL] expression]

[HAVING search_condition]
```

COMPUTE and COMPUTE BY Clause

This COMPUTE clause, with the SELECT statement, is used to generate summary rows by using aggregate functions in the query result.

The COMPUTE BY clause can be used to calculate summary values of the result set on a group of data.

Syntax:

```
SELECT column_list

FROM table_name

ORDER BY column_name

COMPUTE aggregate_function (column_name)

[BY column_name]
```

PIVOT Clause

The PIVOT operator is used to transform a set of columns into values, PIVOT rotates a table-valued expression by turning the unique values from one column in the expression into multiple columns in the output.

Syntax:

```
SELECT *

FROM table_name

PIVOT (aggregate function (value_column)

 FOR pivot_column

 IN (column_list)

) table_alias
```

Determine if object exists

```
IF OBJECT_ID('product.Model', 'U') IS NOT NULL
```

add Check Constraint

```
ALTER TABLE dbo.MyTable
   ADD CONSTRAINT CHK_dbo_MyTable_Value
   CHECK (VALUE > 0.00)
```

T-SQL Processing Order

1. From
2. Where
3. Group by
4. Having
5. Select
6. Order By

Window Function

```
SELECT PurchaseOrderID, ItemCode, subtotal,
 ROW_NUMBER() OVER ( PARTITION BY PurchaseOrderID ORDER BY ItemCode) AS rownum,
 SUM(subtotal) OVER (PARTITION BY PurchoseOrderID) AS purchaseOrderTotal
FROM pur.PurchaseOrderDetail
```

Casting

```
SELECT CAST('12345' AS NUMERIC(12,2))
```

Case Expressions

```
SELECT Vendor_Code =
      CASE
         WHEN VendorItemCode IS NULL THEN ''
         WHEN LEN(VendorItemCode) > 10 THEN LEFT(VendorItemCode, 10)
         ELSE VendorItemCode
      END
FROM pur.PurchaseOrderDetail
```

String Functions

```
select LEN('sql server') -- 10
```

```
select CHARINDEX('e','sql server') -- 6
select PATINDEX('%serv%', 'sql server') -- 5
select REPLACE('sql server', 'sql', 'cookie') -- cookie server
select REPLICATE('sql', 4) -- sqlsqlsqlsql
select STUFF('sql server', 1, 0, 'Microsoft ') -- Microsoft sql server
```

Wildcards

```
SELECT ManufacturerCode
FROM  product.Model
WHERE ManufacturerCode
LIKE 'PG-42%' --PG-42445-01 PG-42600-02
LIKE '%G-42%'--  PG-42445-01 RG-42900-03
LIKE 'RG-_____-__' -- RG-85000-01 RG-42900-03
LIKE 'RG-[8-9]____-__' --  RG-85000-01, RG-95000-01
LIKE '[O-Z]G%' -- RG, PG, but not AG, FG, etc.
```

Date Functions

```
select GETDATE() -- 2014-01-17 07:45:59.730
select DATEADD(year, 1, getdate()) --2015-01-17 07:45:59.730
select DATEADD(month, 1, getdate())-- 2014-02-17 07:45:59.730
select DATEADD(day, 1, getdate()) -- 2014-01-18 07:45:59.730

select DATEDIFF(year,  '20130101', '20131024') -- 0
select DATEDIFF(month,  '20130101', '20131024') -- 9
select DATEDIFF(day,  '20130101', '20131024') -- 296

select DATEPART(year, getdate()) -- 2014
select DATEPART(month, getdate()) -- 1
select DATEPART(day, getdate()) -- 17

select YEAR(GETDATE()) -- 2014
select MONTH(GETDATE()) -- 1
select DAY(getdate()) -- 17

select DATENAME(month, getdate()) -- January
select DATENAME(DAY, GETDATE()) -- 17

select ISDATE('20130101') - 1
select ISDATE('20139999') - 0
```

Metadata Queries

```
USE BikeStore
GO

SELECT SCHEMA_NAME(SCHEMA_ID) AS table_schema_name, name AS table_name FROM sys.tables ORDER
BY table_schema_name, table_name
```

