**SQL SERVER 2005 Courseware**

**Table of Contents**

| **Sr. No.** | **Topic Name** | **Page No.** |
| --- | --- | --- |
| 1 | **Querying Data** |  |
|  | 1.1 Basic Select Statement | 3 |
|  | 1.2 Joins | 8 |
|  | 1.3 Set Operators | 13 |
|  | 1.4 Summary Queries | 16 |
|  | 1.5 Date Functions | 23 |
|  | 1.6 Sub Query | 24 |
| 2 | **DDL Commands** |  |
|  | 2.1 Data types | 33 |
|  | 2.2 Create Table Statement | 34 |
|  | 2.3 Referential Integrity | 40 |
|  | 2.4 Identity | 42 |
|  | 2.5 Rule | 44 |
|  | 2.6 Alter statement | 46 |
|  | 2.7 Temporary Table | 48 |
|  | 2.8 Views | 49 |
|  | 2.9 Indexes | 51 |
| 3 | **Programming** |  |
|  | 3.1 Programming Basics | 53 |
|  | 3.2 Cursors | 59 |
|  | 3.3 Stored Procedures | 64 |
|  | 3.4 Functions | 68 |
|  | 3.5 Triggers | 71 |
| 4 | **New Features of SQL Server 2005** |  |
|  | 4.1 DDL Triggers | 80 |
|  | 4.2 Top Clause enhancements | 81 |
|  | 4.3 DML output Clause | 82 |
|  | 4.5 Pivot | 85 |
|  | 4.6 Common Table Expression | 90 |
|  | 4.7 Set operator enhancements | 99 |
|  | 4.7 Referential Integrity enhancements | 100 |
|  | 4.8 Try Catch | 102 |
| 5 | **Database Architecture** | 107 |

1. **Querying Data**

**1.1.** Basic Select Statement

Relational Operators

1. >
2. >=
3. <
4. <=
5. =
6. != or <> (For Not Equal To)
7. Between *lower value* And *higher value*
8. Not Between *lower value* And *higher value*
9. IN (*List of Values*)
10. NOT IN (*List of Values*)
11. Like
12. NOT Like

Examples

1. Between

select \* from emp

where sal between 1250 And 3000;

(1250 and 3000 will be inclusive)

select \* from emp

where sal Not between 1250 And 3000;

(1250 and 3000 will be exclusive)

1. IN

select \* from emp

where job IN ('CLERK', 'MANAGER');

IN is equivalent to logical OR

Logical Operators

1. AND
2. OR
3. NOT
4. AND operator will check conditions in combination

select \* from emp

where job = 'CLERK' AND sal < 1000

1. OR operator will check conditions in isolation

select \* from emp

where job = 'MANAGER' or sal < 1000

If AND OR are together then AND is executed first then OR

select \* from emp

where job = 'MANAGER' or job = 'CLERK' and sal < 1000

Parentheses will override the default precedence

select \* from emp

where (job = 'MANAGER' or job = 'CLERK' ) and sal < 1000

**% , \_ , [*character range*] and [*number range*] are pattern matching operators used with Like.**

**Example of %**

To see records of employees which have S as the starting character.

select ename,sal,deptno

from emp

**where ename like 'S%';**

select ename,sal,deptno

from emp

where ename **not like** 'S%';

**Example of underscore (‘\_’)**

create table pcodes

(code varchar(20), qty integer);

Insert into pcodes values('A01',100);

Insert into pcodes values('AB01',400);

Insert into pcodes values('ABC01',200);

Insert into pcodes values('ABCD01',90);

Insert into pcodes values('ZX01',90);

Insert into pcodes values('Z01',98);

Insert into pcodes values(‘PABZ-90’,102);

Insert into pcodes values(‘PACZ-90’,102);

Insert into pcodes values(‘PADZ-90’,102);

Insert into pcodes values('PW1-RT',89);

Insert into pcodes values('PW2-RT',56);

Insert into pcodes values('PW3-RT',98);

Insert into pcodes values('PW4-RT',187);

Insert into pcodes values('PW5-RT',5);

**To see records where there are 2 characters before 01**

SELECT \* FROM PCODES

WHERE CODE LIKE '\_\_01';

**Example of [character range] (For SQL server only)**

[] is used for character within range.

To see records where the 3rd character is either B or D in the code.

select \* from pcodes

where code like 'PA**[BD]**Z-90'

To see records where the 3rd character is neither B nor D

^ is used to check any character is not within the range.

select \* from pcodes

where code like **'PA[^BD]Z-90'**

**Example of [Number range]**

To see records where the 3rd character is from 2 to 4 (range)

SELECT \* FROM PCODES

WHERE CODE LIKE 'PW**[2-4]%**'

[^range]

To see records where the 3rd character is not from 2 to 4 (range)

SELECT \* FROM PCODES

WHERE CODE LIKE 'PW[^2-4]%'

**Concatenation :**

+ operator

select lastname **+** firstname "complete name" from employees

(+ needs all data types matching)

For different data types such as number and varchar use cast function

select lastname + ':' + **cast(employeeid as varchar)** "complete name" from employees

--select ename "Name of the Employee", job from emp;

--select ename + job from emp;

--select ename + cast(sal as varchar) from emp;

--select ename as "Name of employee" from emp;

--select ename + ' is working in ' + cast(deptno as varchar) "Details" from emp

--select ename,job,sal, sal \* 12 as "Annual salary" from emp

**Dealing with Nulls**

**Is Null operator** – To check the value is null.

select \* from emp

where comm **is null**

select \* from emp

where comm **is not null;**

**Calculations with null values –**

Select ename,sal,comm, **sal+comm** From emp

**Coalesce** function will consider null as a number specified.

select ename,sal,comm, **sal + coalesce(comm,0) “Take Home”** from emp;

**1.2 JOINS**

**Joins is taking data from more than one tables.**

**Three types of Joins:**

1. **INNER**
2. **OUTER**
3. **CROSS**

**Inner-Join**

**Retrieving common records from the 2 tables on equality condition –**

**To see names and department names from emp and dept table**

**Non-ANSI Method:**

Select ename, dname

From emp, dept

Where emp.deptno = dept.deptno;

**ANSI Method:**

select ename, dname

from emp inner join dept

on emp.deptno = dept.deptno;

In the Select Statement if the common column has to be displayed then it has to be prefixed by table name in both ANSI and NON-ANSI methods.

select ename, dname, **emp.deptno**

from emp join dept

on emp.deptno = dept.deptno;

**Retrieving common records from the 3 tables on equality condition –**

**Between e and d deptno is common and between d and p, pcode is common**

**To see names, department names and product names from the 3 tables on matching values –**

**Non-ANSI Method:**

select ename, dname, pname

from e, d, p

where e.deptno = d.deptno AND d.pcode = p.pcode;

**ANSI Method:**

select ename, dname, pname

from e join d

on e.deptno = d.deptno

join p

on d.pcode = p.pcode;

**Cross Join**

No join condition. So each row of first table gets combined with each record of the other.

Cartesian product takes place.

select ename, dname

from emp, dept;

Cross join is effective for financial applications such as calculations of interest rates for each month.

In the tables period and rates nothing is common. Still cross product can be achieved.

Now each roi is to be multiplied by each month for statistical report –

**Non-ANSI Method:**

select roi, month, roi\*month as "Interest"

from rates, period;

**ANSI Method:**

select roi, month, roi\*month as "Interest"

from rates **CROSS** join period;

**Left Outer Join**

**To take matching records from both the tables and all remaining records from the left table.**

In the tables emp1 and dept1, both the tables have one record non-matching.

To see names of the employee and their departments and also the names of the employees who do not have any department

**ANSI Syntax**

select ename,dname

from emp1 **LEFT** outer join dept1

on emp1.deptno = dept1.deptno

**Right Outer Join**

**To take matching records from both the tables and all remaining records from the right table.**

To see the names of the employees and their department names and also the --department names which do not have any employee;

**ANSI Syntax**

select ename, dname

from emp1 **RIGHT** outer join dept1

on emp1.deptno = dept1.deptno

**Full Outer Join**

**First Matching records from both tables, then remaining records from left table and then the remaining records from the right table are displayed.**

To see employee names and their departments, employees who do not have department as well as department names which are not having any employee

**ANSI Syntax**

select ename, dname

from emp1 **FULL** outer join dept1

on emp1.deptno = dept1.deptno;

**Self Join**

**In self join the table is joined to itself. To images of the same table will get created with different alias name for each table.**

create table EM

(empno varchar(4),

ename varchar(30),

mgr varchar(4));

insert into EM values ('E1', 'John','E4');

insert into EM values ('E2', 'Smith', 'E1');

insert into EM values ('E3','Roger', 'E2');

insert into EM values ('E4','Martin', 'E3');

To see names of employees and their managers

select E.ename "Employee", M.ename "Manager"

from em E,em M

where E.mgr = M.empno;

Example 2

To see if a record is duplicated

The same student’s record is duplicated with different roll number.

create table sj

(roll integer not null primary key,

name varchar(50),

sem varchar(10),

marks integer);

Insert into sj values(1,'Ramesh', 'First', 90);

Insert into sj values(2,'Rajesh', 'First', 95);

Insert into sj values(3,'Mahesh', 'First', 81);

**Insert into sj values(4,'Ramesh', 'First', 90);**

select t1.roll, t1.name, t1.sem, t1.marks

from sj t1, sj t2

where t1.roll <> t2.roll

and t1.name = t2.name

and t1.sem = t2.sem

and t1.marks = t2.marks;

**SQL Server 2000 Syntaxes of Joins**

Join Syntaxes till SQL Server 2000 had some proprietary clauses like **\*=** and =\*.

If we are in 2005 then the compatibility has to be given to 2000 by the inbuilt stored procedure sp\_dbcmptlevel. It has two parameters : Database name and version.

70 version is for SQL Server 7

80 version is for SQL Server 2000

90 version is for SQL Server 2005

sp\_dbcmptlevel master, 80

select ename,dname

from emp1, dept1

where emp1.deptno \*= dept1.deptno -- Left Outer Join

select ename,dname

from emp1, dept1

where emp1.deptno =\* dept1.deptno -- Right Outer Join

(Full outer join is not provided)

**1.3** **SET OPERATORS**

# **UNION**

**UNION ALL**

create table pune

(custid integer,

prodname varchar(20));

insert into pune values(107,'A');

insert into pune values(107,'B');

insert into pune values(107,'C');

insert into pune values(107,'A');

insert into pune values(107,'D');

insert into pune values(107,'E');

insert into pune values(108,'A');

insert into pune values(108,'B');

insert into pune values(108,'B');

insert into pune values(108,'C');

insert into pune values(108,'Y');

insert into pune values(109,'Z');

insert into pune values(109,'A');

insert into pune values(109,'A');

insert into pune values(109,'B');

**--UNION**

--Combines the result of two or more queries eliminating duplicates

--Rule -- The couln list and their data types of all the queries should be same

--To see distinct products sold to 107 and 108

select prodname

from pune

where custid =107

UNION

select prodname

from pune

where custid =108;

**UNION ALL**

Shows all the values form both the queries including duplicates

To see all product names for 107 and 108

select prodname

from pune

where custid =107

UNION ALL

select prodname

from pune

where custid =108;

SET OPERATORS WITH TWO TABLES (Effective when no columns are matching)

create table mech

(rollno integer,

marks integer);

create table comp

(rollno integer,

marks integer);

insert into mech values (101,90);

insert into mech values (102,56);

insert into mech values (103,78);

insert into mech values (104,35);

insert into mech values (105,100);

insert into mech values (106,56);

insert into comp values(201,78);

insert into comp values(202,88);

insert into comp values(203,43);

insert into comp values(204,56);

insert into comp values(205,59);

**UNION BETWEEN MECH AND COMP**

-- To show rollno and marks from mech and comp tables where

-- marks are >= 70

-- Here we cannot go for any type of JOIN since the rollno are not at all

-- matching and the two tables are independent

select rollno,marks

from mech

where marks >= 70

UNION

select rollno,marks

from comp

where marks >= 70

order by marks desc;

Here rollno and marks combination is checked in both queries. So individual --marks duplication is allowed. But no duplication of combination is done.

