Wildcard (*)

- used to select all available columns from a table.
- This effect applies to all tables the query accesses through JOIN clauses.

- To select all values from a specific table, the wildcard character can be applied to table with dot notation

```
SELECT
    Employees.*, Department.Name
FROM
    Employees JOIN Department
ON
    Department.ID = Employees.DeptID
```

Warnings against use:-

- Use of * is avoided in production code cause it can cause following problems.

* 1. Excess IO, network load, memory use and so on
* 2. further excess IO if DB needs to spool internal results to disk as a part of processing query
* 3. Potential ambiguity in column names if columns are added to tables and views later on.

When to use-

- to perform manual queries against db for investigation and prototype work.

Alias-

```
Select  fname as "FirstName"        -> all versions of SQL.
Select  fname as "Firstname"
        Mname as 'middle'           -> mysql
        Lname as [Last Name]           ("", '', [])
    from Employees
```

If alias has a single word that is not a reserved word we can write it without single quote, doublequote or []

eg

```
Select fname as firstName from employees;

Select FullName = firstName + ' ' + lastName from emp -> mysql

Select firstName + ' ' + lastName AS fullName ->
```

'As' - standard rather than '='

If you need to use reserved words as alias you can use brackets or quote to escape

```
select fname as "SELECT"
       MNAME as "FROM"
from emp
```

→ all versions

```
Select fName as "SELECT"
       LName as 'FROM'
       LNAME as [WHERE]
```

→ mySql.

Also, column names may be used any of final clauses of same query

eg.

```
SELECT
  fname as firstname,
FROM
  emp
ORDER BY
  firstName DESC.
```

Select individual columns

```
Selcet name, id, contact from emp
```

* Columns are returned in order in which they appear in SELECT clause

Select specific number of records

SQL 2008 standard -

```
Select id, name from Product
order by price desc
  OFFSET 5 ROWS
  FETCH FIRST 10 ROWS ONLY
```

→ skip 5 rows

→ limit to 10 rows in op

SQL server & MS Access

```
Select TOP 10 id, name from Products
```

→ TOP operates after where clause

MySQL, Postgres

```
Select id, name from Product
  limit 10.
```

Oracle (ROWNUM)

```
Select id, name from xyz
```

where ROWNUM <=10
Ordery by price desc

→ ROWNUM works as a part of where clause so if other condn do not exist in specified no. of rows you will get 0 results.

Select with condition
select name where id > 10

Select with CASE.
When results need some logic applied on the fly one can use CASE statement to implement it.

SELECT CASE WHEN col1 < 50 THEN 'under' ELSE 'over'
END threshold.

Also can be chained
SELECT
CASE WHEN col1 < 50 THEN 'under'
WHEN col1 > 50 AND col1 < 100 THEN 'between'
ELSE "over'
END threshold
FROM tablename

You can also have case inside case.

Select columns which are named after reserved keywords
SELECT
"ORDER",
ID
FROM orders.

Note - This makes column name case sensitive.

SELECT [order], ID from orders. → SQL Server

SELECT 'Order' id FROM orders → mysql, MariaDB

Selecting with table alias.

SELECT e.fname, e.lname FROM Employees e

* Note one you use alias, you cannot use canonical table name anymore ie
Select e.fname, Employee.id from employee e
will throw an error.

Selecting with more than 1 condition

```
Select name From persons where gender = 'M' AND age > 20;
                                      OR  age > 20;
```

Can be combined

```
SELECT name from Persons WHERE (gender = 'M' AND age < 20)
                            OR
                           (gender = 'F' AND age > 20);
```

Selecting without locking the table

Sometimes when tables are used mostly (or only) for reads, indexing does not help anymore and every little bit counts, one might use selects without LOCK to improve performance

SQL Server

```
SELECT * FROM TableName WITH (nolock)
```

MySQL

```
SET SESSION TRANSACTION ISOLATION LEVEL READ UNCOMITTED;
Select * from tablename
```

```
SET SESSION TRANSACTION ISOLATION LEVEL REPEATABLE READ;
```

Oracle

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMITTED;
SELECT * FROM TableName;
```

DB2