```
SELECT name
FROM sys.columns
WHERE OBJECT_ID = OBJECT_ID('product.Category')


SELECT * FROM INFORMATION_SCHEMA.TABLES
SELECT * FROM INFORMATION_SCHEMA.TABLE_PRIVILEGES -- table specific privileges granted to accounts
SELECT * FROM INFORMATION_SCHEMA.VIEW_TABLE_USAGE -- tables referenced by views

EXEC sys.sp_tables
EXEC sys.sp_help @objname = N'product.Model' -- returns general info about the object


SELECT SERVERPROPERTY('ProductLevel') -- current value is 'SP1'
SELECT SERVERPROPERTY('Edition') -- Standard Edition (64-bit)
SELECT @@VERSION -- Microsoft SQL Server 2008 R2 (SP1) - 10.50.2550.0 (X64)   Jun 11 2012 16:41:53   Copyri
ght (c) Microsoft Corporation  Standard Edition (64-bit) on Windows NT 6.1 <X64> (Build 7601: Service Pack 1) (
Hypervisor)
SELECT DATABASEPROPERTYEX('enterprise', 'Collation') -- SQL_Latin1_General_CP1_CI_AS
SELECT OBJECTPROPERTY(OBJECT_ID('product.PartNumber), 'TableHasPrimaryKey') -- 1
```

CROSS JOIN

```
CROSS JOIN
   DECLARE @digits TABLE (digit INT)
   INSERT INTO @digits (digit) values (1),(2), (3)

   SELECT  d2.digit, d1.digit -- returns 9 record result
      FROM @digits d1
         CROSS JOIN @digits d2
```

Derived Tables

```
   SELECT SalesYear, SUM(LineTotal) AS TotalSold
   FROM (
         SELECT YEAR(DateSold) AS SalesYear, LineTotal
         FROM sales.SkuSales
      ) SalesByYear
   GROUP BY SalesYear
   ORDER BY TotalSold DESC
```

Common Table Expression's / CTE's

```
   WITH <CTEname> (<column1>, <column2>) -- column list optional
   as
   (
      <subquery>
   )
   <outer query>
```

```
WITH SalesByYearCTE ( SalesYear, LineTotal ) -- list of columns optional
AS
(
   SELECT YEAR(DateSold) AS SalesYear, LineTotal
      FROM sales.SkuSales
) -- to add more CTE's, add a comma here
SELECT SalesYear, SUM(LineTotal) AS TotalSold
FROM SalesByYearCTE
GROUP BY SalesYear
ORDER BY TotalSold DESC
```

Recursive CTE

useful for querying tables that are self-referencing

```
declare @Location_Table table
   (
      Location_ID int,
      Location_Name varchar(25),
      Location_Parent int null
   )

   insert into @Location_Table (Location_ID, Location_Name, Location_Parent)
   select 1, 'United States', null union all
   select 2, 'Iowa', 1 union all
   select 3, 'South Dakota', 1 union all
   select 4, 'Minnesota', 1 union all
   select 5, 'Nebraska', 1 union all
   select 6, 'Orange City', 2 union all
   select 7, 'Sioux Center', 2 union all
   select 8, 'Hospers', 2 union all
   select 9, 'Sioux Falls', 3 union all
   select 10, 'Brookings', 3 union all
   select 11, '102 Michigan Ave SW', 6 union all
   select 12, '412 4th St SE', 6 union all
   select 13, 'Utility Room', 11 union all
   select 14, 'Kitchen', 11 union all
   select 15, 'Chest Freezer', 13;


   with Location_CTE as
   (
      -- anchor
      select location_ID, location_Name, Location_Parent
      from @Location_Table where Location_ID = 2 --  iowa

      union all

      -- recurse
      select lt.Location_ID, lt.Location_Name, lt.Location_Parent
      from Location_CTE lc
         join @Location_Table lt
         on lt.Location_Parent = lc.Location_ID
```

```
    )
    select Location_ID, Location_Name,
    (select Location_Name from @Location_Table where Location_ID = Location_CTE.Location_Parent) as loc
ation_parent_name
    from Location_CTE;
```

Correlated Queries

