CSE 302
Database Management Systems
Sessional

Lab - 7

GROUP FUNCTIONS

Group Function

- Also known as "Multiple-Row Functions".
- They operates on set of rows to give one result per group.
- These set may be the whole table or the table split into groups.
- These are similar to the "aggregate functions" or "Group By" functions in Access

Group Functions

- SUM
- AVG
- COUNT
- MIN
- MAX

Group Functions

- **GROUP BY clause**
  - To identify groups of records to be processed
- **ORDER BY clause**
  - To sort the records
- **HAVING clause**
  - To restrict the groups displayed

```
SELECT * | column1, column2, ...
FROM tableName
WHERE Condition
GROUP BY column1, column2, ...
ORDER BY cloumn1, column2,...
HAVING group condition
```

Group Functions

SUM function

- Calculates the total amount in a numeric field for a group of records.
    - SUM(n) - where n is a numeric column
    - SUM(ALL n) - the same as above
    - SUM(DISTINCT n) - returns only the unique numeric values

SUM function

- *Display total salary of all employees.*

```
SELECT SUM(Salary) "Total Salary"
FROM Employee;
```

Total Salary
1160900

SUM function

- *Display total salary of the employees of e_city_001.*

```
SELECT SUM(Salary) "Total Salary of E_CITY_001"
FROM Employee
WHERE Employee_city=’e_city_001’;
```

Total Salary of E_CITY_001
32000

AVG function

- AVG(column containing numeric data)

- AVG(DISTINCT [column containing numeric data])
  - DISTINCT keyword returns only unique values

AVG function

- *Display average salary of all employees.*

```
SELECT AVG(Salary) " Average Salary"
FROM Employee;
```

Average Salary
116090

AVG function

- *Display average salary of the employees of e_city_001.*

```
SELECT AVG (Salary) " Average Salary of E_CITY_001"
FROM Employee
WHERE Employee_city=’e_city_001’;
```

Average Salary of E_CITY_001
32000

MAX and MIN function

- Returns the largest and smallest values in a specified column.

- MAX(ALL c) or MIN(ALL c)
  - where c is any numeric, character, or date field

- MAX(c) or MIN(c)
  - the same result as above

- MAX(DISTINCT c) or MIN(DISTINCT c)
  - returns the highest or lowest distinct value

MAX and MIN function

- *Display the maximum salary of employees.*

```
SELECT MAX(Salary) "Highest Salary"
FROM Employee;
```

- *Display the minimum DOB of employees.*

```
SELECT MIN(EMPLOYEE_DOB)
FROM EMPLOYEE;
```

COUNT function

- Counts the records that have non-NULL values
- Counts the total records meeting a specific condition

COUNT function

- *Display the count of cities.*

```
SELECT COUNT(EMPLOYEE_CITY)
FROM EMPLOYEE;
```

  - This counts all categories (including duplicates)

- *Display the count of unique cities only.*

```
SELECT COUNT(DISTINCT  EMPLOYEE_CITY)
FROM EMPLOYEE;
```

  - This counts unique (or distinct) categories

Group functions and NULL values

- All Group functions except COUNT(*) ignore null values in the column.

COUNT Function – NULL Values

- Including the NULL values
  - COUNT(*) counts all the records, even NULLS
  - Whenever NULL values may affect the COUNT the function, use an * as the argument, rather than a column name.

```
SELECT COUNT(*)
FROM EMPLOYEE;
```

GROUP BY Clause

GROUP BY Clause

- Divides the table of information into smaller groups.

```
SELECT …..
FROM …..
GROUP BY column1, column2,… ;
```

GROUP BY Clause

- *Divide the Employee table into groups by City. Then calculate the average salary for each group.*

```
SELECT Employee_city, Avg(Salary)
FROM Employee
GROUP BY Employee_city;
```

- The query execution goes like this:
  - The records in the Employee table are grouped by City
  - The average Salary for each City is calculated.

GROUP BY Clause

EMPLOYEE_CITY	AVG(SALARY)
e_city_001	32000
e_city_002	28500
e_city_003	20000
e_city_004	15300
e_city_005	16500
e_city_006	15700
e_city_007	900000
e_city_008	50900
e_city_009	41000

GROUP BY Clause

- *Display the Sum of All Balance of the Same City according to their account type.*

```
SELECT Cust_city, SUM(Balance), Type
FROM Customer JOIN Depositor USING
(CUST_ID) JOIN Account USING
(ACCOUNT_ID)
GROUP BY Cust_city, Type;
```

- The GROUP BY first groups the results by cust_city
- Then groups the Account TYPE within each customer City group.
- Then the SUM function calculates the Balance total.