If only marks was the column the 78 will not be repeated.

**UNION ALL**

-- To see the master list of mech and com tables

select rollno,marks

from mech

UNION ALL

select rollno,marks

from comp;

**1.4** **Summary Queries**

**Aggregate Functions**

1. SUM()
2. MAX()
3. MIN()
4. AVG()
5. COUNT()

Group By clause will eliminate duplicates for a value and sort the values in ascending manner.

Select deptno

From emp

Group by deptno;

Group By clause can also do aggregation within that value.

**Group by clause**

**-- To see maximum salary in each department**

select deptno,max(sal) from emp

group by deptno;

**Having clause**

Having clause is used to filter summary records

**-- To see deptno who have maximum salary more than 3000**

select deptno,max(sal)

from emp

group by deptno

Having max(sal) > 3000

**Nested Grouping**

**-- To see Job wise maximum salary within each department**

select deptno, job,max(sal) from emp

group by job,deptno;

**-- To see Department-wise maximum salary within each job**

select Job,deptno,max(sal) from emp

group by deptno,job;

#### **COMPUTE**

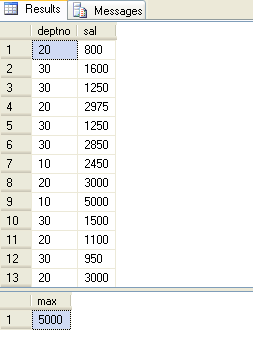
Generates totals that appear as additional summary columns at the end of the result set.

After showing all the records, it shows the summary record

select deptno,sal

from emp

**Compute Max(Sal)**

****

*After showing the 14 records the highest salary is displayed.*

select deptno,sal

from emp

Compute Max(Sal),min(sal),avg(sal),sum(sal)

Rule for Compute –

It can only show summary of the fields mentioned in the select column list

Cannot give alias after the compute function. The column heading will be the function name.

**To show deptno wise highest salary, as well as the highest salary amongst those.**

select deptno,max(sal)

from emp

group by deptno

**compute max(max(sal))**

#### **COMPUTE BY**

Compute By gives break up of summary values**(sub-totals)** and the normal values.

To show deptno ,salaries and the sub-total of salary at each deptno change.

select deptno,sal

from emp

**order by deptno**

#### **Compute sum(sal) by deptno**

* **For successful execution of the Compute By clause the grouping field must be sorted.**

select job,deptno,sal

from emp

order by job,deptno

Compute sum(sal) by job,deptno

In this case here sorting is done for deptno within each job. So once the combination is getting changed the total for that combination is shown.

* If only one column is there in order by then that same column has to be there in compute by clause

select job,deptno,sal

from emp

**order by job**

**Compute sum(sal) by deptno --- Wrong**

* **If more than one columns are in the order by clause then at least the first column has to be there in the compute by clause**

select job,deptno,sal

from emp

**order by job,deptno**

**Compute sum(sal) by deptno ----- Wrong**

select job,deptno,sal

from emp

**order by job, deptno**

###### **Compute sum(sal) by job**

**In this case the deptno are sorted within each job and when job changes the total of salary is shown**

* **When the number of columns and names in the Order By and Compute by are matching then still the Order By clause first column and the Compute By clause first column needs to be same.**

select job,deptno,sal

from emp

**order by job, deptno**

**Compute sum(sal) by deptno, job -- Wrong**

select job,deptno,sal

from emp

**order by job, deptno**

**Compute sum(sal) by job, deptno -- Correct**

##### ***ROLLUP***

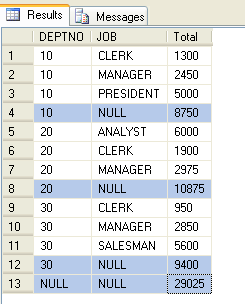
To generate sub-totals and grand totals

#### **To show sub-total for each deptno and the grand total**

SELECT DEPTNO,JOB,SUM(SAL)

FROM EMP

GROUP BY DEPTNO, JOB **WITH ROLLUP**

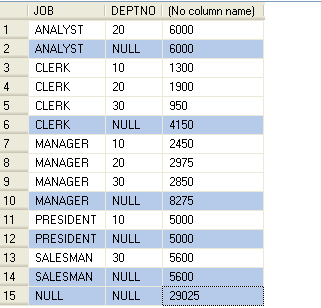
****

#### **To show sub-total for each job and the grand total**

SELECT JOB, DEPTNO, SUM(SAL)

FROM EMP

GROUP BY JOB,DEPTNO **WITH ROLLUP**



#### **CUBE**

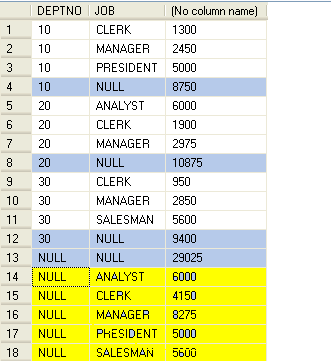
Cube produces an super-aggregate row. It creates all possible combinations of groupings from the list of columns in the Group By clause.

#### **To see the sub-totals as per deptno as well as jobs and the grand total**

SELECT DEPTNO,JOB,SUM(SAL)

FROM EMP

GROUP BY DEPTNO, JOB **WITH CUBE**



**1.5 Date Functions**

select getdate()

select datename(dd,getdate()) returns the dd part

select datename(dw,getdate()) returns the day part (Sunday)

select \* from employee

where datename(mm,hire\_date) = 'december'

##### ***Syntax***

DATENAME **(** *datepart* **,** *date* **)**

##### ***Arguments***

*datepart*

Is the parameter that specifies the part of the date to return.

| **Datepart** | **Abbreviations** |
| --- | --- |
| **Year** | **yy, yyyy** |
| **Quarter** | **qq, q** |
| **Month** | **mm, m** |
| **Dayofyear** | **dy, y** |
| **Day** | **dd, d** |
| **Week** | **wk, ww** |
| **Weekday** | **dw** |
| **Hour** | **hh** |
| **Minute** | **mi, n** |
| **Second** | **ss, s** |
| **Millisecond** | **ms** |

**1.6 SUB QUERY**

**Single Row Sub Query**

**To see records of person who earn salary higher than Martin’s salary.**

select \* from emp

where sal > **(select sal**

**from emp**

**where ename = 'MARTIN')**

**To see records of person who earn the highest salary**

select ename,sal from emp

where sal = **(select max(sal)**

**from emp)**

**Multi Row Sub Query**

#### **SubQuery with IN operator**

**To see records of persons from emp table who earn salary equal to salaries of deparment\_id 30 of the employee table.**

**select sal, ename from emp**

**where sal IN** **(select salary**

**from employee**

**where department\_id = 30**

**)**

**order by sal desc**

#### **ANY in subqueries**

**To see the salary, names, job of the employees who earn more than any employee in the job Salesman.**

select sal,job,ename

from emp

where sal >**ANY** **(select sal**

**from emp**

**where job = ‘SALESMAN’)**

#### **ALL in Sub queries**

**To see the salary, names, job of the employees who earn more than all employees in the job SALESMAN.**

select sal,job,ename

from emp

where sal >**ALL** **(select sal**

**from emp**

**where job = ‘SALESMAN’)**

**order by sal desc;**

**Multiple Column Sub-Queries –**

**To display the records of employees who have the deptno and job same as that os SMITH.**

## ***select \* from emp***

## ***where job =*** *(select job*

## *from emp*

## *where ename = 'SMITH')*

## ***AND***

## ***deptno =*** *(select deptno*

## *from emp*

## *where ename = 'SMITH')*

**Nested Sub queries**

**To display records having salary above the highest salaries for the job of Miller.**

**SELECT \* FROM EMP**

**WHERE SAL >** **(SELECT MAX(SAL)**

### **FROM EMP**

**WHERE JOB = (SELECT JOB**

**FROM EMP**

**WHERE ENAME = 'MILLER');**

**Display ename, dname and salary**

**for all those who are earning salary above Martin's salary**

**select ename,dname,sal**

**from emp inner join dept**

**on emp.deptno = dept.deptno**

**where emp.sal > (select sal**

**from emp**

**where ename = 'MARTIN');**

**CORRELATED Sub Queries**

**Correlated sub queries in where clause**

To see employees who earn salary less than the average salary of their own job

Parent query takes each row and submits it to child query.

Child query gets executed for each row

select \* from emp **E**

where sal < (select avg(sal)

from emp

where job = **E.job**)

**To see top two salary values**

select distinct sal

from emp **E**

where 2 > (select count(distinct sal)

from emp

where sal > **E.sal**)

Order by Sal Desc

To see the third top salary

select distinct sal

from emp **e**

where 3 = (select count(distinct sal)

from emp

where sal >= **e.sal**);

OR

select distinct sal

from emp e

where 2 = (select count(distinct sal)

from emp

where sal > e.sal)

**Co related sub queries in having clause**

Display avg sal and jobs having avg sal above the (lowest sal + 200) of their own jobs

select avg(sal), job

from emp e

group by job

having avg(sal) > (select (min(sal) + 200)

from emp

where job = e.job);

Cross-check---

select job,avg(sal) "avg", min(sal) "min" , (min(sal) + 200) "Diff"

from emp

where job in ('clerk','manager')

group by job;

**Co related sub queries in update statement**

alter table emp

add dname varchar(10);

update emp

set dname = (select dname

from dept D

where Emp.deptno = D.deptno);

**Co related sub queries in delete statement**

Creating the table retired\_emp without taking data of the emp table.

select \* into retired\_emp from emp

where 1= 2;

Insert into Retired\_Emp

select \* from Emp

where ename = 'JONES';

Insert into Retired\_Emp

select \* from Emp

where ename = 'MARTIN';

Now to delete records from Emp table for all those have been retired.

delete from emp

where empno = (select empno

from Retired\_Emp R

where Emp.empno = R.empno);

**Exists Operator:**

The EXISTS operator tests for existence of rows in the results set of the sub query.