```
SELECT * FROM TABLENAME WITH UR.
                            Uncomitted read.
```

Selecting with Aggregate functions

```
SELECT AVG(Salary) FROM Employees. where DepId = 1;
```

Can be combined with where and group By clause too

MIN()

MAX()

AVG()

Count(), Count(*) (null values are not counted)

SUM(Salary)

Select with condition of multiple values from column.

SELECT * FROM cars WHERE status IN ('wait', 'work')

=

Select * from cars where status = 'wait' OR status = 'work'

→ NOT IN ( )

DISTINCT keyword.

Select Count (DISTINCT DeptId) FROM Emp

Get Aggregated Results for Row Group.

Counting rows based on a specific column value

```
SELECT category, COUNT(*) AS item_count
 FROM item
 GROUP BY category
```

Getting average income by department

```
SELECT dept, AVG(income)
 FROM employees
 GROUP BY dept
```

* The important thing is to select only columns specified in the GROUP BY clause or used with aggregate functions.

WHERE clause can be used with GROUP BY, but WHERE filters out records before any grouping is done

```
Select dept, Avg(income)
 FROM employees
Where dept <> 'ACCOUNTING'
Group by dept ;
```

If you need to filter result after grouping use HAVING.

```
Select dept, avg(income)
 FROM employees
 Group by dept
 Having avg(income) > 1000 ;
```

Selection with sorted Results.

Select * from emp order by lname, fname DESC.

↳ first sort by lname (ASC) then by fname (DESC)

default - ASC.

- Select id, name, number from Employees order by 3;

↳ column number

Select id, fname, name, number from Employees order by fname

ORDER BY CASE WHEN lname = 'Jonas' then 0 ELSE 1 ASC

↓

This will sort to have all records with lname Jones at top

Selecting with null

Select Name FROM customers where number IS NULL

IS NOT NULL

Select distinct (unique values only).

Select DISTINCT · continentcode FROM countries;

Select from multiple tables.

Select * FROM table1, table2

cross product

Select table1.col1, table1:col2 FROM table1, table2

GROUP BY

Result of a select query can be grouped by one or more columns using GROUP BY statement: all results with the same value in the grouped columns are aggregated together.

- Group by can be used in conjunction with aggregation functions using HAVING statement to define how non grouped columns are aggregated

```
Select empId, Sum(MonthlySalary)
  FROM Employee
  GROUP BY EmpId
```

filter Group By using HAVING clause.

```
Select a.id, b.name, count(*) BooksWritten
 from BookAuthors ba
      INNER JOIN                  → Returns all authors who
        Authors a                   wrote more than one book.
  ON a.id = ba.authorId
  GROUP BY
    aid, a.Name
  HAVING (count(*) > 1.          → having BooksWritten > 1
```

Use Group by to count number of rows for each unique entry in given column.

eg Name House

Arya Stark

Sansa	Stark
Cersei	Lannister
Yara	Greyjoy

```
select house, count(*) number_of_westerosis
  from westerosians
  Group by House
  order by number_of_westerosians DESC
```

o/p

Stark	2
Lannister	1
Greyjoy	1

★ Read: CASE from the doc

Like Operator

```
select * from emp where fname LIKE '%on%'
```

- '%on%' → any no. of chars before & after on.
- LIKE '245%' → start with 245.
- LIKE '%246' → end with 246
- LIKE '__n%' → 2 _ → skip first 2 chars before n.
- LIKE 'a_n' → single char match aon, adn, etc
- LIKE '_A_T' → _SAET_, WATI, MATI

Escape statement in LIKE query

```
Select *
From tablename
 where name like CONCAT('%', @in_searchText, '%') ESCAPE '\'
```

⇒ prepend '\' to every char in seach text so that results are correct

eg - in_searchText	o/p without ESCAPE	with ESCAPE
a_b	X	✓
a%bcd_a	X	✓

Search for range of chars.