GROUP BY Clause

CUST_CITY	SUM(BALANCE)	TYPE
Rye	1050	CURRENT
Brooklyn	750	SAVINGS
Harrison	1800	CURRENT
Stamford	700	CURRENT
Stamford	750	Savings
Palo alto	750	SAVINGS
Pittsfield	750	CURRENT
Pittsfield	750	SAVINGS

ORDER BY Clause

ORDER BY Clause

- *Divide the Employee table into groups by City. Then calculate the average salary for each group and order the result by average salary.*

```
SELECT Employee_city, Avg(Salary)
FROM Employee
GROUP BY Employee_city
ORDER BY Avg(Salary);
```

- *Order by Descending order-*

```
SELECT Employee_city, Avg(Salary)
FROM Employee
GROUP BY Employee_city
ORDER BY Avg(Salary) DESC;
```

HAVING Clause

HAVING Clause

- To further restrict groups returned by a query (Specifies which groups will be returned)
- Use a HAVING clause instead of a WHERE clause when group functions are involved.

HAVING(condition)

HAVING Clause

- *Display the cust_city, total balance and account type of customers by grouping them according to their city and account type with total balance>1000.*

```
SELECT Cust_city, SUM(Balance), Type
FROM Customer NATURAL JOIN Depositor
                    NATURAL JOIN Account
GROUP BY Cust_city, Type
HAVING SUM(Balance)>1000;
```

HAVING Clause

CUST_CITY	SUM(BALANCE)	TYPE
Rye	1050	CURRENT
Harrison	1800	CURRENT

WHERE and HAVING

- Both can be used in the same query.

Display the cust_city, total balance and account type of customers WHO HAVE BORN AFTER 1980 by grouping them according to their city and account type with total balance>1000.

```
SELECT Cust_city, SUM(Balance), Type
FROM Customer JOIN Depositor USING (CUST_ID) JOIN ACCOUNT
USING (ACCOUNT_ID)
WHERE Cust_dob > to_date('01-JAN-1981', 'DD-MON-YYYY')
GROUP BY Cust_city, Type
HAVING SUM(Balance)>1000;
```

WHERE and HAVING

- Both can be used in the same query.

Display the city, average salary of the employees WHO HAVE BORN AFTER 1980 by grouping them according to their city and account type with total balance>1000.

```
SELECT Employee_city, Avg(Salary)
FROM Employee
WHERE Employee_startdate>'01-JAN-81'
GROUP BY Employee_city
HAVING AVG(Salary)>1000;
```

Nesting Group Functions

- Group Functions can be nested to a **depth of two.**

```
SELECT Max(Avg(Salary))
FROM Employee
GROUP BY Employee_city;
```

Some general rules

- For using a mixture of individual items(Employee_city) and group functions (AVG) in the same SELECT statement, you must include a GROUP BY Clause that specifies the individual items.
- You can’t use WHERE Clause to restrict groups.
- You have to use the HAVING Clause to restrict groups.

Practice Problems
for
Group Functions

- **Write a query to display the number of customer with the same city.**

- **Display the Manager Number and the Salary of the lowest paid employee for that manager.**

- **Display the Manager Number and the difference between the highest and the lowest Salary of the employee for that manager.**

- **Display the minimum, maximum, sum and average salary for each group of employee having the same city.**

CONSTRAINTS

Constraints

- Constraints are rules to enforce business rules, practices, and policies

- **Why do we need constraints?**
  - To keep the database reliable.
  - To prevent a user from entering non-sensical data.
  - The business or other organization has certain rules that cannot be violated.

- Constraints are used for implementing the rules.

Reasons for using Constraints

- Enforce rules at the table level whenever a row is **inserted, updated** or **deleted** from the table. The constraints must be satisfied for the operation to be succeed.
- **Prevent** the **deletion** of a table if there are **dependencies** from other tables.
- Provide **rules** fro Oracle tools such as Oracle Developer.