```
  -- inside WHERE clause
  SELECT TOP 1000 BillOfMaterialsID, BillOfMaterialsRevisionID, Quantity
  FROM manufacturing.BillOfMaterials bom_outer
  WHERE BillOfMaterialsID =
  (
    SELECT TOP 1 BillOfMaterialsID
    FROM manufacturing.BillOfMaterial  bom_inner
    WHERE bom_inner.BillOfMaterialsRevisionID = bom_outer.BillOfMaterialsRevisionID -- outer reference
    ORDER BY Quantity DESC
  )

  -- inside SELECT clause
  SELECT BillOfMaterialsID, BillOfMaterialsRevisionID, MaterialID, Quantity,
    (Amount / (SELECT SUM(Quantity) FROM manufacturing.BillOfMaterials bom_inner WHERE bom_inner.Bil
lOfMaterialsRevisionID  = bom_outer.BillOfMaterialsRevisionID) ) * 100
      AS percent_of_recipe
  FROM manufacturing.BillOfMaterialsl bom_outer
  WHERE BillOfMaterialsRevisionID = 10004
```

EXISTS

```
SELECT mfg.Name
FROM product.Manufacturer mfg
WHERE  NOT EXISTS ( SELECT * FROM pur.PurchaseOrder po WHERE po.ManufacturerID = mfg.ManufacturerI
D)
```

Views

```
CREATE VIEW PUR.ViewManufacturersWithPurchases
AS
  SELECT mfg.Name
          FROM product.Manufacturer mfg
          WHERE  NOT EXISTS ( SELECT * FROM pur.PurchaseOrder po WHERE po.ManufacturerID = mfg.Ma
nufacturerID)
GO

SELECT * FROM PUR.ViewManufacturersWithPurchases
```

Apply Operator

```
-- Query 5 most recent orders for each active product
SELECT a_left.PartNumberID, p_right.DateSold, p_right.LineTotal
FROM product.PartNumber a_left
CROSS APPLY -- replace with OUTER APPLY to include products from left that do not have any batches
(
      SELECT TOP(5) PartNumberID, DateSold, LineTotal
      FROM sales.SkuSales AS p_inner
      WHERE p_inner.PartNumberID = a_left.PartNumberID
      ORDER BY DateSold DESC
   ) AS p_right
WHERE a_left.StatusID = 1
ORDER BY a_left.PartNumberID, p_right.DateSold DESC
```

SET Operators

```
<query 1> -- column names define output columns
[Set Operation] -- specifying ALL will return duplicates
<query 2>

UNION [ALL] - return rows from both queries - using all will return rows from both queries
            even if they are duplicates

INTERSECT - return rows if they appear in both queries

EXCEPT - return rows if they appear in the 1st query but not the 2nd query
```

PIVOT

```
Pivot = rows-> columns

Pivot Process

1. Group
2. Spread
3. Aggregate
   SELECT ...
   FROM <source_table_or_table_expression>
     PIVOT(<agg_func>(<aggregation_element>)
         FOR <spreading_element>
          IN (<list_of_target_columns>)) AS <result_table_alias>
   WITH ProductSalesYearCTE
   AS
   (
      SELECT PartNumberID, [1],[2],[3],[4],[5],[6],[7],[8],[9],[10],[11],[12]
      FROM
      (
         SELECT PartnUmberID, LineTotal, MONTH(DateSold) AS SalesMonth
         FROM sales.SkuSales
         WHERE YEAR(DateSold) = 2016
```

```
    ) AS SalesInYear
    PIVOT(SUM(LineTotal) FOR SalesMonth IN ([1],[2],[3],[4],[5],[6],[7],[8],[9],[10],[11],[12]))  AS P
  )
  SELECT PartNumberID, COALESCE([1], 0) AS January, COALESCE([2], 0) AS February, COALESCE([3], 0) AS Mar
ch, COALESCE([4], 0) AS April
            , COALESCE([5], 0)AS May, COALESCE([6], 0) AS June, COALESCE([7], 0) AS July, COALESCE([8], 0) AS A
ugust
            , COALESCE([9], 0) AS September, COALESCE([10], 0) AS October, COALESCE([11], 0) AS November, C
OALESCE([12], 0) AS December
   FROM ProductSalesYearCTE
  ORDER BY PartNumberID
```

UNPIVOT