```
select * from emp where fname like '[A-P]%'
```

⇒ fname that starts with A to P.

Match by range or set

- LIKE '[a-a]an' → gan but not man

- LIKE '[lmnop]any' → many but not gany
- LIKE '[^a-g]gany' → ^ → negation. many but not geny.
- LIKE '[^lmnop]any' → gany but not many

Wildcard Characters

SQL wildcards are —

1) % → zero or more chars.
2) _ → single char
3) [charlist] → set and range of chars to match [a-g], [abc]
4) [^charlist] → matches only a char not specified in brackets

Filter results using WHERE and HAVING.

BETWEEN → is inclusive.

select * from sales where qty BETWEEN 10 and 17.

qty ≥ 10 && qty ≤ 17

→ date BETWEEN '2013-07-11' AND '2013-05-24'

→ lname BETWEEN 'D' AND 'L' → lname alphabetically falls b/w D & L.

→ 'A, B, C, M → not included

Use HAVING with AGGREGATE functions.

Unlike WHERE clause, HAVING can be used with aggregate functions.

Where EXISTS.

Select * from tablename t WHERE EXISTS
(Select 1 from tablename1 t1 where t.id = t1.id)

Select records in tablename that have records matching in tablename1.

OFFSET & LIMIT

Select * from tablename LIMIT 20, 20. -- offset, limit.
(mysql)

EXCEPT

select dataset except where values are in other dataset
--dataset schemas must be identical

```
SELECT 'Data1' as 'Column' UNION ALL
SELECT 'Data2' as 'Column'
EXCEPT
SELECT 'Data3' as 'Column'
```

Returns data1, [redacted].

Explain and Describe

Explain → shows you how the query will be executed This shows if query uses an index or if you could optimize your query by adding an index.

Chapter 16: EXPLAIN and DESCRIBE

Section 16.1: EXPLAIN Select query

An `Explain` infront of a `select` query shows you how the query will be executed. This way you to see if the query uses an index or if you could optimize your query by adding an index.

Example query:

```
explain select * from user join data on user.test = data.fk_user;
```

Example result:

```
id  select_type  table  type   possible_keys  key      key_len ref        rows  Extra
1   SIMPLE       user   index  test           test     5       (null)     1     Using where; Using
index
1   SIMPLE       data   ref    fk_user        fk_user  5       user.test  1     (null)
```

on `type` you see if an index was used. In the column `possible_keys` you see if the execution plan can choose from different indexes of if none exists. `key` tells you the acutal used index. `key_len` shows you the size in bytes for one index item. The lower this value is the more index items fit into the same memory size an they can be faster processed. `rows` shows you the expected number of rows the query needs to scan, the lower the better.

Section 16.2: DESCRIBE tablename;

`DESCRIBE` and `EXPLAIN` are synonyms. `DESCRIBE` on a tablename returns the definition of the columns.

```
DESCRIBE tablename;
```

Exmple Result:

```
COLUMN_NAME  COLUMN_TYPE   IS_NULLABLE  COLUMN_KEY  COLUMN_DEFAULT  EXTRA
id           int(11)       NO           PRI         0               auto_increment
test         varchar(255)  YES                      (null)
```

Here you see the column names, followed by the columns type. It shows if `null` is allowed in the column and if the column uses an Index. the default value is also displayed and if the table contains any special behavior like an `auto_increment`.

MacBook Air

Describe tablename ;
→ returns definition of columns.

EXISTS | NOT EXISTS.
select * from table1 where not exists (
select * from table2 where table1.id = table2.id)

EXIST, IN, JOIN could be sometimes used for same result, however they are not equals:
- EXISTS should be used to check if a value exists in a table
- IN should be used for static list.
- JOIN should be used to retrieve data from other(s) table(s)

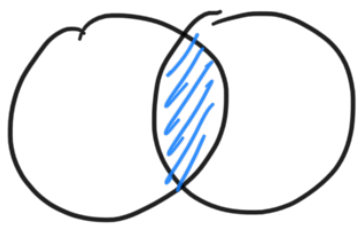
JOIN

JOIN is a method of combining (joining) information from two tables
- A table may be joined to itself
- If information from more than two table needs to be accessessed, multiple JOINS can be specified in a FROM clause.

Self Join

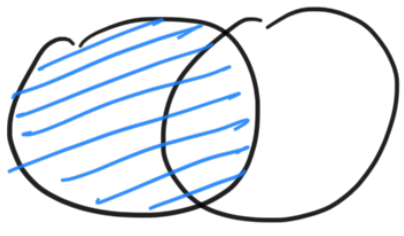
SELECT e.fname, m.fname from Employee e JOIN Manager m
ON e.ManagerId = m.Id

INNERJOIN



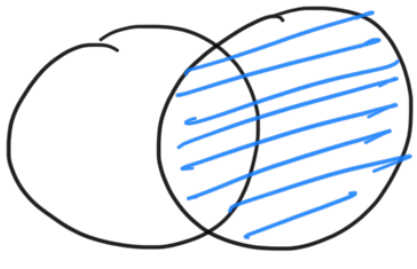
Select <fields> from tableA A
INNER JOIN
tableB B
ON A.key = B.key

LEFT JOIN. || LEFT OUTER JOIN (If no matching rows in right, set to NULL)



select <fields> from tableA A
LERT JOIN
TableB B
ON A.key = B.key

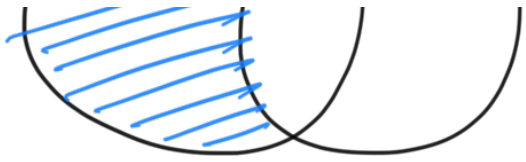
RIGHT JOIN | RIGHT OUTER JOIN (If no match in left set to null)



select <fields> from table A
RIGHT JOIN
tableB B
ON A.key = B.key

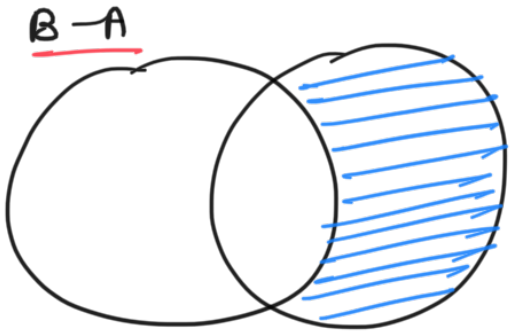
A - B

select <fields> from table A



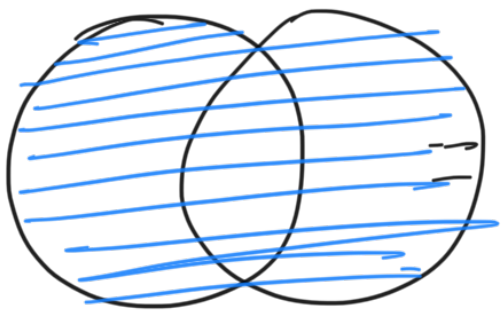
JOIN
tableB B on A.key = B.key
where B.key IS NULL.

B − A

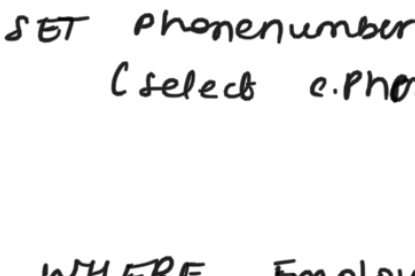


A.key IS NULL

FULL OUTER JOIN



Select <fields> from tableA A
FULL OUTER JOIN
tableB B on
A.key = B.key.



Select <fields> from tableA A
FULL OUTER JOIN
tableB B
ON A.key = B.key
where A.key IS NULL OR B.key IS NULL

Implicit Join → (not advised.)
join using from, where clauses on tables
-Does not contain JOIN command.

CROSS JOIN (A x B.) → Cartesian Product.
num in A x num in B → table size

UPDATE

Update with data from other table

Q) Set Phonenum. for any employee who is also a customer and currently does not have a phone number set.

```
UPDATE
    Employees
SET Phonenumber =
    (select c.Phonenumber FROM customers c WHERE
                c.fname = Employees.fname
                AND c.Lname = Employees.Lname )
WHERE Employees.Phonenumber is NULL.
```

Using Inner Join

```
UPDATE Employees
SET phonenumber = c.Phonenumber
FROM Employees e.
    INNER JOIN Customer c
        ON e.fname = c.fname AND e.lname = c.lname
WHERE
    Phonenumber is null
```

Modifying existing values.