Types of Constraints

Constraint	Abbr.	Description
PRIMARY KEY	_pk	•Determine which column(s) uniquely identifies each record. •It can not be NULL. •Data values must be unique.
FOREIGN KEY	_fk	•In a one-to-many relationship, it is added to the ‘many’ table. •The constraint ensures that if a value is inserted into a specified column, it must already exist in the ‘one’ table, or the record is not added.
UNIQUE	_uk	•Ensures that all data values stored in a specific column are unique. •It differs from the PK in that it allows NULL values.
CHECK	_ck	•Ensures that a specified condition is true before the data value is added to the table.
NOT NULL	_nn	•Ensures that a specified column can not contain any NULL value. •It can only be created in the column level approach to table creation.

Ways of applying Constraints

- As part of a **CREATE TABLE** command

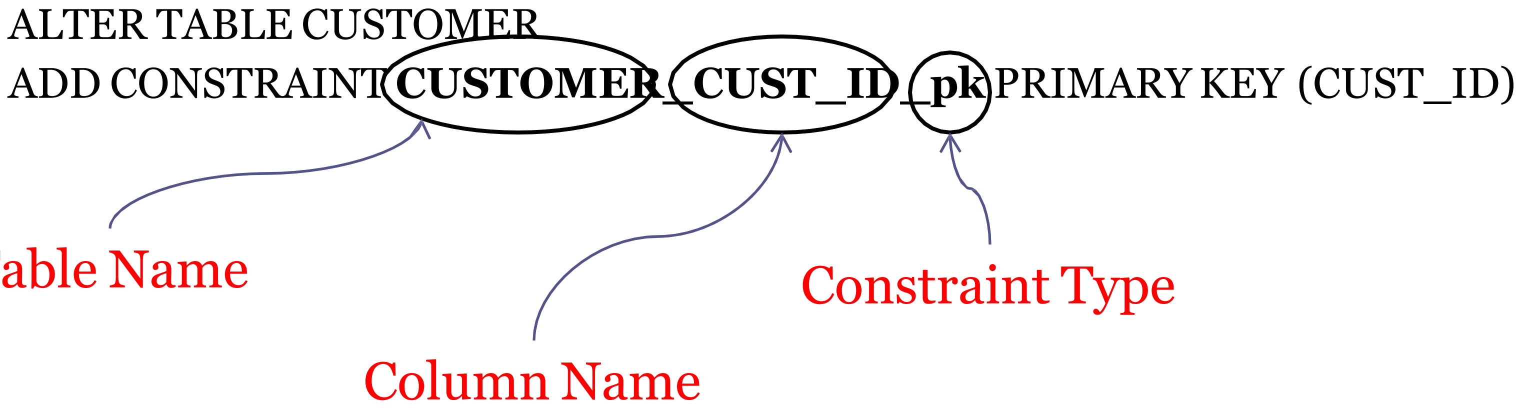
or

- As part of an **ALTER TABLE** command

Syntax for entering a constraint name

TableName_ColumName_ConstraintType

Apply the Primary Key constraint on the CUST_ID column of Customer table.



```
ALTER TABLE CUSTOMER
ADD CONSTRAINT CUSTOMER_CUST_ID_pk PRIMARY KEY (CUST_ID)
```