• If a subquery row value is found:

– The search does not continue in the inner query

– The condition is flagged TRUE

• If a subquery row value is not found:

– The condition is flagged FALSE

-The search continues in the inner query till either the condition becomes TRUE or if not then till the last record.

Drop table e1;

Drop table d1;

create table e1

(empno integer,

ename varchar(20),

deptno integer);

create table d1

(deptno integer,

location varchar(20));

Insert Into E1 Values(1,'a',10);

Insert Into E1 Values(2,'b', 40);

Insert Into E1 Values(3,'c', 10);

Insert Into E1 Values(4,'d',30);

Insert Into E1 Values(5,'e',10);

Insert Into E1 Values(6,'f',20);

Insert Into E1 Values(7,'g',30);

Insert Into E1 Values(8,'h',30);

Insert Into E1 Values(9,'i',20);

Insert Into E1 Values(10,'j',50);

Insert Into E1 Values(11,'k',10);

Insert Into E1 Values(12,'l',40);

Insert Into D1 Values(10,'Pune');

Insert Into D1 Values(10,'Mumbai');

Insert Into D1 Values(10,'Bangalore');

Insert Into D1 Values(20,'Mumbai');

Insert Into D1 Values(20,'Chennai');

Insert Into D1 Values(20,'Bangalore');

Insert Into D1 Values(30,'Chennai');

Insert Into D1 Values(30,'Delhi');

Insert Into D1 Values(30,'Pune');

**To display records of employees from e1 table who have the same deptno as that of d1 table.**

select e1.deptno,e1.ename

from e1, d1

where e1.deptno = d1.deptno; **--- Odd Result**

select \* from e1

where **exists**

(select \* from d1

where e1.deptno = d1.deptno);

**To display records of employees from e1 table who do not have the same deptno as that of d1 table.**

select \* from e1

where **not** **exists**

(select \* from d1

where e1.deptno = d1.deptno);

## *Sub-Query in From Clause (Inline Views)*

**Example –**

**To see names, salaries, job, average salary and difference (raise) with average salary of those employees who earn more than the average salary in their jobs.**

**SELECT a.ename, a.sal, a.job, b.salavg, a.sal-b.salavg as "Raise"**

**FROM emp a, (SELECT job,**

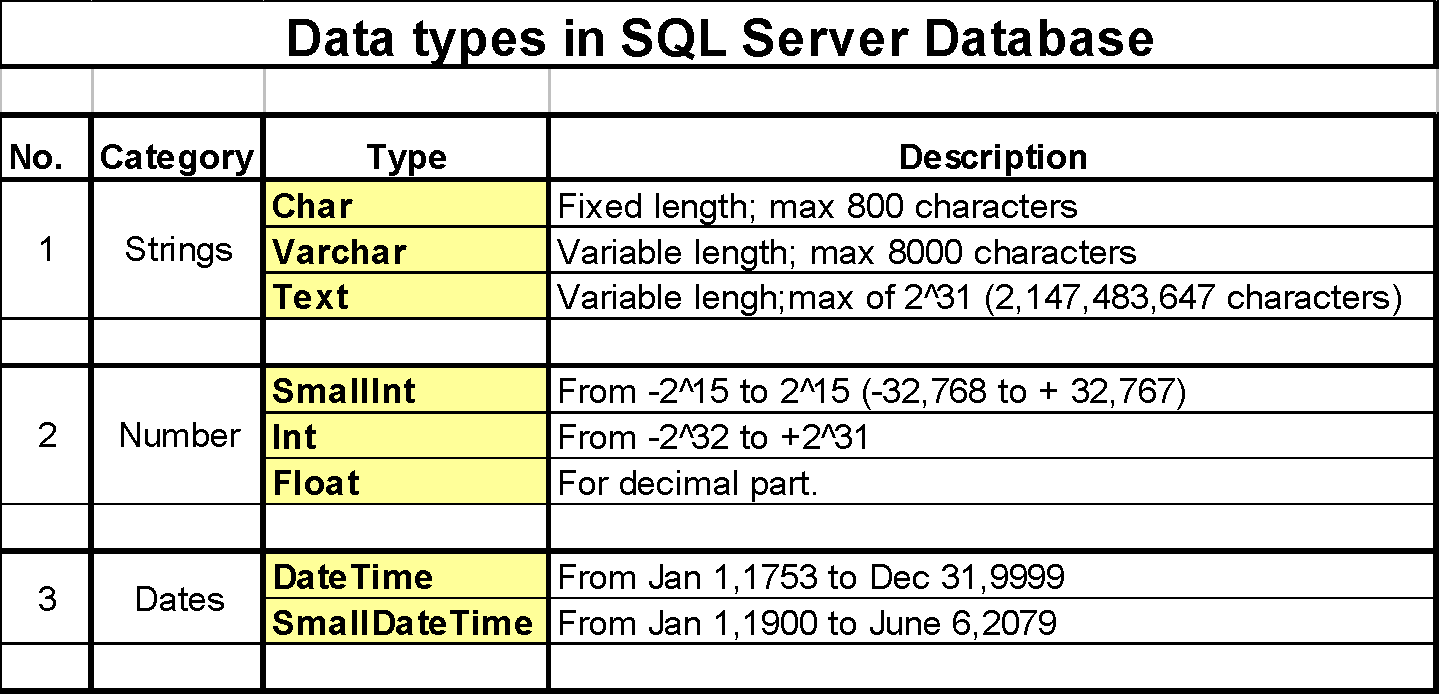
**AVG(sal) salavg**

### **FROM emp**

**GROUP BY job) b**

**WHERE a.job = b.job**

**AND a.sal > b.salavg;**

**2 DDL Commands**

**Create Table Statement**

# **Creating table with column level constraints**

create table student\_details

(**Roll\_No int primary key**,

Name varchar(30) not null,

Invoice\_No char(4) Unique,

Fees int check(fees > 5000),

City varchar(40) Default 'Pune')

# **Foreign key constraint**

create table result\_details

(**roll\_no int references student\_details(roll\_no),**

sem varchar(10),

marks int,

grade char(2))

# **Creating table with table level constraint**

Whenever two columns of the same table are getting compared or it is composite primary key then constraints have to be declared after column definitions.

create table cost\_details

(product\_id char(5),

product\_name varchar(30),

sales int,

tax int,

**check (sales>tax)**);

**Creating composite primary key**

create table first\_attempt

(rollno int,

sem char(7),

marks int,

**primary key(rollno,sem));**

**Creating foreign key on table level.**

CREATE TABLE Y1(A INTEGER PRIMARY KEY, B INTEGER UNIQUE, C INTEGER);

CREATE TABLE Y5

(A INTEGER,

Z INTEGER,

**FOREIGN KEY (A)** REFERENCES Y4)

**-- Giving names to the constraints**

create table t1

(rollno int **constraint pk1** primary key,

name varchar(30),

invno char(2) **constraint u2** unique)

Computed Columns

create table allowance\_details

(empid integer,

hra integer,

da integer,

ta integer,

total as (hra + da + ta)

)

Note –

Insert Into allowance\_details Values(1,900,1200,400)

Select \* from allowance\_details

* It is not possible to insert value in the total column and also the total column cannot be updated manually.

But when the values of the arguments are updated then the computed column gets refreshed implicitly.

* Only UNIQUE or PRIMARY KEY constraints are allowed on computed

**Foreign Key referring to the Primary or Unique key**

CREATE TABLE Y1(A INTEGER PRIMARY KEY, B INTEGER UNIQUE, C INTEGER);

CREATE TABLE Y2(A INTEGER REFERENCES **Y1**,D INTEGER);

CREATE TABLE Y3(B INTEGER REFERENCES **Y1(B),** E INTEGER)

columns.

CREATE TABLE Y5

(A INTEGER,

Z INTEGER,

FOREIGN KEY (A) REFERENCES Y4)

**Creating a new table from an existing table.**

1. Structure plus all the records.

select \* into new\_emp from emp;

1. Structure plus limited records

select \* into new\_emp1 from emp

where job = 'clerk';

1. Structure without records

select \* into new\_emp2 from emp

where 1=2;

1. Structure with different column names.

select empno as empid,ename as name,sal as salary

into new\_emp3 from emp

where 1=2

1. Summary Table.

select deptno, max(sal) **as highest\_sal** into new\_emp4

from emp

group by deptno;

*(Caption is must for the aggregate values)*

1. Data from multiple tables

select ename,dname into new\_emp5

from emp inner join dept

on emp.deptno = dept.deptno;

1. The constraints are not copied in the new table.Only Not Null status is applied to the new table.

create table tab1

(a integer primary key,

b integer unique,

c integer not null,

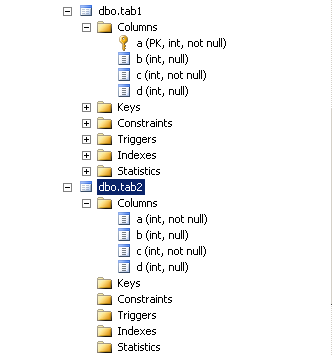
d integer check(d >= 0)

);

select \* into tab2 from tab1;

exec sp\_helpconstraint tab1;

exec sp\_helpconstraint tab2;



**Referential Integrity**

Drop table student1;

Drop table result1;

CREATE TABLE STUDENT1

(ROLL INTEGER PRIMARY KEY,

NAME VARCHAR(40));

CREATE TABLE RESULT1

(ROLL INTEGER REFERENCES STUDENT1(ROLL),

MARKS INTEGER);

insert into student1 values(1,'a');

insert into result1 values(1,99);

Now if the parent record is tried to delete then it will throw dependency error

**delete from student1 where roll = 1;**

Also if the parent record’s roll is tried to update then it will show dependency error.

**update student1**

**set roll = 55**

**where roll=1;**

To resolve these issues in the foreign key definition on delete cascade and on update cascade have to be specified.

Drop table result1;

Drop table student1;

CREATE TABLE STUDENT1

(ROLL INTEGER PRIMARY KEY,

NAME VARCHAR(40));

CREATE TABLE RESULT1

(ROLL INTEGER REFERENCES STUDENT1(ROLL) on delete cascade on update cascade,

MARKS INTEGER);

insert into student1 values(1,'a');

insert into result1 values(1,99);

**delete from student1 where roll = 1;**

**To drop the primary key constraint or to drop the primary key column the dependencies have to be taken care of.**

**alter table student1**

**drop column roll;**

**alter table student1**

**drop constraint PK\_\_STUDENT1\_\_4707859D;**

**Drop the foreign key constraint**

alter table result1

drop constraint FK\_\_RESULT1\_\_ROLL\_\_48EFCE0F;

**Identity**

Identity is used to generate auto numbers.

It is used to generate unique numbers.