```
Update Cars
set totalcost = Totalcost+100
where Id=3 or Id=4
```

Update specified rows

```
Update Cars
set status = 'READY'
WHERE Id>4
```

Update all rows

```
Update Cars
set status = 'READY'
```

Capturing updated records.

Sometimes one wants to capture the records that have just been updated

```
CREATE TABLE #TempUpdated (ID INT)

Update table1 set col = 42
    OUTPUT inserted.ID INTO #TempUpdated
        WHERE ID >50 -
```

CREATE DATABASE

Create database database_name ⟶ creates empty db

Create Table

Create table from select

```
Create table clone AS select * from employees.

Create table clone AS
```

```
SELECT id, CONCAT( fname, ' ', lname) AS FullName
FROM Employees
WHERE ID > 10
```

Create New Table.

```
Create table tablename (
    ID int identity(1,1) primary key not null,
    fname varchar(10) not null,
    lname varchar(10 not null,
    number varchar(10),
    deptId int FOREIGN KEY REFERENCES Dept(Id)
);
```

identity(1,1) ⇒ col will have auto generated values starting at 1 and incrementing by 1 for each new row

Duplicate a table

```
CREATE TABLE newtable LIKE oldtable;
INSERT INTO newtable Select * from oldtable;
```

Creating a temporary or a in memory table

Create temp TABLE MyTable(...) → SqLite or Postgres.

SQL server.

imp.

Create table #TempPhysical(...); → create a temp table local to the session

Create table ##TempphysicalVisibletoEveryone(....) → visible to everyone.

DECLARE @TempMemory TABLE(...); → create an in memory table

CREATE FUNCTION

Argument	Description
function_name	name of fn.
list_of_parameters	params
return_data_type	return type
function_body	code
Scalar_expression	Scalar value returned by function

```
CREATE FUNCTION FirstWord (@input varchar(1000))
RETURNS varchar(1000)
AS
BEGIN
    DECLARE @output varchar(1000)
    SET @output = SUBSTRING(@input, 0, CASE CHARINDEX
```

(' ',@input)

```
        WHEN 0 THEN LEN(@input)+1.
         ELSE CHARINDEX(' ', @input)
         END)
       RETURN @Output
END
```

TRY/CATCH

Transaction in a TRY/CATCH
This will rollback both inserts due to Invalid Datetime.

```
BEGIN TRANSACTION
BEGIN TRY
        INSERT INTO dbo.sale VALUES (5-2,GETDATE(),1)
        ------------------------------------------------
        COMMIT TRANSACTION
END TRY
BEGIN CATCH
     THROW
      ROLLBACK TRANSACTION.
END CATCH.
```

(---, 'notadate', 2)
'invalid.

UNION / UNION ALL

- used to combine two SELECT statement results without any duplicate.
- In order to use UNION both select statement should have same no. of columns with same datatype in same order, but the length of column can be different.

```
Select fname, lname
 FROM HR_EMPLOYEES.
 WHERE pos = 'Mgr'
  UNION ALL
 SELECT fname, lname
FROM FW-EMPLOYEE
 WHERE pos = 'Mgr'
```

→ 2ppl in diff dept with same fname, lname keep both.

UNION → removes duplicates
UNION ALL → does not remove duplicate

ALTER TABLE

- modify column/constraint in a table

Add Column(s)