Table Name

Constraint Type

Column Name

PRIMARY KEY

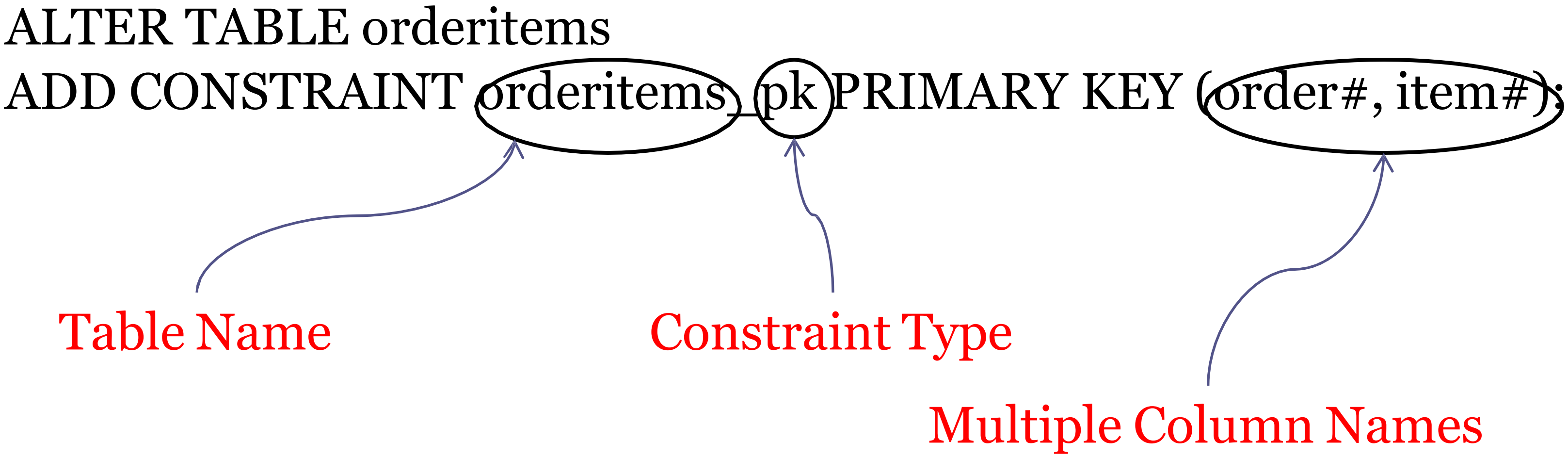
```
Create table Customer
(
    Cust_id VARCHAR2(12) PRIMARY KEY,
    Cust_nam VARCHAR2(12),
    Cust_dob DATE, Cust_street
    VARCHAR2(12), Cust_city
    VARCHAR2(12),
);
```

```
Create table Customer
(
    Cust_id VARCHAR2(12) ,
    Cust_nam VARCHAR2(12),
    Cust_dob DATE, Cust_street
    VARCHAR2(12), Cust_city
    VARCHAR2(12),
    CONSTRAINT Customer_CUST_ID_pk PRIMARY KEY(CUST_ID)
);
```

```
ALTER TABLE CUSTOMER
ADD CONSTRAINT Customer_CUST_ID_pk PRIMARY  KEY(CUST_ID);
```

PRIMARY KEY - COMPOSITE

- Simply list the column names within parentheses after the constraint type.



```
ALTER TABLE orderitems
ADD CONSTRAINT orderitems_pk PRIMARY KEY (order#, item#);
```

Table Name

Constraint Type

Multiple Column Names

- After this constraint is added to the ORDERITEMS table, a user can enter only a unique combination of Order# and Item# for each new row.

FOREIGN KEY

```
ALTER TABLE Depositor
ADD CONSTRAINT Depositor_Cust_ID_fk FOREIGN KEY (Cust_ID)
    REFERENCES Customer (Cust_ID);
```

- A record cannot be deleted in the parent table (CUSTOMER) if matching entries exist in the child table.

- That is, you cannot delete a customer form the CUSTOMERS table if there are Account in the DEPOSITOR table that Customer.

- But what if you really want to remove a customer (from the CUSTOMERS table) that does have related Account (in the DEPOSITOR table).

FOREIGN KEY

- **The conventional method is to**
  - First, remove the related records from the child table (DEPOSITOR)
  - Then, remove the customer record form the CUSTOMER table.

- **A simpler method is available:**

```
ALTER TABLE DEPOSITOR
ADD CONSTRAINT DEPOSITOR _CUST_ID_fk
FOREIGN KEY (CUST_ID) REFERENCES CUSTOMER (CUST_ID) ON
   DELETE CASCADE;
```

- **If a record is deleted from the parent table, then any corresponding records in the child table are also automatically deleted.**
  - To try the above, you have to first remove the original FOREIGN KEY constraint:

```
ALTER TABLE DEPOSITOR
DROP CONSTRAINT DEPOSITOR _CUST_ID_fk;
```

FOREIGN KEY - Composite

```
CREATE TABLE Depositor
(
   Cust_id VARCHAR2(12) NOT NULL,
   Account_id VARCHAR2(12) NOT NULL,
   COSNTRAINT DEPOSITOR_CUST_ID_FK FOREIGN
   KEY(CUST_ID) REFERENCES CUSTOMER(CUST_ID),
   COSNTRAINT DEPOSITOR_ACCOUNT_ID_FK FOREIGN
   KEY(ACCOUNT_ID) REFERENCES ACCOUNT(ACCOUNT_ID)
);
```