```
unpivot - columns->rows ```sql
SELECT ...
FROM <source_table_or_table_expression>
 UNPIVOT(<target_col_to_hold_source_col_values>
  FOR <target_col_to_hold_source_col_names> IN(<list_of_source_columns>)) AS
<result_table_alias> ```
              select PartNumberID, SalesMonth, LineTotal
              from sales.ViewProductSalesYear
                          unpivot (LineTotal for SalesMonth in (January, February, March, April, May, June, July, A
ugust,September,October,November,December) ) as U
```

Select from VALUES

```
  SELECT * FROM
  (
    VALUES (1,2), (2,3), (3,4), (4,5)
  ) AS t (field_1, field2)

  SELECT *
  FROM ( VALUES
        (10003, '20090213', 4, 'B'),
        (10004, '20090214', 1, 'A'),
        (10005, '20090213', 1, 'C'),
        (10006, '20090215', 3, 'C') )
  AS O(orderid, orderdate, empid, custid);
```

Grouping Sets

```
  SELECT PartNumberID, Store, SUM(LineTotal) AS TotalSold
  FROM sales.SkuSales
  WHERE YEAR(DateSold) = 2013
  group by
    GROUPING sets
    (
      (), -- total 2013 Sales
```

```
        (Store), -- total 2013 sales by store
        (store,PartNumberID) -- total 2013 sales by Store + Part Number
    );

-- ROLLUP clause
-- will return
    -- 1) all Sales
    -- 2) Each Store
    -- 3) Each Store + SKU
    -- 4) Does not return just the SKU because PartNumberID listed after Store in rollup clause
SELECT PartNumberID, Store, SUM(LineTotal) AS TotalSold
FROM sales.SkuSales
WHERE YEAR(DateSold) = 2013
GROUP BY ROLLUP( Store, PartNumberID); -- Left-right order important!
```

Group by cube

```
- total sales at each store, and for each sku
-    PartNumberID    Store
    null    null    - total 2013 sales
    <value>    null    - total 2013 sales for sku (any store)
    null    <value>    - total 2013 sales at store
    <value> <value>    - total 2013 sales for sku at store

SELECT PartNumberID, Store, SUM(LineTotal) AS TotalSold
FROM sales.SkuSales
WHERE YEAR(DateSold) = 2013
GROUP BY CUBE(Store, PartNumberID)
ORDER BY PartNumberID
```

Insert from Select

```
    INSERT INTO tempdb.dbo.ModelListing ( ManufacturerCode, Name)
        SELECT  ManufacturerCode, Name
        FROM product.Model
```

Insert from Sproc

```
    INSERT INTO tempdb.dbo.ModelListing
    EXEC product.ModelList
```

Select into a new table

```
        -- note that ModelID is an identity column, and will be created as such
        SELECT ModelID, ManufacturerCode, Name
        INTO tempdb.dbo.Model
        FROM product.Model
```

Bulk Insert

```
test file:
   1,Brian,2003
   2,Kit,2006
   3,Dean, 2007
   4,Ryan,2010
   USE tempdb;

   if OBJECT_ID('dbo.SSBulk_Insert_Test','U') is not null drop table dbo.SSBulk_Insert_Test;

   create table dbo.SSBulk_Insert_Test
   (
      id int,
      person varchar(25),
      year_started int
   )

   BULK INSERT dbo.SSBulk_Insert_Test FROM 'c:\temp\SSBulk_Insert_Test.txt' -- MUST be releative to the s
erver
    WITH
     (
       DATAFILETYPE    = 'char',
       FIELDTERMINATOR = ',',
       ROWTERMINATOR   = '\n'
     );
```

Last Identity

```
SCOPE_IDENTITY() -- @@identity is legacy, do not use

-- scope_identity() will be null if no inserts in the current session
-- do not use the following as replacement for scope_identity()
select IDENT_CURRENT('product.Model')
```

DELETE