```
ALTER TABLE Employees
```

ADD StartDate Date NOT NULL DEFAULT GETDATE(),
DOB3 date NULL

Drop column
ALTER TABLE emp
DROP COLUMN columnname;

ADD Primary Key
ALTER TABLE emp ADD pk-EmpID PRIMARY KEY (ID)
(ID, fname)

ALTER COLUMN a
ALTER TABLE EMPLOYEES
ALTER COLUMN StartDate DATETIME NOT NULL DEFAULT (GETDATE())

DROP CONSTRAINT
ALTER TABLE Emp
DROP CONSTRAINT DefaultSalary

Note: Ensure constraints of a column are dropped before dropping a column

INSERT
from another table
INSERT INTO Table1 Select * from table2;

INSERT NEW ROW
INSERT INTO CUSTOMERS VALUES (1, 'Zaek', '221105')

Insert only specified columns
Insert into customer (fname, contact)
VALUES ('Smith', '21034')

Insert multiple rows.
Insert into tbl_name(field1, field2, field3)
VALUES (1,2,3), (4,5,6), (7,8,9);

MERGE
=) UPSERT => Update + Insert.

CROSS APPLY, OUTER APPLY

SELECT *
FROM Department D
CROSS APPLY (
SELECT *

→ Selects data from Department table & uses CROSS APPLY

```
    FROM EMPLOYEE E
    WHERE E.DepartmentID = D.DeptId
  ) A
GO
SELECT *
FROM DEPARTMENT D
INNER JOIN Employee E
  ON E.DepartmentId = D.DeptId.
```

→ to evaluate Employee table for each record of Dept table.

→ Join emp and dpt and all matching records are produced.

```
SELECT *
FROM Department D
OUTER APPLY (
    SELECT *
    FROM Employee E
    WHERE E.DepartmentID = D.DeptID.
  ) A
GO
SELECT *
FROM Department D
LEFT OUTER JOIN Employee E
    ON D.DepartmentID = E.DepartmentID
GO
```

→ Similar o/p but execution plan will be different.

* When APPLY is required?

eg:

```
CREATE FUNCTION dbo.fnname (@DeptId AS int)
RETURNS TABLE
AS
  RETURN (
    SELECT * FROM Employee E
    where E.DeptId = @DeptId
  )
GO
SELECT * FROM Department D
 CROSS APPLY dbo.fnname(D.DepartmentID)
 GO
 SELECT * FROM Department D
OUTER APPLY dbo.fnname(D.DepartmentID)
GO
```

Can we use join here? → NO

* If you replace CROSS ↕ innerjoin / OUTER ↕ left outer join APPLY with

You will get error -

The multiple identical DataFrames could not be found

This is because with JOIN the execution context of outer query is different from execution context of function (or a derived table) and you cannot bind a value/variable from outer query to the function as parameter.
Hence APPLY operator is used for such queries.

DELETE

Delete all rows (Truncate is better in performance as it ignores triggers and indexes and logs to just delete the data)

```
DELETE FROM Employees
```

Delete certain rows with WHERE

```
DELETE FROM Emp WHERE ID > 10;
```

TRUNCATE → delete all rows and resets values such as autoincrement. It also doesn't log individual row deletion.

```
TRUNCATE TABLE Employees;
```

Delete	Truncate
1) remove one or many (all) rows	1) removes all rows
2) row based operation ie - each row is deleted	2) Page operation Entire data page is reallocated.
3) slower	3) faster way to delete all rows.
4) does not reset auto inc.	4) Reset AutoIncrement counters
5)	- NO foreign keys must be present otherwise you will get an error.

DROP TABLE

Check for existence before dropping

```
DROP TABLE IF EXISTS myTable;.
```

Simple Drop

```
DROP TABLE tablename;
```

DROP/DELETE DATABASE

```
DROP DATABASE [dbo].[Employee]
-remove Eemployee db
```

* - Always have a backup before dropping if required further

CASCADING DELETE

```
ALTER TABLE room
ADD CONSTRAINT FK-clientID FOREIGN KEY (RM-CLI-ID)   (constraint name)
REFERENCES CLIENT(ID)
ON DELETE CASCADE
ON UPDATE CASCADE
```

GRANT / REVOKE

```
REVOKE/GRANT SELECT, UPDATE
  ON Employees
  TO User1, User2
```

PRIMARY KEYS.

```
Create table Emp(
   id INT NOT NULL,
   fname varchar(50),
   Primary Key (id, fname)
);
```

Auto Increment

MySQL

```
CREATE TABLE Employees (
   Id int NOT NULL AUTO-INCREMENT,
   PRIMARY KEY (ID)
);
```

SQL Server

```
CREATE TABLE Employees(
      Id int NOT NULL IDENTITY,
      PRIMARY KEY (Id)
     );
```

Indexes

- Data structure that contains pointers to the contents of a table arranged in a specific order, to help database optimize queries.
- eg: index of a book.
- When an index exists on a column used in query's WHERE, JOIN, ORDER BY, it can substantially improve query performance.

Create Index.