CHECK

```
Create table Account
(
Account_id VARCHAR2(12) NOT NULL UNIQUE,
Balance NUMBER(20,5) CHECK( Balance>0),
Type VARCHAR2(8)
);
```

```
Create table Account
(
Account_id VARCHAR2(12) NOT NULL UNIQUE,
Balance NUMBER(20,5) CHECK( Balance>0),
Type VARCHAR2(8)
CONSTRAINT Account_Balance_ck  CHECK(Balance>0)
);
```

UNIQUE

```
Create table Account
(
Account_id VARCHAR2(12) NOT NULL ,
Balance NUMBER(20,5),
Type VARCHAR2(8),
CONSTRAINT Account_ACCID_uk UNIQUE(Account_id)
);


ALTER TABLE ACCOUNT
ADD CONSTRAINT ACCOUNT_ACCOUNT_ID_uk
   UNIQUE(ACCOUNT_ID);
```

UNIQUE

- A **UNIQUE** constraint allows **NULL** values unless define **NOT NULL** in the same column

- A **PRIMARY KEY** constraint does not allow **NULL** values

NOT NULL

```
Create table Customer
(
  Cust_id   VARCHAR2(12)  NOT NULL,
  Cust_name       VARCHAR2(12),
  Cust_dob        DATE,
  Cust_street     VARCHAR2(12),
  Cust_city       VARCHAR2(12)
);

ALTER TABLE CUSTOMER
MODIFY (CUST_ID NOT NULL);
```

ADD Constraints

- You can add, drop, enable or disable a constraint, but you cannot modify its structure.

- You can add a NOT NULL constraint to an existing column by using the MODIFY Clause of the ALTER TABLE statement.

DROP Constraints

- To drop a constraint, you can identify the constraint name from the USER_CONSTRAINTS and then use ALTER TABLE command with the DROP clause.
- To remove the primary key constraint from the Customer Table and drop the associated FOREIGN KEY constraint-

```
ALTER TABLE CUSTOMER
DROP PRIMARY KEY CASCADE;
```

Viewing constraints

- Query the USER_CONSTRAINTS table to view all the constraint definition and names.

```
SELECT CONSTRAINT_NAME, CONSTRAINT_TYPE,
    SEARCH_CONDITION
FROM USER_CONSTRAINTS
WHERE TABLE_NAME='CUSTOMER';
```

- *Viewing The Columns Associated With Constraints*

```
SELECT CONSTRAINT_NAME,COLUMN_NAME
FROM USER_CONS_COLUMNS
WHERE TABLE_NAME='CUSTOMER';
```

Practice Problems for Constraints

- **Add a FOREIGN KEY CONSTRAINT on the EMPLOYEE table that ensures that each Employee's Manager also exists in Employee Table.**

- **CREATE TABLE BORROWER in such a way that Cust_ID must be in Customer table and Loan_ID must be in LOAN table.**

FUNCTIONS

CASE Based Functions

CASE CONVERSION FUNCTIONS

- To convert letters to lower or upper case
- Most database administrators rarely need to use character functions
- Application developers frequently include them to create user-friendly database interfaces
- In Oracle, the comparisons of data are case-sensitive.

CASE CONVERSION FUNCTIONS

```
SELECT branch_name
FROM branch
WHERE branch_city = “HORSENECK”;
```

- Executing it No rows will be returned.
    - Why?
        - The branch city we’re looking for is stored in as “Horseneck”. But the search key has been entered in upper case as “HORSENECK”.

CASE CONVERSION FUNCTIONS

- 2 Functions
  - **LOWER** – Converts character strings to lower-case
  - **UPPER** - Converts character strings to upper-case

LOWER

```
SELECT branch_name
FROM branch
WHERE lower(branch_city) = “horseneck”;
```

- The LOWER function temporarily converts the branch_city values to lower case.
- Thus, the “Horseneck” branch_city is converted to “horseneck”, which matches the search key “horseneck”

UPPER

```
SELECT branch_name
FROM branch
WHERE upper(branch_city) = “HORSENECK”;
```

UPPER

```
SELECT
UPPER(branch_name)
FROM branch
WHERE upper(branch_city) = “HORSENECK”;
```

UPPER(BRANCH_NAME)
PERRYRIDGE
MIANUS
ROUND HILL
POWNAL

INITCAP

- To convert character strings to mixed case, with each word beginning with a capital letter.

```
SELECT Branch_name "BRANCH NAME
AS IN DATABASE", INITCAP
(branch_name) "BRANCH NAME INIT
CAP EXAMPLE"
FROM BRANCH
```