```
SELECT TOP 100 *
INTO tempdb.dbo.Model
FROM product.Model

-- Does not update identity seed -
DELETE FROM tempdb.dbo.Model -- entire table
DELETE FROM tempdb.dbo.MOdel WHERE ModelID = 10000

-- resets identity seed
TRUNCATE TABLE tempdb.dbo.Model

-- delete based on join
DELETE FROM modelTemp
FROM tempdb.dbo.Model modelTemp
```

```
JOIN product.Manufacturer mfg
   ON modelTemp.ManufacturerID = mfg.ManufacturerID
WHERE mfg.MaufacturerID = 10000
```

UPDATE based on join

```
DECLARE @table1 TABLE (date_field DATE, value_field VARCHAR(25))

INSERT INTO @table1 (date_field, value_field) VALUES
   ('4/1/2012', 'intial value 1'), ('4/2/2012', 'intial value 2'),
   ('4/3/2012', 'intial value 3'), ('4/4/2012', 'intial value 4')

SELECT * FROM @table1

DECLARE @table2 TABLE (date_field DATE, value_field VARCHAR(25))

INSERT INTO @table2 (date_field, value_field) VALUES
   ('4/1/2012', 'modified value 1'),('4/2/2012', 'modified value 2'),
   ('4/3/2012', 'modified value 3'),('4/4/2012', 'modified value 4')

UPDATE t1
  SET t1.value_field = t2.value_field
FROM @table1 t1
  JOIN @table2 AS t2
    ON t1.date_field = t2.date_field
```

MERGE

```
MERGE INTO <destination table> as dest
using <source table> as source
   on dest.key = source.key
WHEN MATCHED THEN
   UPDATE SET
      dest.val1 = source.val1,
      dest.val2 = source.val2
WHEN NOT MATCHED THEN
   INSERT (key, val1, val2)
   VALUES (source.key, source.val1, source.val2)
```

OUTPUT clause

```
DECLARE @temp_table TABLE (ManufacturerCode VARCHAR(50))

INSERT INTO @temp_table (ManufacturerCode)
SELECT ManufacturerCode
FROM product.ManufacturerCode
WHERE ManufacturerCode LIKE 'PG-%';


-- same as above, but with optional OUTPUT clause
```

```
INSERT INTO @temp_table (ManufacturerCode)
OUTPUT inserted.ManufacturerCode -- each displayed field must be preceded by inserted.<field name>
SELECT ManufacturerCode
FROM product.ManufacturerCode
WHERE ManufacturerCode LIKE 'PG-%';

-- delete

DELETE FROM @temp_table
OUTPUT deleted.ManufacturerCode
WHERE ManufacturerCode LIKE 'PG-%'

-- use output clause to view changed values (there is no 'updated' value per se)
update @temp_table
   set ManufacturerCode = SUBSTRING(ManufacturerCode, 1,4)
output
   deleted.ManufacturerCode as prior_value,
   inserted.ManufacturerCode as new_value
```

Transactions

```
BEGIN TRAN -- if not specified, each statement runs as an implicit transaction

-- Statement #1
-- Statement #2
...
-- Statement N

COMMIT TRAN -- All statements since BEGIN TRAN committed to database

ROLLBACK -- ALL statements since BEGIN TRAN are canceled, no data will be changed
```

Detailed Transaction Template

```
-- http://stackoverflow.com/questions/2073737/nested-stored-procedures-containing-try-catch-rollback-patter
n/2074139#2074139
SET XACT_ABORT, NOCOUNT ON

DECLARE @starttrancount INT

BEGIN TRY
      SELECT @starttrancount = @@TRANCOUNT

      IF @starttrancount = 0
            BEGIN TRANSACTION

            -- Do work, call nested sprocs


      IF @starttrancount = 0
            COMMIT TRANSACTION
END TRY
```

```
BEGIN CATCH
	IF XACT_STATE() <> 0 AND @starttrancount = 0
		ROLLBACK TRANSACTION

	DECLARE @error INT, @message VARCHAR(4000)

	SET @error = ERROR_NUMBER()
	SET @message = ERROR_MESSAGE()

	RAISERROR('<Sproc Name> : %d: %s', 16, 1, @error, @message)
END CATCH
```

Locking