Syntax --Identity(Seed, Increment)

Seed is the starting value

Increment is the step value (Can be negative)

Example –

Create table employees1

(EmpId int **Identity(1,1),**

EmpName Varchar(30));

Insert into employees1(EmpName) values('John');

Select \* from employees1;

*Seed and increment values should be only of INTEGER data type.*

*Only one identity column per table is allowed.*

**Set Identity\_Insert *tablename* ON** clause

Allows explicit values to be inserted into the identity column of a table.

create table X(y integer identity, z char);

Insert into X(z) Values('A');

Insert into X(z) Values('B');

select \* from X;

**Insert into X(y,z) Values(8,'C');**

**SET IDENTITY\_INSERT X ON**

**Insert into X(y,z) Values(8,'C')**

***It is must to mention the column list after the table name***

**To see the last identity value use IDENT\_Current built-in**

**SELECT IDENT\_CURRENT('X');**

SET IDENTITY\_INSERT X OFF

Insert into X(z) Values('C');

SELECT \* FROM x; -- Y HAS GOT 9 VALUE

**RULE**

Rules are similar to check constraints.

# **Salient features of rules –**

1. Many check constraints can be applied to one column, but only one rule can be applied to one column.
2. Check constraints are either defined at the time of table creation, or after table creation by Alter statement. Check constraints are dedicated for that specific table. But rules are created as separate objects and then bound to the column.
3. When the rule is bound to the column, then the rule cannot be dropped. First the rule needs to unbind from the column.
4. Advantage – Reusability. The same rule can be used for columns of many tables without need of defining it.

create table northsales(month varchar(10), cost integer);

create rule r1 as @cost\_value >= 100

Binding the rule R1 to the column cost of northsales table –

**sp\_bindrule** r1, 'northsales.cost'

insert into northsales values('Jan',**99**); **-- Throws error**

insert into northsales values('Jan'**,101**); -**- Row inserted**

To unbind the rule from the column

**sp\_unbindrule** 'northsales.cost'. – No need to specify the rule name.

To drop a rule

drop rule r1

A rule cannot reference columns or other database objects

*`condition\_expression* includes one variable. The at sign (@) precedes each local variable.

*When a rule is attached to a column it does not check the old values of that column.*

**ALTER Statement**

create table emp\_data

(empid integer,

name varchar(40));

**Adding column**

alter table emp\_data

add address varchar(30)

# **Dropping a column**

alter table emp\_data

drop **column** address

**Changing the data type**

alter table emp\_data

**alter column** empid varchar

*(If data is present in the column then still from integer to varchar conversion is possible, but varchar to integer is not possible)*

# **Adding constraint**

alter table emp\_data

**add constraint** **u1** unique(empid)

# **Dropping constraint**

Alter table emp\_data

Drop constraint u1

# **Dropping table**

Drop table emp\_data

**Constraints – Enable and Disable**

**Disabling the constraint –**

**Drop table Sales;**

create table sales(sid integer, amt integer constraint ch\_amt check(amt >= 100));

Insert Into Sales Values(1,800);

Insert Into Sales Values(2,600);

Alter Table Sales **NOCHECK** CONSTRAINT ch\_amt;

Insert Into Sales Values(3,**35**);

Alter Table Sales **CHECK** CONSTRAINT ch\_amt;

Insert Into Sales Values(4,**35**);

**Adding the constraint later --**

create table sales1(sid integer, amt integer);

Insert Into Sales1 Values(1,800);

Insert Into Sales1 Values(2,600);

**Alter table sales1 add constraint ch\_amt1 check(amt >= 1000); -- Checks the past data**

**Alter table sales1 WITH NOCHECK add constraint ch\_amt1 check(amt >= 1000);**

**Insert Into Sales1 Values(3,700);**

**Insert Into Sales1 Values(4,1001);**

# **2.7 Temporary table**

There are two types of temporary tables.

1. Local temporary table
2. Global temporary table

1 Local temporary table: The scope of Local temporary table is limited to the current session only. It is created by prefixing single # to the table name.

Example of local temporary table.

create table #city\_records

(city\_id int,

city\_name varchar(40),

sales int);

insert into #city\_records values(1,'Pune',9000);

Select \* from #city\_records;

**Open another session**

**Select \* from #city\_records;**

**But the city\_records table is limited to the first session. Once the session is closed then the table is no more.**

2. Global temporary table : It is visible to all the sessions.

create table ##state\_records

(city\_id int,

sales\_summary int);

insert into ##state\_records values(1,89000);

select \* from ##state\_records

## ***Open another session***

**select \* from ##state\_records**

**If the first session is closed then the table gets dropped.**

**2.8 VIEWS**

* Views are virtual tables
* Views are always based on the tables
* Views are queries stored through which data goes in the underlying table
* Views do not actually store the data So for object definition of view the memory is required

**Advantage**: Due to views we can hide the original table and give some conditional access of records to the user

create view V1

as

select \* from emp

where sal >= 3500

Views can be based on data from more than one table through join.

create view v3

as

select ename,dname

from emp join dept

on emp.deptno = dept.deptno

Summary view –

create view v4

as

select deptno,max(sal) "Top Sal" from emp

group by deptno

*(Note – Must provide column name for the aggregate function max(sal))*

**With Check Option** – This clause is used to restrict data from the view to get entered in the base table. Also data cannot be updated through the view of the criteria specified.

create view v7

as

select \* from emp

where deptno = 10

WITH CHECK OPTION

**Will throw error if deptno is tried to update through view.**

**update v7**

**set deptno = 8**

**Will throw error if deptno entered through view is not 10.**

**insert into v7(empno,ename,deptno) values(12,'abc',30);**

**With Schema Binding option –**

**It prevents the table on which view is based getting dropped.**

**Two part naming convention has to be used.**

**Ownername.TableName**

**Create view clerk\_data**

**with schemabinding**

**as**

**select ename,sal,deptno from dbo.emp**

**where job = 'clerk'**

**drop table emp – Throws error**

# **2.9 Indexes**

# **Guidelines for creating Indexes**

**A column can be selected for indexing based on the following criteria**

Frequently searched column

If the table is large

**Do not index the column in the following cases**

If column is not frequently searched.

If the table is small

# **Clustered Index**

A Clustered index determines the storage order of data in a table (physical order)

A table can have only one clustered index

A clustered index is analogous to a telephone directory, which arranges data by last name

It is effective on columns that are accessed very frequently and Queries that return large result sets

# **Non-Clustered Index**

A non-clustered index specifies a logical ordering only.

A table can have multiple non-clustered indexes (max 249).

It is similar to a textbook index

The data is stored in one place and the index in another

# **Example of Non cluster Index**

Some books contain multiple indexes. For example, a gardening book can contain one index for the common names of plants and another index for the scientific names because these are the two most common ways in which the readers find information. The same is true for nonclustered indexes. You can define a nonclustered index for each of the columns commonly used to find the data in the table.

# **Composite Index**

A composite index consists of two or more columns indexed together

Maximum 16 columns can be combined together

Composite indexes require fewer overheads than single column indexes

# **Unique Index**

A Unique Index ensures that the indexed column contains no duplicate values

Both clustered and non-clustered indexes can be unique

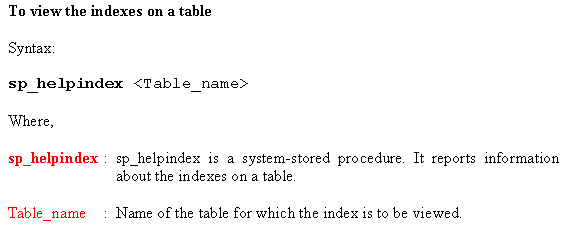
# **Considerations**

## ***Specifying a unique index makes sense only when uniqueness is a characteristic of the data***

## ***If uniqueness must be enforced to ensure data integrity, create a UNIQUE or PRIMARY KEY constraint on the column rather than a unique index***

## ***Creating a PRIMARY KEY or UNIQUE constraint automatically creates a unique index on the specified columns in the table***

# **Viewing Indexes**



**3.1** **Programming Basics**

1)

**declare**

@x integer

**begin**

set @x = 98

print @x

print 'Value of x variable is ' + cast(@x as varchar)

**end**

2) Initializing variable in the declare block

declare

@x integer

set @x = 9

begin

print @x

end

# **Taking value from the table**

declare

@x integer

begin

**select @x = sal from emp**

**where ename = 'SMITH'**

print @x

end

If condition is not given then shows the last record’s salary value from the table

declare

@x integer

begin

**select @x = sal from emp**

print @x

end

# **If Condition**

Syntax –

IF <condition>

[begin]

----

----

[end]

else

[begin]

-----

-----

[end]

**declare**

**@x integer**

**begin**

**set @x = 56**

**if @x >= 40**

**print 'Passed'**

**else**

**print 'Failed'**

**end**

# **Multiple statements in true and false block with begin and end**

declare

@x integer

begin

set @x = 56

if @x >= 40

**begin**

print 'Passed'

print 'Marks are ' + cast(@x as varchar)

**end**

else

**begin**

print 'Failed'

print 'Marks are ' + cast(@x as varchar)

**end**

end

**Multiple If’s If … else if ….. else ….**

declare

@x integer

begin

set @x = 41

if @x >= 70

print 'Dist'

else if @x >= 60

print 'FC'

else if @x >= 50

print 'SC'

else if @x >= 40

print 'Passed'

else

print 'Failed'

end

# **Nested if’s ……**

declare

@s integer,

@j varchar(9)

begin

select @s = sal from emp

where ename = 'SMITH'

**if @s <= 1000**

**begin**

select @j = job from emp

where ename = 'SMITH'

**if @j = 'MANAGER'**

**PRINT '30 PERCENT RAISE'**

**else**

**PRINT '20 PERCENT RAISE'**

**end**

**else**

PRINT 'NO RAISE'

**End**

**While Loop**

**To display 1 to 10**

declare

@w integer

begin

set @w = 1

**while @w <= 10**

**begin**

**print @w**

**set @w = @w + 1**

**end**

end

**break statement …. To terminate the loop**

declare

@w integer

begin

set @w = 1

**while @w <= 10**

**begin**

**if @w = 5**

**break**

**else**

**print @w**

**set @w = @w + 1**

**end**

print 'End of loop'

end

Drop table part;

create table PART (Part\_Id int, Category\_Id int, Description varchar(50));

**Continue statement …..**

**OUTPUT**

**Part\_Id Category\_Id Description**

----------- ----------- --------------------------------------------------

1 1 Part\_Id is 1 Category\_Id 1

1 3 Part\_Id is 1 Category\_Id 3

2 1 Part\_Id is 2 Category\_Id 1

2 2 Part\_Id is 2 Category\_Id 2

2 3 Part\_Id is 2 Category\_Id 3

To generate records in the table part. For part\_id 1 and category\_id 2 the record should not be inserted. So using **continue** clause for that condition.