INITCAP

BRANCH NAME AS IN DATABASE	**BRANCH NAME INIT CAP EXAMPLE**
Downtown	Downtown
Redwood	Redwood
Perryridge	Perryridge
Mianus	Mianus
Round **h**ill	**R**ound **H**ill
Pownal	Pownal
north **t**own	**N**orth **T**own
brighton	**B**righton

CHARACTER MANIPULATION FUNCTIONS

SUBSTR

- Used to return a substring, or portion of a string

  SUBSTR(character string, beginning character position, length of string to be returned)

- SELECT branch_name, **SUBSTR(branch_name,1,3)** FROM branch;

BRANCH_NAME	SUBSTR(BR
Downtown	**Dow**
Redwood	**Red**
Perryridge	**Per**
Mianus	**Mia**
Brighton	**Bri**

SUBSTR

- SELECT branch_name, **SUBSTR(branch_name,4,2)** FROM branch;

BRANCH_NAME	SUBSTR
Downtown	
Redwood	
Perryridge	
Mianus	
Brighton	

SUBSTR

- SELECT branch_name, **SUBSTR(branch_name,4,2)** FROM branch;

BRANCH_NAME	SUBSTR
Downtown	nt
Redwood	wo
Perryridge	ry
Mianus	nu
Brighton	gh

LENGTH

```
LENGTH(character string)
```

```
SELECT branch_name, LENGTH(branch_name)
FROM branch;
```

BRANCH_NAME	LENGTH(BRANCH_NAME)
Downtown	8
Redwood	7
Perryridge	10
Mianus	6

LPAD

LPAD(string to be padded, length of string after padding, symbol used to pad)

```
SELECT branch_name, LPAD(branch_name,12,'*')
FROM branch;
```

BRANCH_NAME	LPAD(BRANCH_NAME,12,'*')
Downtown	****Downtown
Mianus	******Mianus
Round Hill	**Round Hill
Pownal	******Pownal

RPAD

RPAD(string to be padded, length of string after padding, symbol used to pad)

SELECT branch_name, **RPAD(branch_name,12,'*')**
FROM branch;

BRANCH_NAME	RPAD(BRANCH_NAME,12,'*')
Downtown	Downtown****
Perryridge	Perryridge**
Mianus	Mianus******
Round Hill	Round Hill**
Pownal	Pownal******

LTRIM

- Removes a specific string of characters from the left side of the data

  LTRIM(data, specific string to be removed from the left side of the data)

```
SELECT cust_id, LTRIM(cust_id,'C')
FROM customer;
```

LTRIM

CUST_ID	LTRIM(CUST_ID,'C')
C00000000001	00000000001
C00000000002	00000000002
C00000000003	00000000003
C00000000004	00000000004
C00000000005	00000000005
C00000000006	00000000006

RTRIM

- Removes a specific string of characters from the right side of the data

  RTRIM(data, specific string to be removed from the right side of the data)

```
SELECT RTRIM ('***Sample***', '*')
from customer;
```

RTRIM

'***SAMPLE***'	RTRIM('***SAMPLE***', '*')
Sample	***Sample
Sample	***Sample
Sample	***Sample
Sample	***Sample
Sample	***Sample

REPLACE

- Similar to "search and replace" in some application programs

REPLACE(column, string to be found, string replacement)

```
SELECT cust_id, REPLACE(cust_id, 'C000', 'Cust')
FROM customer;
```

REPLACE

CUST_ID	REPLACE(CUST_ID,'C000','CUST')
C00000000001	Cust00000001
C00000000002	Cust00000002
C00000000003	Cust00000003
C00000000004	Cust00000004
C00000000005	Cust00000005

CONCAT

- Concatenates the data from two columns

- Combines only two items (columns or string literals)

CONCAT(column or string, column or string)

SELECT cust_name, CONCAT('Customer Number: ', cust_id) "Number" FROM customer;

- To concatenate more than two items, you must nest a CONCAT function inside another CONCAT function

TRY YOURSELF

CONCAT

```
SELECT cust_name, CONCAT('Customer Number: ', cust_id) "Customer ID"  FROM customer;
```

CUST_NAME	Customer ID
Jones	Customer Number: C00000000001
Smith	Customer Number: C00000000002
Hayes	Customer Number: C00000000003
Curry	Customer Number: C00000000004
Lindsay	Customer Number: C00000000005

NUMERIC FUNCTIONS

ROUND

- To round numeric fields to the stated precision
  - If position is a positive number, it refers to the right side of the decimal point.
  - If position is a negative number, function rounds to the left side of the decimal point.