```
- Exclusive Locks
   - generated by update operations
   - no other session can update or read an item that has an exclusive lock
- Shared Locks
   - generated by selects

- Isolation levels affect how sql interacts with shared locks:
   READ UNCOMMITTED - SELECT does not generate shared locks - dirty reads
   READ COMMITTED (default) - SELECT requires shared locks
   REPEATABLE READ - shared lock open for the entire transaction
   SERIALIZABLE - locks range of keys returned to prevent phantom records
   SNAPSHOT - reads previous row reivsion stored in tempdb
   READ COMMITTED SNAPSHOT - Gets the last committed version of the row that was available when the stat
ement started
- can be set at database level or transaction level:
   SET TRANSACTION ISOLATION LEVEL <name>
```

Variables

```
DECLARE @i AS INT;
SET @i = 10;


DECLARE @i AS INT = 10; -- only works in SS2008 or higher

DECLARE @ManufacturerCode VARCHAR(25)
DECLARE @Name VARCHAR(100)

-- select into variables
SELECT @ManufacturerCode = ManufacturerCode,
	@Name = Name
FROM product.Model
WHERE ModelID = 10000
```

Flow Control

```
-- single statements
IF <expression>
   -- will execute if expression is true
ELSE
   -- will execute if expression is false or unknown

-- Multiple statements
IF <expression>
   BEGIN

   END
ELSE
   BEGIN

   END

WHILE <expression
   begin

   end

-- FOR replacement
DECLARE @i AS INT;

SET @i = 1
WHILE @i <= 10
BEGIN
  PRINT @i;
  SET @i = @i + 1;
END;
```

Cursors

```
   DECLARE @ModelID  INT,
         @ManufacturerCode VARCHAR(25)

   DECLARE C CURSOR FAST_FORWARD /* read only, forward only */ FOR
   SELECT ModelID, ManufacturerCode
   FROM Product.Model
   ORDER BY ModelID

   OPEN C

   FETCH NEXT FROM C INTO @ModelID, @ManufacturerCode;

   WHILE @@FETCH_STATUS = 0
   BEGIN
   -- do work
   FETCH NEXT FROM C INTO @ModelID, @ManufacturerCode;
   END
```

```
    CLOSE C;

    DEALLOCATE C;
```

Min Key approach

```
    DECLARE @ModelID  INT
    DECLARE @ManufacturerCode VARCHAR(25)

    SET @ModelID = (SELECT MIN(ModelID) FROM product.model)

    WHILE @ModelID IS NOT NULL
    BEGIN
        SELECT @ManufacturerCode = ManufacturerCode
        FROM product.model
        WHERE ModelID = @ModelID

        -- do work

        SET @ModelID = (SELECT MIN(ModelID) FROM product.model WHERE ModelID > @ModelID)
    END
```

Temporary Tables

```
CREATE TABLE #TempTable
(
    ModelID INT,
    ManufacturerCode VARCHAR(25)
)

CREATE TABLE ##GlobalTempTable
(
    ModelID INT,
    ManufacturerCode VARCHAR(25)
)
```

Table Variables

```
DECLARE @TempTable TABLE
(
    ModelID INT,
    ManufacturerCode VARCHAR(25)
);
```

Table Types

```
IF TYPE_ID('dbo.OrderTotalsByYear') IS NOT NULL
  DROP TYPE dbo.OrderTotalsByYear;
```

```
CREATE TYPE dbo.OrderTotalsByYear AS TABLE
(
  orderyear INT NOT NULL PRIMARY KEY,
  qty       INT NOT NULL
);

DECLARE @MyOrderTotalsByYear AS dbo.OrderTotalsByYear;
```

Dynamic SQL

```
-- EXEC
DECLARE @sql AS VARCHAR(100);
SET @sql = 'PRINT ''This message was printed by a dynamic SQL batch.''';';
EXEC(@sql);

-- sp_execute - supports params
DECLARE @sql NVARCHAR(MAX);

SET @sql = N'select * from product.model
where Name LIKE ''%'' +  @Name + ''%'' AND CategoryId = @CategoryId'