**declare @Part\_Id int**

**declare @Category\_Id int**

**declare @Desc varchar(50)**

**set @Part\_Id = 0**

**set @Category\_Id = 0**

**while @Part\_Id < 2**

**begin**

**set @Part\_Id = @Part\_Id + 1**

**while @Category\_Id < 3**

**begin**

**set @Category\_Id = @Category\_Id + 1**

**If @Part\_ID = 1 and @Category\_ID = 2**

**Continue**

**set @Desc = 'Part\_Id is ' + cast(@Part\_Id as char(1)) +**

**' Category\_Id ' + cast(@Category\_Id as char(1))**

**insert into PART values(@Part\_Id, @Category\_Id,**

**@Desc )**

**end**

**set @Category\_Id = 0**

**end**

**GOTO Label:**

create table flight\_allowance(ename varchar(20));

create table rail\_allowance(ename varchar(20));

create table petrol\_allowance(ename varchar(20));

declare

@s integer,

@n varchar(40)

begin

delete from flight\_allowance;

delete from rail\_allowance;

delete from petrol\_allowance;

select @s = sal,@n = ename from emp

where ename = 'FORD'

IF @S >= 5000

**goto FA**

else if @s >= 3000

**goto RA**

else if @s >= 2000

**goto PA**

**FA:**

**Insert into flight\_allowance values(@n);**

**RA:**

**Insert into rail\_allowance values(@n);**

**PA:**

**Insert into petrol\_allowance values(@n);**

end

select \* from flight\_allowance;

select \* from rail\_allowance;

select \* from petrol\_allowance;

**3.2** **Cursors**

### **Scroll Attribute – It means that all fetch options such as absolute, next, prior, first,last, relative are available. If Scroll is not mentioned then only Next is available.**

declare c1 cursor **SCROLL**

FOR SELECT \* FROM EMP

OPEN C1

FETCH **ABSOLUTE** 1 FROM C1

FETCH ABSOLUTE 4 FROM C1

FETCH ABSOLUTE -1 FROM C1 -- From the last record

declare c2 cursor SCROLL

FOR SELECT \* FROM EMP

OPEN C2

FETCH ABSOLUTE 1 FROM C2

**FETCH NEXT FROM C2**

**FETCH NEXT FROM C2**

**FETCH PRIOR FROM C2**

# **FETCH relative 3 FROM C2**

**FETCH FIRST FROM C2**

## ***FETCH LAST FROM C2***

## ***To see the first ‘MANAGER’ record***

declare c3 cursor SCROLL

for select \* from emp

where job = 'MANAGER'

open c3

Fetch absolute 1 from c3

### **To see many records use LOOP**

declare c4 cursor SCROLL

for select \* from emp

where job = 'MANAGER'

open c4

Fetch First from c4

**while @@Fetch\_Status = 0**

Begin

Fetch next from c4

End

print 'The number of rows in the cursor are ' + cast**(@@cursor\_rows** as varchar)

**@@Fetch\_Status** is the global variable. It returns an integer value 0 for the last cursor fetch statement..i.e. After the last record is fetched it becomes not equal to zero.

**@@cursor\_rows** returns the number of qualifying rows that are in the currently opened cursor.

### **To see the first 2 highest salaries**

declare @n integer

Declare

c5 cursor SCROLL

**for select distinct sal from emp**

**order by sal desc**

set @n = 1

open c5

fetch first from c5

while @n <= 1

begin

fetch Next from c5

set @n = @n + 1

end

**To see the first 5 records**

declare @n integer

Declare c6 cursor SCROLL for select \* from emp

set @n = 1

open c6

fetch first from c6

while @n <= 4

begin

fetch Next from c6

set @n = @n + 1

end

### **To see the last 3 records**

declare @n integer

Declare

c7 cursor SCROLL

for select \* from emp

set @n = 1

open c7

fetch **Last** from c7

while @n <= 2

begin

fetch **Prior** from c7

set @n = @n + 1

end

##### ***STATIC***

**declare c10 cursor STATIC SCROLL**

**FOR SELECT \* FROM EMP**

**Open c10**

**update emp set sal = 3000 where ename = 'SMITH'-- Independent update**

# **FETCH ABSOLUTE 1 FROM C10 ---Shows the old salary of 800**

**DYNAMIC**

**declare c11 cursor DYNAMIC SCROLL**

**FOR SELECT \* FROM EMP**

## ***OPEN C11***

**update emp set sal = 2000 where ename = 'SMITH'**

**FETCH FIRST FROM C11 --- Shows the changed salary 2000**

***(Note -- The fetch type Absolute cannot be used with dynamic cursors.)***

###### **For Update …. Where Current Of Clause**

**Whenever multiple conditions are there for updating a table then instead of executing them separately and having a full table scan for every update statement, we can use where current of clause. The cursor will update each row as per the conditions.**

**declare @s integer**

**declare c12 cursor SCROLL DYNAMIC**

**for select sal from emp**

**for update of sal**

**open c12**

## ***Fetch First from c12 into @s***

**while @@Fetch\_Status = 0**

## ***Begin***

**if @s >= 5000**

**set @s = 10000**

**else if @s >= 4000**

**set @s = 9000**

**else if @s >= 3000**

**set @s = 8000**

**else if @s >= 2000**

**set @s = 7000**

**else if @s >= 1000**

**set @s = 6000**

**else if @s < 1000**

**set @s = 5500**

**update emp**

**set sal = @s**

**where current of c12**

**Fetch next from c12 into @s**

**End**

**For Update Of Sal means Sal column can be updated.**

**Where current of means the cursor will update the present record. It is known as positioned update**

**3.3 Stored Procedures**

# **Example 1 – Parameter less procedure**

create procedure p1

as

select \* from emp

Execute p1

OR Exec p1

# **Example 2 -- Procedure with parameters to create a new record**

create procedure p2( @eno integer, @name varchar(40), @dno integer)

as

insert into emp(empno,ename,deptno)

values(@eno, @name,@dno);

Print 'One record created'

exec p2 **123,'abc',10**

# **Example 3 – Default values for the last parameter(s)**

create procedure p3 @eno integer, @name varchar(40), **@dno integer = 20**

as

insert into emp(empno,ename,deptno)

values(@eno, @name,@dno);

Print 'One record created'

Exec p3 456,'def'

**Example 4 – Default values for first or mid parameter(s)**

create procedure p4 @eno integer, **@dno integer = 30**, @name varchar(50)

as

insert into emp(empno,ename,deptno)

values(@eno, @name,@dno);

Print 'One record created'

exec p4 789,**default**,'pqr'

# **Example 5 - Parameterized Query**

create procedure p5 @j varchar(50) = 'MANAGER'

as

select \* from emp

where job = @j

EXEC P5 'CLERK'

OR EXEC P5

# **Example 6 – Return Codes of Procedure**

**Creating a procedure that stores the sum of two values.**

**The procedure returns the answer to a outside local variable .**

create procedure p7 @x integer, @y integer

as

Declare @z integer

set @z = @x + @y

## ***Return @z***

**To execute the procedure ---**

**declare @a integer**

**exec @a = p7 3,5**

**select @a**

(Note - If the procedure is executed independently then the answer will not be displayed)

**Example 7 – Output Parameter**

**The procedure will return value**

In procedure p8 the ename will be supplied by the user as input parameter and the procedure will return the sal for that ename as the output parameter.

create procedure p8 @name varchar(50), @salary integer **output**

as

select @salary = sal

from emp

where ename = @name

# **To execute the procedure --**

Declare a local variable.

Associate it in the parameter list of the procedure while calling the procedure.

declare @x integer

exec p8 'KING', @x OUTPUT

SELECT @x

**ALTER PROCEDURE Statement**

Drop Table proc\_tab1;

Drop Procedure show\_proctab1;

create table proc\_tab1(a integer);

Insert into proc\_tab1 Values(1);

create procedure show\_proctab1

as

select a from proc\_tab1

exec show\_proctab1

-- now column a of proc\_tab1 table is renamed to b.

**exec show\_proctab1 -- Throws error**

**sp\_helptext show\_proctab1 – To get the code**

**alter procedure show\_proctab1**

**as**

**select b from proc\_tab1**

**exec show\_proctab1 – Works properly**

With Encryption Clause – Code gets encrypted.

create procedure p100

**with encryption**

as

select \* from emp

sp\_helptext p100 ….. Message of encryption

**3.4 Functions**

create function annual\_salary(@s integer)

returns integer

as

begin

return @s \*12

end

A function can be directly called through an select statement

select ename,sal,**dbo.annual\_salary(sal)** from emp

Function can be called in the Insert Statement

create table emp\_sal1(empid integer, gross integer, tax integer);

create function get\_Tax1(@s integer)

returns integer

as

begin

return @s \* 0.15;

end

Insert into emp\_sal1 Values(1,50000, **dbo.get\_Tax1(50000)**)

Function can be called in the Update Statement

Insert into emp\_sal Values(3,60000,null)

Update emp\_sal1

set tax = **dbo.get\_Tax(60000)**

where empid = 2

OR

Insert Into Emp\_Sal Values(3,40000,null)

**Inline table-valued function**

Inline functions can be used to achieve the functionality of parameterized views.

create function get\_emp(@j varchar(30))

**returns table**

as

return (select \* from emp

where job = @j)

select \* from dbo.get\_emp('CLERK')

Inline user-defined functions follow these rules:

* The RETURNS clause contains only the keyword **table**. You do not have to define the format of a return variable because it is set by the format of the result set of the SELECT statement in the RETURN clause.
* There is no *function\_body* delimited by BEGIN and END.
* The RETURN clause contains a single SELECT statement in parentheses.

**Calling a function for a computed column of a table**

create function get\_total(@n integer, @w integer, @s integer)

returns integer

as

begin

return @n + @w + @s

end

create table sales\_data

(salesid integer,

north integer,

west integer,

south integer,

**TOTAL AS (dbo.get\_total(north,west,south))**

)

Insert Into Sales\_Data Values(1, 800,900,1200)

select \* from sales\_data

**Since the total is non-deterministic so unique or primary keys cannot be created.**

**3.5** **Triggers**

DROP TABLE TAB1;

CREATE TABLE TAB1(A INTEGER);

DROP TABLE TAB2;

CREATE TABLE TAB2(B INTEGER);

CREATE TABLE TAB3(B INTEGER)

# **Example 1 -- Inserted row from tab1 should also get inserted in table tab2**

**create trigger trig1**

**on tab1**

**for insert**

**as**

**begin**

**insert into tab2**

**select \* from Inserted**

**end**

**insert into tab1 values(100);**

**select \* from tab2;**

Example 2 – If the new value inserted is < 100 then the record should not be inserted.(Conditional Insert)

**create trigger trig2**

**on tab1**

**for insert**

**as**

**begin**

**if (select a from inserted) < 100**

**begin**

**Print 'Value cannot be less than 100'**

**Rollback Tran**

**end**

**end**

**insert into tab1 values(99) --- Throws error**

**Example 3 -- Deleted row from tab1 should get inserted in table tab3**

**create trigger trig3**

**on tab1**

**for delete**

**as**

**begin**

**insert into tab3**

**select \* from Deleted**

**end**

**delete from tab1 where a = 100**

**select \* from tab3**

Example 4 – Not allowing records of 800 value to delete.(Conditional Delete)

**create trigger trig4**

**on tab1**

**for delete**

**as**

**begin**

**if (select a from deleted) = 800**

**begin**

**Print 'Cannot delete records of 800'**

**rollback tran**

**end**

**end**

**delete from tab1 where a = 800 --- Throws error**

insert into tab1 values(400);

insert into tab1 values(300);

Example 5 – To show how inserted and deleted tables of trigger work with Update statement

**create trigger trig5**

**on tab1**

**for Update**

**as**

**begin**

**Print 'Showing the updated values'**

**select \* from inserted**

## ***Print 'Showing the old values'***

**select \* from deleted**

**end**

**select \* from tab1;**

**update tab1**

**set a = 1000**

**where a = 400; *(Note – See the Results in Text)***

Example 6 – Table Level Update Trigger. It gets fired when any field from the row gets updated.