ROUND(numeric field to be rounded, position of the digit to which the data should be rounded)

ROUND

```
SELECT ROUND(3162.845, 1) AS ROUNDED
FROM dual;
```

- 3162.8

```
SELECT ROUND(3162.8451297, 5) AS ROUNDED
FROM dual;
```

- 3162.84513

ROUND

SELECT **ROUND(3162.845, -2)** AS ROUNDED FROM dual;

- 3200

SELECT **ROUND(1234,-2)** AS ROUNDED FROM dual;

- 1200

SELECT **ROUND(5232.85, -3)** from dual

- 5000

TRUNC

- To truncate a numeric value to a specific position
  - If position is a positive number, it refers to the right side of the decimal point.
  - If position is a negative number, function rounds to the left side of the decimal point.

TRUNC (numeric field to be rounded, position of the digit from which the data should be removed)

TRUNC

```
SELECT TRUNC(15.79,1) "Truncate"
FROM DUAL;
```

- 15.7

```
SELECT TRUNC(123456.76,-4) "Truncate"
FROM DUAL;
```

- 120000

DATE FUNCTIONS

Difference between Two dates in Days

```
SELECT Emp_id, Emp_dob,  Emp_startdate,
   Emp_startdate – Emp_dob
FROM Employee;
```

E_ID	E_DOB	E_STARTDATE	E_S_DATE - E_DOB
E000002	01/22/1958	01/22/1978	7305
E000010	04/21/1956	04/21/1986	10957

Difference between Two dates (Weeks)

- **The delay between the two dates in weeks:**

```
SELECT Employee_id, Employee_dob,
Employee_startdate,
(Employee_startdate – Employee_dob)/7
"DELAY IN WEEKS"
FROM Employee;
```

Difference between Two dates

EMP_ID	EMP_DOB	EMP_S_DATE	DELAY IN WEEKS
E000002	01/22/1958	01/22/1978	1043.57142
E000010	04/21/1956	04/21/1986	1565.285714

MONTHS_BETWEEN

- Determines the number of months between two dates

MONTHS_BETWEEN(later date, earlier date)

```
 SELECT Employee_id,
MONTHS_BETWEEN (Employee_startdate,
Employee_dob) “Delay in Months”
FROM Employee;
```

MONTHS_BETWEEN

EMPLOYEE_ID	Delay in Months
E00000000002	240
E00000000003	360

ADD_MONTHS

ADD_MONTHS(beginning date, number of months to add to the date)

```
SELECT Employee_id,
Employee_startdate, ADD_MONTHS
(Employee_startdate, 60)
FROM Employee;
```

ADD_MONTHS

EMP_ID	EMP_STARTDATE	ADD_MONTHS(EMP_STARTDATE,60)
E0000002	01/22/1978	01/22/1983
E0000003	02/23/1982	02/23/1987

NEXT_DAY

- Determines the next occurrence of a specific day of the week after a given date – Output is a DATE

NEXT_DAY(starting date, day of week to be identified)

```
SELECT Employee_id,
  NEXT_DAY(Employee_startdate, 'MONDAY')
  “First Monday After Joining”
FROM Employee;
```

The Nesting of Functions

- A function is used as an argument inside another function
- Rules
  - One must include all arguments for each function.
  - For every open parenthesis, there must be a corresponding closed parenthesis.
  - The inner function is resolved first, then the outer function.

The Nesting of Functions

To determine the Number of months between the Employee_startdate and Employee_dob, we use the MONTHS_BETWEEN function.

```
SELECT Employee_id, MONTHS_BETWEEN (Employee_startdate, Employee_dob) “Delay in Months” FROM Employee;
```

The Nesting of Functions

To suppress the decimal places generated by the Months_Between function, we can use the result of the Months_Between function as an input to the TRUNC function.

```
SELECT Employee_id,
TRUNC(MONTHS_BETWEEN
(Employee_startdate, Employee_dob),0) “Delay
in Months”  FROM Employee;
```

THANK YOU