EXEC sp_executesql
   @stmt = @sql,
   @params = N'@Name as varchar(100), @CategoryId as int',
   @Name = 'John Deere',
   @CategoryId = 2
```

User Defined Functions

Scalar

```
CREATE FUNCTION dbo.fn_age
(
  @birthdate AS DATETIME,
  @eventdate AS DATETIME
)
RETURNS INT -- <-- Defines as scalar
AS
BEGIN
  RETURN
    DATEDIFF(year, @birthdate, @eventdate)
     - CASE WHEN 100 * MONTH(@eventdate) + DAY(@eventdate)
            < 100 * MONTH(@birthdate) + DAY(@birthdate)
       THEN 1 ELSE 0
      END
END
GO
```

Table Value

```
ALTER FUNCTION [dbo].[Split_MultiValue_Parameter]
```

```
    (    @delimitedString    VARCHAR(MAX)
       ,@delimiter          VARCHAR(1)
    )
    RETURNS @Table TABLE (VALUE VARCHAR(100)) -- <-- Defines as table

    AS
    BEGIN
      DECLARE @tempString VARCHAR(MAX);
      SET @tempString = ISNULL(@delimitedString,'') + @Delimiter;
      WHILE LEN(@tempString) > 0
      BEGIN
        INSERT INTO @Table
        SELECT SUBSTRING(@tempString,1,
             CHARINDEX(@Delimiter,@tempString)-1);

        SET @tempString = RIGHT(@tempString,
          LEN(@tempString)-CHARINDEX(@Delimiter,@tempString)) ;
      END
      RETURN;
    END
```

Stored Procedures

```
  CREATE PROCEDURE product.ModelList
  AS
  BEGIN
  SET NOCOUNT ON;

  SELECT ModelId, Name, ManufacturerCode, CategoryId, Description
  FROM product.Model

  END
```

Triggers

DML - Data Modification

```
    ALTER TRIGGER  product.trProductPartNumberDateModified ON product.PartNumber FOR UPDATE AS
    DECLARE @PartNumberID INT;
    SELECT @PartNumberID = PartNumberId FROM Inserted;

    UPDATE Product.PartNumber
    SET DateModified = GETDATE()
    WHERE PartNumberID = @PartNumberID;
```

Structural modification

```
      CREATE TRIGGER [DDL_Notify]
      ON DATABASE
      FOR DROP_TABLE, ALTER_TABLE,CREATE_TABLE, DROP_FUNCTION, ALTER_FUNCTION, CREATE_FUNCTI
ON,
```

```
        DROP_PROCEDURE, ALTER_PROCEDURE, CREATE_PROCEDURE

    AS

    -- actions
```

Try / Catch

```
BEGIN TRY

   TRUNCATE TABLE tempdbo.dbo.DoesNotExist
   -- following statement will not execute
   select top 1 * from product.model

END TRY

BEGIN CATCH
   -- 4701 Cannot find the object "DoesNotExist" because it does not exist or you do not have permissions.
   SELECT ERROR_NUMBER(), ERROR_MESSAGE()

END CATCH
```

Template for basic table + relationships

```
CREATE TABLE tempdb.dbo.temp1
(
   Temp1_pk INT IDENTITY(1,1) NOT NULL
   CONSTRAINT PK_dbo_Temp1 PRIMARY KEY CLUSTERED,
   VALUE VARCHAR(25)
)
GO

CREATE TABLE tempdb.dbo.temp2
(
   Temp2_pk INT IDENTITY(1,1) NOT NULL
      CONSTRAINT pk_dbo_temp2 PRIMARY KEY CLUSTERED,
   Temp1_fk INT NOT NULL
      CONSTRAINT FK_dbo_Temp2_dbo_Temp1 FOREIGN KEY (Temp1_fk)
         REFERENCES dbo.temp1(Temp1_pk),
   VALUE VARCHAR(25) NULL
)
GO
```

Identity Insert

```
SET IDENTITY_INSERT product.Model ON

   INSERT INTO product.Model (ModelID, Name)
   VALUES (12345, 'Test')
```

```
SET IDENTITY_INSERT product.Model OFF
```