**create trigger trig6**

**on emp**

**for update**

**as**

**begin**

**if (select sal from inserted) > 10000**

**begin**

**Print 'Raise of sal cannot exceed 10000'**

**Rollback tran**

**end**

**end**

**update emp**

**set sal = 90000**

**where ename = 'KING' – Throws error**

Example 7 – Column Level Update Trigger. It gets fired only when a particular field from the row gets updated.

**create trigger trig7**

**on emp**

**for update**

**as**

**begin**

**if update(deptno)**

**begin**

**Print 'Dept number cannot be updated'**

**Rollback Tran**

**end**

**end**

**update emp**

**set deptno = 20**

**where ename = 'MARTIN' --- Throws error**

**(For more than one columns use OR operator.**

**if update(ename) or update(comm) )**

Example 8 – To ensure that more than 5 records cannot get deleted in one stroke.

**create trigger trig8**

**On Emp**

**for delete**

**as**

**if (select count(\*) from deleted) > 5**

**Begin**

**Print 'You cannot delete more than 5 records'**

**rollback tran**

**end**

**Delete from Emp ------ Throws error**

**create table lowcost(pid integer, cost integer);**

**create table highcost(pid integer, cost integer);**

**insert into lowcost values(1,900);**

**insert into lowcost values(2,1100);**

**insert into lowcost values(3,1300);**

Example 9 –Triggers referring to a column of another table.

**There are two table lowcost and highcost. The cost value in high cost table should be always more than all the cost of lowcost table.**

**create trigger trig9**

**on highcost**

**for Insert**

**as**

**Begin**

**if (select cost from inserted) < (select max(cost) from lowcost)**

**begin**

**Print 'The cost in high cost table cannot be less than the cost in low cost table'**

### **Rollback Tran**

**end**

**End**

**insert into highcost values(1,300); -- Throws error**

**insert into highcost values(1,1500); -- Row gets inserted**

**create table Emp\_Details(empid integer, name varchar(50), city varchar(20));**

**create table Emp\_Performance(empid integer, sales integer, profit integer, loss integer);**

**insert into emp\_details values(1,'John', 'Pune');**

**insert into emp\_details values(2,'Martin', 'Mumbai');**

**insert into emp\_performance values(1,7000,300,120);**

**insert into emp\_performance values(2,18000,7000,15);**

**select \* from emp\_details;**

**select \* from emp\_performance;**

**Example 10 – Cascade Triggers**

**Similar to cascade delete and cascade update options**

**When a record from Emp\_Details is deleted then the corresponding same record from Emp\_performance should also get deleted.**

**create trigger trig10**

**on Emp\_Details**

**for delete**

**as**

**Begin**

**delete Emp\_performance**

**from Deleted Inner Join Emp\_performance**

**on Emp\_performance.empid = Deleted.empid**

**End**

**delete from emp\_details**

**where name = 'John'**

**select \* from emp\_details;**

**select \* from emp\_performance;**

**Example 11 – Instead Of Triggers**

**In case of instead of triggers, the code contained inside the triggers gets executed in place of the original data manipulation statement.**

**When a record is tried to insert in emp\_performance table then instead of record getting inserted the system date and time should get inserted in check\_inserts table.**

**create table CHECK\_INSERTS(dates varchar(30))**

**CREATE TRIGGER TRIG11**

## ***ON EMP\_PERFORMANCE***

**INSTEAD OF INSERT**

## ***AS***

**BEGIN**

**INSERT INTO CHECK\_INSERTS**

**VALUES(getdate())**

**Print 'Cannot insert record'**

**END**

**insert into emp\_performance values(3,6000,90,6)**

**select \* from check\_inserts**

**Example 12 – Instead Of Trigger**

**Directly DML cannot be done on a view based on more than one tables. But using Instead of triggers it is possible.**

**Create table s(roll integer, name varchar(40));**

**Insert into s values(1,'A');**

**Insert into s values(2,'B');**

**Insert into s values(3,'C');**

**Insert into s values(4,'D');**

**create table r(roll integer,marks integer);**

**Insert into r values(2,90);**

**Insert into r values(3,98);**

**create view sr**

**as**

**select s.roll,s.name,r.marks**

**from s,r**

**where s.roll = r.roll**

**create trigger trig12 on SR**

#### **INSTEAD OF INSERT**

**as**

**begin**

**insert into s**

**select roll,name**

**from Inserted**

**insert into r**

**select roll,marks**

**from inserted**

**end**

**insert into sr values(5,'z',80);**

**select \* from sr;**

**select \* from s;**

**select \* from r;**

**Restricting DML on Sunday**

**create trigger trig11 on emp**

**for insert,delete,update**

**as**

**declare @v\_day varchar(20)**

**Begin**

**select @v\_day= datename(dw,getdate())**

**if @v\_day = 'Sunday'**

**begin**

**print 'No Transactions on Sunday'**

**rollback tran**

**end**

**end**

**Updated value should be always greater than the old value**

**create trigger ttt22 on emp**

**for update**

**as**

**begin**

**if(select sal from inserted) < (select sal from deleted)**

**begin**

**Print 'Salary cannot be decremented '**

**rollback tran**

**end**

**end**

**update emp**

**set sal = 1000**

**where ename = 'ALLEN'**

**When a new record is inserted then the cost should be greater than all the existing records cost value**

**Create Table Costing(Record\_id Integer, Cost Integer)**

**Insert Into Costing Values(1,900)**

**Create Trigger tx1 on Costing**

**for insert**

**as**

**declare @max\_cost integer**

**begin**

**select @max\_cost=max(cost) from costing**

**if (select cost from inserted) < @max\_cost**

**Begin**

**Print 'Cost Value has to be greater than all existing cost'**

**Rollback Tran**

**End**

**End**

**Insert Into Costing Values (2,800)**

**4. New Features of SQL Server 2005 –**

**4.1. DDL Triggers**

SQL Server 2005 has extended the trigger functionality you normally use with Data Manipulation Language (DML) commands such as INSERT, UPDATE, and DELETE to incorporate Data Definition Language (DDL) commands like CREATE DATABASE, DROP TABLE, and ALTER TABLE.

Also, like DML triggers, DDL triggers run in the same transaction as the DML statement. So for instance, when a DDL TSQL statement has completed, you can rollback a transaction whenever appropriate.

Unlike DML triggers, DDL triggers respond to completely different events. As previously stated, DDL triggers respond to changes to a database or a server. Each time DDL statements make changes using commands such as CREATE TABLE, an event is fired. DDL triggers can subscribe to those events and execute other TSQL instructions in response to the events.

Some other differences between DDL and DML triggers include the following:

* DDL triggers do not support the INSTEAD of functionality in the CREATE TRIGGER statement.
* DDL triggers are missing the inserted and deleted tables common to DML triggers.

**Example – Preventing a user from dropping or altering any table.**

**create trigger tr1**

**on database for drop\_table, alter\_table**

**as**

**PRINT 'WRONG'**

**ROLLBACK**

**sys.triggers and sys.trigger\_events are the data dictionary tables.**

* 1. **TOP clause Enhancements**

TOP was introduced in SQL Server 7. Until SQL Server 2005, the TOP clause allowed the user to specify the number or percent of rows to be returned in a SELECT statement. In SQL Server 2005, the TOP clause can be used also for INSERT, UPDATE, and DELETE (in addition to SELECT), and the syntax is as follows: TOP (expression) [PERCENT]. Notice the parentheses around the expression; this is required when TOP is used for UPDATE, INSERT, and DELETE.

CREATE TABLE toptest (col1 VARCHAR(150));

Insert Into TopTest Values('1');

Insert Into TopTest Values('2');

Insert Into TopTest Values('3');

Insert Into TopTest Values('4');

Insert Into TopTest Values('5');

SELECT TOP(2) \* FROM toptest;

New of 2005

UPDATE TOP(2) toptest SET col1 = '100';

select \* from toptest;

UPDATE TOP (50) percent toptest SET col1 = '500';

select \* from toptest;

DELETE TOP(2) toptest;

select \* from toptest;

Delete from toptest;

Insert Into TopTest Values('1');

Insert Into TopTest Values('2');

Insert Into TopTest Values('3');

Insert Into TopTest Values('4');

Insert Into TopTest Values('5');

Insert Into TopTest Values('6');

Insert Into TopTest Values('7');

Insert Into TopTest Values('8');

Insert Into TopTest Values('9');

Insert Into TopTest Values('10');

**Expression in the Top clause**

select top(select count(\*) from toptest where col1 <=5) \*

from emp

* 1. **DML OUTPUT clause**

The execution of a DML statement such as INSERT, UPDATE, or DELETE does not produce any results that indicate what was changed. Prior to SQL Server 2005, an extra round trip to the database was required to determine the changes. In SQL Server 2005 the INSERT, UPDATE, and DELETE statements have been enhanced to support an OUTPUT clause so that a single round trip is all that is required to modify the database and determine what changed. You use the OUTPUT clause together with the inserted and deleted virtual tables, much as in a trigger. The OUTPUT clause must be used with an INTO expression to fill a table. Typically, this will be a table variable. The following example creates a table, inserts some data, and finally deletes some records

CREATE TABLE outputtbl

(id INT IDENTITY, col1 VARCHAR(15));

INSERT INTO outputtbl VALUES ('row1');

INSERT INTO outputtbl VALUES ('row2');

INSERT INTO outputtbl VALUES ('row3');

INSERT INTO outputtbl VALUES ('row4');

INSERT INTO outputtbl VALUES ('row5');

INSERT INTO outputtbl VALUES ('row6');

INSERT INTO outputtbl VALUES ('row7');

INSERT INTO outputtbl VALUES ('row8');

INSERT INTO outputtbl VALUES ('row9');

INSERT INTO outputtbl VALUES ('row10');

Select \* from outputtbl;

**OUTPUT with DELETE**

--make a table variable to hold the results of the OUTPUT clause

-- delete two rows and return through the output clause

DECLARE @del AS TABLE (deletedId INT, deletedValue VARCHAR(15))

DELETE outputtbl

OUTPUT DELETED.id, DELETED.col1 INTO @del

WHERE id < 3

SELECT \* FROM @del

The example inserted the id and col1 values of the rows that were deleted into the table variable @del.

**OUTPUT with UPDATE**

When used with an UPDATE command, OUTPUT produces both a DELETED and an INSERTED table. The DELETED table contains the values before the UPDATE command, and the DELETED table has the values after the UPDATE command. An example follows that shows OUTPUT being used to capture the result of an UPDATE.

--update records, this populates

--both the inserted and deleted tables

DECLARE @changes TABLE

(id INT, oldValue VARCHAR(15), newValue VARCHAR(15))

UPDATE outputtbl

SET col1 = 'updated'

OUTPUT inserted.id, deleted.col1, inserted.col1

INTO @changes

WHERE id < 5

SELECT \* FROM @changes;

**OUTPUT with INSERT**

DECLARE @new\_record TABLE

(id INT, newValue VARCHAR(15))

Insert into outputtbl (col1)

OUTPUT inserted.id, inserted.col1 into @new\_record

Values ('row11');

select \* from @new\_record

**4.5 Pivot**

**Pivot** -- It is a summary table.

1. One of the column’s distinct values will be the column headings
2. One of the column’s distinct values will be the row headings.
3. In the intersection (cell) the aggregation of the third column’s value will be done.

**Example 1 –**

1. Create a table P\_Emp from the 3 columns of emp table – sal,deptno and job.

SELECT SAL,DEPTNO,JOB INTO P\_EMP FROM EMP;

Generating a report in which Job values will be the column heading, the deptno values will be row headings and the total (sum) of salaries for the combination of job and deptno will be the data.

select \* from P\_emp

**PIVOT**

(

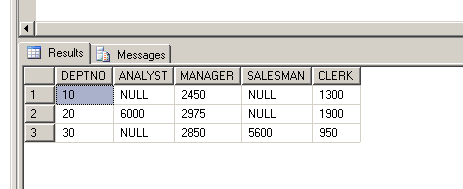
SUM(Sal)

For JOB in (ANALYST,MANAGER,SALESMAN,CLERK)

)

AS P;

Output --



To get summary for a particular deptno use the where clause after the pivot

select \* from P\_emp

PIVOT

(

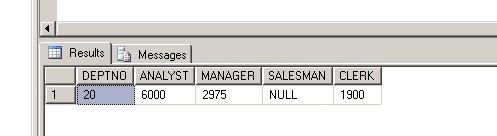
SUM(Sal)

For JOB in (ANALYST,MANAGER,SALESMAN,CLERK)

)

AS P

where deptno = 20



Example 2—

CREATE TABLE SALES

(

[Year] INT,

Quarter CHAR(2),

Amount FLOAT

)

INSERT INTO SALES VALUES (2001, 'Q2', 70)

INSERT INTO SALES VALUES (2001, 'Q3', 55)

INSERT INTO SALES VALUES (2001, 'Q3', 110)

INSERT INTO SALES VALUES (2001, 'Q4', 90)

INSERT INTO SALES VALUES (2002, 'Q1', 200)

INSERT INTO SALES VALUES (2002, 'Q2', 150)

INSERT INTO SALES VALUES (2002, 'Q2', 40)

INSERT INTO SALES VALUES (2002, 'Q2', 60)

INSERT INTO SALES VALUES (2002, 'Q3', 120)

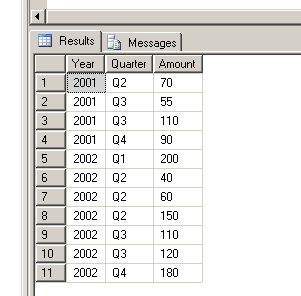
INSERT INTO SALES VALUES (2002, 'Q3', 110)

INSERT INTO SALES VALUES (2002, 'Q4', 180)

Select \* from sales

order by year, quarter, amount;

Original Data Sorted –



To get the sum of Amount for each Quarter within each year.

Quarter values will be column headings, year values will be row headings and sumation of amount will be done.

SELECT \* FROM SALES

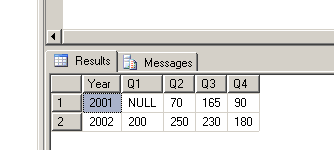
PIVOT

(SUM (Amount)

FOR [Quarter]

IN (Q1, Q2, Q3, Q4))

AS P;



**Example 3 –**

Pivot report can also have only column headings (*No row values*)

For that the from clause query has to be used and only two columns have to be mentioned, one column or aggregation and the other column for the column headings.

SELECT \* FROM (select quarter,amount from sales) a

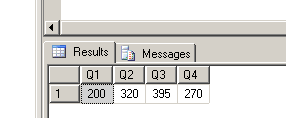
PIVOT

(SUM (Amount)

FOR [Quarter]

IN (Q1, Q2, Q3, Q4))

AS P;



Example 4-

If a table has more than 3 columns then the from clause query should be used to select the 3 columns for the pivot table.

Pivot on the emp table.

select \* from

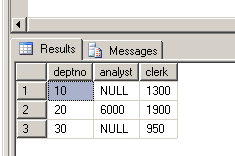
(select sal,job,deptno from emp) as A

pivot

(sum(sal) for job in (analyst,clerk)

)

as P;



Example 5 –

Genearting pivot report on a join query.

select \* from

(select dname, job, sal

from emp inner join dept

on emp.deptno = dept.deptno) as A

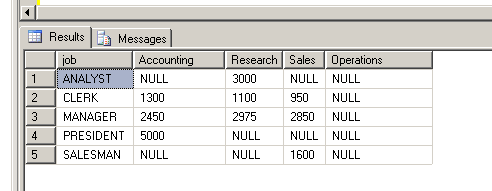
pivot

(

max(sal) for dname in (Accounting, Research,Sales, Operations)

)

as P



**4.5. Common Table Expression (CTE)**

SQL Server 2005 significantly enhances both the functionality and performance of SQL to address the requirements of business intelligence queries. The SELECT statement’s WITH clause, introduced in SQL Server 2005, provides powerful new syntax for enhancing query performance. **It optimizes query speed by eliminating redundant processing in complex queries.**

Consider a lengthy query that has multiple references to a single sub query block. Processing sub query blocks can be costly, so re-computing a block every time it is referenced in the SELECT statement is highly inefficient. The WITH clause enables a SELECT statement to define the sub query block at the start of the query, process the block just once, label the results, and then refer to the results multiple times.

The WITH clause, formally known as the sub query factoring clause, is part of the SQL-99 standard. The clause precedes the SELECT statement of a query and starts with the keyword “WITH.” The WITH is followed by the sub query definition and a label for the result set. The query below shows a basic example of the clause:

Query1 – To display maximum salaries department number wise for the department numbers having max salary greater than the max salary of department number 20.

select deptno, max(sal)

from emp

group by deptno

having max(sal) > (select max(sal)

from emp

where deptno = 20);

In the above query there is lot of performance overhead due to the following factors:

1. Initially the max(sal) for deptno 20 is calculated.
2. Once the max(sal) is calculated and returned by the sub query then again the parent query will do the job of finding the max(sal) deptno wise and compare with the value given by max(sal) of deptno 20.

**with summary as**

**( select max(sal) as highest, deptno**

**from emp**

**group by deptno)**

**select deptno, highest**

**from summary**

**where highest > (select highest**

**from summary**

**where deptno = 20);**

A temporary table summary gets created which does the job of finding deptno wise highest salaries. Using this summary table then simply the max(sal) of deptno 20 is filtered. Here the aggregation is done only once.

Query 2- To list the Sum of Salaries for departments comprising more than 1/3 of the firm's annual salary.

select dname,sum(sal) as DEP\_TOTAL

from emp,dept

where emp.deptno = dept.deptno

group by dname

having sum(sal) >

(select sum(sal) \* 1/3

from emp

);

**with summary as**

**(select dname,sum(sal) as DTOTAL**

**from emp,dept**

**where emp.deptno = dept.deptno**

**group by dname)**

**select dname, DTOTAL**

**from summary**

**where DTOTAL >**

**(select sum(DTOTAL) \* 1/3**

**from summary);**

**The SQL WITH clause in SQL SERVER 2005 significantly improves performance for complex business intelligence queries.**

**6. Analytical functions**

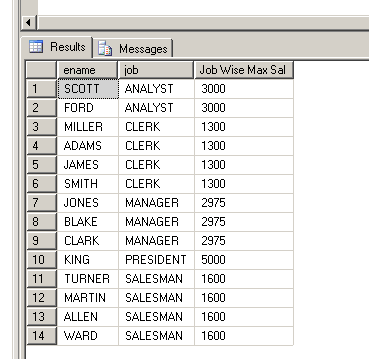
Though analytic functions give aggregate result they do not group the result set. They return the group value multiple times with each record. As such any other non-"group by" column or expression can be present in the select clause.

Partition by will do aggregation and display it for all the records (After aggregating the value grouping is not done)

**Example 1 - Over (Partition by) clause**

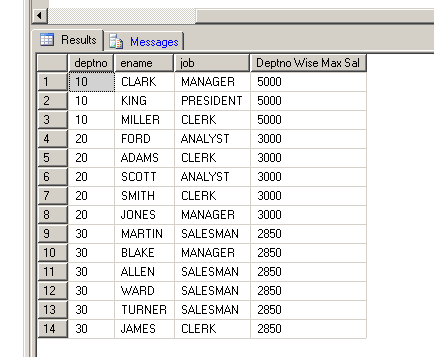
To see name, job, salary and maximum salary (repeating).

**select ename, job, max(sal) over (partition by job) "Job Wise Max Sal" from emp;**



# **Example 2**

**select deptno,ename, job, max(sal) over (partition by deptno) "Deptno Wise Max Sal" from emp**

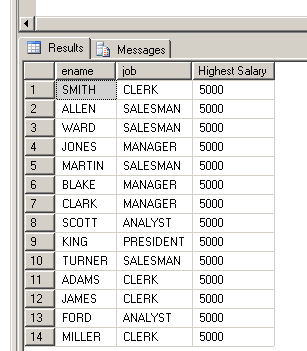


# **Example 3 - Over () clause [Without partition clause]**

In absence of any PARTITION inside the OVER( ) portion, the function acts on entire record set returned by the where clause.

**SQL> select ename, job, max(sal) over () "Highest Salary" from emp;**

***(The max(sal) value is repeated for all the rows)***



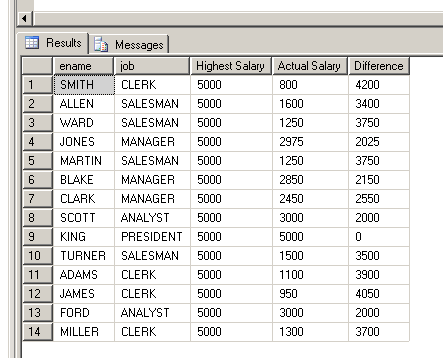
# **Example 4 – To perform calculations with aggregate values and actual values**

To see the difference in the max(sal) and sal for each employee.

**select ename, job, max(sal) over () "Highest Salary", sal "Actual Salary",**

**max(sal) over() - sal "Difference"**

**from emp;**



RANK and DENSE\_RANK both provide rank to the records based on some column value or expression. In case of a tie of 2 records at position N, RANK declares 2 positions N and skips position N+1 and gives position N+2 to the next record. While DENSE\_RANK declares 2 positions N but does not skip position N+1.

**Rank()**

select ename, sal, deptno,

rank() over(partition by deptno order by sal desc) "Rank" from emp

order by 3,4

ENAME SAL DEPTNO Rank

---------- ---------- ---------- ----------

KING 5000 10 1

CLARK 2450 10 2

MILLER 1300 10 3

SCOTT 3000 20 1

FORD 3000 20 1

JONES 2975 20 3

ADAMS 1100 20 4

SMITH 800 20 5

BLAKE 2850 30 1

ALLEN 1600 30 2

TURNER 1500 30 3

WARD 1250 30 4

MARTIN 1250 30 4

JAMES 950 30 6

For DEPTNO 20 there are two contenders for the first position Scott and Ford. So it has given the same number 1 for those but has skipped 2 and directly given rank number 3 for Jones of the same deptno 20.

Same case is for 3 records of deptno 30.

**Dense\_Rank()**

select ename,sal,deptno,

dense\_rank() over(partition by deptno order by sal desc) "Dense Rank" from emp

order by 3,4

/

ENAME SAL DEPTNO Dense Rank

---------- ---------- ---------- ----------

KING 5000 10 1

CLARK 2450 10 2

MILLER 1300 10 3

SCOTT 3000 20 1

FORD 3000 20 1

JONES 2975 20 **2**

ADAMS 1100 20 3

SMITH 800 20 4

BLAKE 2850 30 1

ALLEN 1600 30 2

TURNER 1500 30 3

WARD 1250 30 4

MARTIN 1250 30 4

JAMES 950 30 **5**

For DEPTNO 20 there are two contenders for the first position Scott and Ford. So it has given the same number 1 for those and directly given rank number 2 for Jones of the same deptno 20.

Same case is for 3 records of deptno 30.

**Row\_Number()**

It will provide the row numbers for the result set once the records are sorted.

Order By is the mandatory clause required for row\_number().

select ename, row\_number() over(order by ename) from emp;



* 1. **Set operators Enhancements**

**INTERSECT**

-- Shows the common values from the queries.

-- It eliminates the duplicates

-- To see the common products of 107 and 108

select prodname

from pune

where custid =107

INTERSECT

select prodname

from pune

where custid =108;

**EXCEPT**

-- Shows the records retrieved from the first query which are not present in the

-- second query

--Eliminates duplicates

-- To see products sold to 107 and not to 108

select prodname

from pune

where custid =107

EXCEPT

select prodname

from pune

where custid =108

**4**.**8. Referential Integrity Enhancements**

**Set Null and Set Default are new options availabe with On Delete and On Update.**

**On Delete Set Null – Whenever the parent record is deleted then the corresponding child table(s) foreign key will become null. But the child record is not deleted.**

**Drop table ri2;**

**Drop table ri1;**

**create table ri1(a integer primary key);**

**create table ri2(a integer references ri1 on delete set null, b integer);**

**Insert Into ri1 Values(1);**

**Insert into ri2 Values(1,100);**

**delete from ri1;**

**select \* from ri2;**

**On Delete Set Default – Whenever the parent record is deleted then the corresponding foreign key value gets the default value and the child record is intact.**

**create table ri3(a integer primary key);**

**create table ri4(a integer default 1,**

**foreign key(a) references ri3 on delete set default, b integer);**

**Insert into ri3 values(1);**

**Insert Into ri3 values(2);**

**Insert Into ri4 values(1, 100);**

**Insert into ri4 Values(2, 100);**

**Delete from ri3 where a = 2;**

**select \* from ri4;**

**Delete from ri3 where a = 1;**

**4.9 Try Catch**

SQL Server versions before 2005 offered only one simple way to work with exceptions: the @@ERROR function. This function can be used to determine if an error occurred in the last statement that was executed before evaluating @@ERROR. For example:

SELECT 1/0  
SELECT @@ERROR

-----------  
Msg 8134, Level 16, State 1, Line 1  
Divide by zero error encountered.   
  
-----------  
8134

(1 row(s) affected)

In this case @@ERROR returns 8134, which is the error number for a divide-by-zero error.

Using @@ERROR, you can detect errors and control them to some degree. However, proper use of this function requires that you check it after every statement; otherwise it will reset, as shown in the following example:

SELECT 1/0

IF @@ERROR <> 0

BEGIN

SELECT @@ERROR

END

-----------  
Msg 8134, Level 16, State 1, Line 1  
Divide by zero error encountered.   
  
-----------  
0

(1 row(s) affected)

Trying to catch the error in this case actually ends up resetting it; the @@ERROR in the SELECT returns 0 rather than 8134 because the IF statement did not throw an exception.

In addition to the fact that the exception resets after each statement, @@ERROR does not actually *handle* the exception -- it only reports it. The exception is still sent back to the caller, meaning that even if you do something to fix the exception in your T-SQL code, the application layer will still receive a report that it occurred. This can mean additional complexity when creating application code because you need to handle exceptions that may needlessly bubble up from stored procedures.

In SQL Server 2005, exceptions can now be handled with a new T-SQL feature: TRY/CATCH blocks. This feature emulates the exception handling paradigm that exists in many languages derived from the C family, including C/C++, C#, Java and JavaScript. Code that may throw an exception is put into a *try* block. Should an exception occur anywhere in the code within the try block, code execution will immediately switch to the *catch* block, where the exception can be handled.

The term "catch" is of special importance here. When TRY/CATCH is used, the exception is not returned to the client. It is "caught" within the scope of the T-SQL that caused it to be thrown.

For an example of TRY/CATCH, consider a divide-by-zero error:

BEGIN TRY

SELECT 1/0

END TRY

BEGIN CATCH

SELECT 'Error Caught'

END CATCH

-----------

(0 row(s) affected)   
  
------------  
Error Caught

(1 row(s) affected)

When this batch is run, no exception is reported. Instead, the message "Error Caught" is selected back. Of course, your T-SQL code does not have to send back any kind of specific message in the CATCH block. Any valid T-SQL can be used, so you can log the exception or take action to remedy the situation programmatically, all without reporting it back to the caller.

While merely being able to catch an exception is a great enhancement, T-SQL is also enhanced with new informational functions that can be used within the CATCH block. **These functions are: ERROR\_MESSAGE(), ERROR\_NUMBER(), ERROR\_LINE(), ERROR\_SEVERITY(), ERROR\_STATE() and ERROR\_PROCEDURE().** Unlike @@ERROR, the values returned by these functions will not reset after each statement and, as a result, the functions will return consistent values over the entire time a CATCH block is executed. For instance:

BEGIN TRY

SELECT 1/0

END TRY

BEGIN CATCH

SELECT 'Error Caught'

SELECT

ERROR\_MESSAGE(),

ERROR\_NUMBER()

END CATCH

-----------

(0 row(s) affected)   
  
------------  
Error Caught

(1 row(s) affected)

-------------------------------------------- ---------------  
Divide by zero error encountered. 8134

(1 row(s) affected)

Error control is important in database programming because it gives you the ability to roll back transactions in response to problems. By default, SQL Server typically does not stop transactions due to exceptions, which can result in invalid data. Consider the following batch:

CREATE TABLE Funds

(

Amount INT

CHECK (Amount > 0)

)

BEGIN TRANSACTION

INSERT Funds VALUES (10)

INSERT Funds VALUES (-1)

COMMIT TRANSACTION

SELECT \*

FROM Funds

(1 row(s) affected)  
Msg 547, Level 16, State 0, Line 9  
The INSERT statement conflicted with the CHECK constraint "CK\_\_Funds\_\_Amount\_\_67A95F59". The conflict occurred in database "master", table "dbo.Funds", column 'Amount'.  
The statement has been terminated.  
Amount  
-----------  
10

(1 row(s) affected)

In this case, a table called Funds is created, which includes a CHECK constraint on the Amount column to ensure that amounts are greater than 0. Once the table is created, a transaction starts. This implies that INSERTs will be atomic -- all values or no values should be inserted. However, even though an exception occurs due to violation of the CHECK constraint, the transaction is committed and one of the values remains in the table.

Implementing TRY/CATCH in this situation solves the problem outright:

BEGIN TRY

BEGIN TRANSACTION

INSERT Funds VALUES (10)

INSERT Funds VALUES (-1)

COMMIT TRANSACTION

END TRY

BEGIN CATCH

ROLLBACK

END CATCH

Now, any exception in the TRY block immediately causes code execution to shift to the CATCH block, thereby rolling back the transaction and ensuring that invalid values stay out of the table.

Exception handling is new to SQL Server, so the question of when to handle exceptions may be new to many DBAs and database developers. Here I will supply a few general guidelines to help you get started.

1. Overuse is much better than underuse when it comes to dealing with exceptions. As illustrated in the transaction example, failure to properly handle exceptions when they occur leaves you with invalid data in the database. Imagine a database being used to back financial transactions and think of the possibilities. Exception handling is an absolute necessity when you care about the quality of your data.

2. Strive to use a TRY/CATCH block whenever you use an explicit transaction and whenever you modify data. Some practitioners advocate using TRY/CATCH blocks in every stored procedure in order to log any exception that occurs in the database. Although this seems like overkill for some applications, it can be a good model for applications that require extreme integrity. Again, consider financial transactions.

3. Even though you may often use TRY/CATCH to facilitate structured exception logging, try to remember that not getting exceptions at all is far more desirable than just catching them when they occur. Heavily test your code and the code around problems you know exist, rather than letting exception handlers deal with them for you. Just because the exception is caught does not mean that it didn't occur. Exception handling is no excuse for sloppy coding techniques. If anything, it should give you a chance to more readily discover where your problems lie and fix them.

**5 Database Architecture**

SQL Server 2005 maps a database over a set of operating-system files. Data and log information are never mixed in the same file, and individual files are used only by one database. Filegroups are named collections of files and are used to help with data placement and administrative tasks such as backup and restore operations.

# **Database Files**

SQL Server 2005 databases have three types of files:

* Primary data files   
    
  The primary data file is the starting point of the database and points to the other files in the database. Every database has one primary data file. The recommended file name extension for primary data files is .mdf.
* Secondary data files   
    
  Secondary data files make up all the data files, other than the primary data file. Some databases may not have any secondary data files, while others have several secondary data files. The recommended file name extension for secondary data files is .ndf.
* Log files   
    
  Log files hold all the log information that is used to recover the database. There must be at least one log file for each database, although there can be more than one. The recommended file name extension for log files is .ldf.

SQL Server 2005 does not enforce the .mdf, .ndf, and .ldf file name extensions, but these extensions help you identify the different kinds of files and their use.

In SQL Server 2005, the locations of all the files in a database are recorded in the primary file of the database and in the **master** database. The Database Engine uses the file location information from the **master** database most of the time.

# **Database File groups**

Database objects and files can be grouped together in file groups for allocation and administration purposes. There are two types of file groups:

**Primary**

The primary file group contains the primary data file and any other files not specifically assigned to another file group. All pages for the system tables are allocated in the primary file group.

**User-defined**

User-defined file groups are any file groups that are specified by using the FILEGROUP keyword in a CREATE DATABASE or ALTER DATABASE statement.

Log files are never part of a file group. Log space is managed separately from data space.

No file can be a member of more than one file group. Tables, indexes, and large object data can be associated with a specified file group.