```
Create INDEX ix-car-emp ON cars (Employeeid);
```

- creates index on employeeid in cars.

index can contain more than 1 column.

CREATE INDEX xindex ON Cars (id, EmpId, Name)

* The index will be useful in queries asking to sort or select by those columns

Sorted Index

If you use an index that is sorted the way you would retrieve it, the SELECT statement would not do additional sorting when in retrieval.

create index myscore ON Scoreboard (Score DESC);

When you execute

Select * from scoreboard order by score DESC

The db would not do additional sorting since the idx is in that order.

Partial or Filtered Index:

Create index started-orders
ON orders (productid)
where order_state = 'started';

Drop, Disable, Rebuild an index.

DROP INDEX index.name ON Cars;

* Re creating an index may be slow and computationally expensive.
* As an alternative the index can be disabled

ALTER INDEX cars-emp on Cars DISABLE;

This allows the table to retain the structure, along with metadata about the index.

Critically, this retains index statistics, so it is possible to easily evaluate the change. If warranted the index can later be rebuilt instead of being recreated completely.

ALTER INDEX cars-emp ON Cars REBUILD;

Critically, this retains the index...

Critically, this retains the index statistics, so that it is possible to easily evaluate the change. If warranted, the index can then later be rebuilt, instead of being recreated completely;

```
ALTER INDEX ix_cars_employee_id ON Cars REBUILD;
```

Section 37.5: Clustered, Unique, and Sorted Indexes

Indexes can have several characteristics that can be set either at creation, or by altering existing indexes.

```
CREATE CLUSTERED INDEX ix_clust_employee_id ON Employees(EmployeeId, Email);
```

The above SQL statement creates a new clustered index on Employees. Clustered indexes are indexes that dictate the actual structure of the table; the table itself is sorted to match the structure of the index. That means there can be at most one clustered index on a table. If a clustered index already exists on the table, the above statement will

fail. (Tables with no clustered indexes are also called heaps.)

```
CREATE UNIQUE INDEX ug_customers_email ON Customers(Email);
```

This will create an unique index for the column *Email* in the table *Customers*. This index, along with speeding up queries like a normal index, will also force every email address in that column to be unique. If a row is inserted or updated with a non-unique *Email* value, the insertion or update will, by default, fail.

```
CREATE UNIQUE INDEX ix_eid_desc ON Customers(EmployeeID);
```

This creates an index on Customers which also creates a table constraint that the EmployeeID must be unique. (This will fail if the column is not currently unique - in this case, if there are employees who share an ID.)

```
CREATE INDEX ix_eid_desc ON Customers(EmployeeID Desc);
```

This creates an index that is sorted in descending order. By default, indexes (in MSSQL server, at least) are ascending, but that can be changed.

Section 37.6: Rebuild index

Over the course of time B-Tree indexes may become fragmented because of updating/deleting/inserting data. In SQLServer terminology we can have internal (index page which is half empty) and external (logical page order doesn't correspond physical order). Rebuilding index is very similar to dropping and re-creating it.

We can re-build an index with

```
ALTER INDEX index_name REBUILD;
```

By default rebuilding index is offline operation which locks the table and prevents DML against it , but many RDBMS allow online rebuilding. Also, some DB vendors offer alternatives to index rebuilding such as `REORGANIZE` (SQLServer) or `COALESCE`/`SHRINK SPACE`(Oracle).

Section 37.7: Inserting with a Unique Index

```
UPDATE Customers SET Email = 'richard0123@example.com' WHERE id = 1;
```

This will fail if an unique index is set on the *Email* column of *Customers*. However, alternate behavior can be defined for this case:

```
UPDATE Customers SET Email = 'richard0123@example.com' WHERE id = 1 ON DUPLICATE KEY;
```

GROUP BY vs DISTINCT

Group by is used in a combination with aggregation functions.
Distinct is used to list a unique combination of distinct values for the specified columns.

eg.

storename	userid	orderid
A	43	1
B	57	2
A	43	3
C	82	4
A	21	5

Select distinct storename, userid from table1;

o/p

A	43
B	57
C	82
A	21

String functions

- perform operations on string values and return either numeric or string values

Concatenate

Select 'Hello' || 'World' || '!' ; → returns HelloWorld!
not supported in SQL Server

SELECT CONCAT ('H','W') → HW.

SELECT CONCAT ('H','W','1') → HW1 (Oracle does not support concat of more than two strings)

Select 'FOO' + CAST (42 AS varchar(5)) + 'Bar';
- SQL Server < 2012

Length

- SELECT LEN('HELLO ') → 5
Does not count trailing spaces.

- SELECT DATALENGTH('HELLO ') → 6
counts trailing spaces.

Note - Datalength returns the length of underlying byte representation of string which depends on charset used to store string (ASCII, unicode etc)

eg - DATALENGTH ('Hello') = 5 (ASCII)
= 10 (Unicode)

Oracle
Select length('Hello') FROM dual → 5

Trim empty spaces.
SELECT LTRIM(' HELLO')
RTRIM (
LTRIM(RTRIM('HELLO'))
} MS SQL

mysql & Oracle
Select TRIM (' Hello ')
LTRIM
RTRIM.

Upper and lowercase

SELECT UPPER('hello') → HELLO
LOWER → hello

Split a string.

Select value from STRING_SPLIT('Hi I am..', ' ');
→ split on space.

Replace
REPLACE (String to search, String to search for & replace,
String to replace with)

REPLACE ('Peter Steve Tom', 'Steve', 'Billy').
Olp Peter Billy Tom.

REGEXP
Select 'bedded' REGEXP '[a-f]' → returns True
Select 'beam' REGEXP '[a-f]' → returns False

Substring.
SUBSTRING (String_expression, start, length)
SQL strings are 1 indexed
SELECT SUBSTRING ('Hello', 1, 2) → 'He'
(, 3, 3) → 'llo'

LEFT, RIGHT
SELECT LEFT ('Hello', 2) → He
Select RIGHT ('Hello', 2) → lo

REVERSE

select REVERSE ('Hello') → olleH

REPLICATE

select REPLICATE ('Hello', 2) → HelloHello

Replace fn in sql select and update query.

REPLACE (str, find, repl)

select FirstName, REPLACE (Address, 'South', 'Southern')
Address

from emp

We can also use it with update

INSTR

Return first occurence of substring, return 0 if not found.

select INSTR ('FOOBarbar', 'Bar') → 4

→ 4 , 'Xar') → 0

PARSENAME

returns specific part of given string (object name).

PARSENAME ('obj.id.sever.name', 2)

o/p — sever.

Functions (Aggregate)

conditional Aggregation -

select customer,

sum(case when payment-type = 'credit' then amount else 0 end)
as credit

sum(case when pay-type = 'debit' then amount else 0 end)
as debit

from payments

group by customer

* List concatenation (later)

Aggregate fn's

sum(), min(), max(), avg(), count()

Functions (Scalar/single row)

Scalar fn → take one value as input and returns one value as output for each row in result set.

Date Time -

Datatype	format
time	hh:mm:ss [.nnnnnnn]
date	YYYY-MM-DD
smalldatetime	YYYY-MM-DD hh:mm:ss
datetime	YYYY-MM-DD hh:mm:ss[.nnn]
datetime2	YYYY-MM-DD hh:mm:ss[.nnnnnnn]
datetimeoffset	YYYY-MM-DD hh:mm:ss[.nnnnnnn] [+/-]hh:mm

Datename() → returns name/value of specific part of date.
select Datename(weekday, '2017-01-14') as Datename
op saturday

Getdate → determines current date and time of computer running the current SQL instance.
→ doesnt include the timezone difference.
select GETDATE() as Systemdate.
op- 2017-01-14 11:11:47.7230728

DATEDIFF → diff btw two dates.
select DateDiff (day, orderdate, shipdate)

You can have year, month, week, day, hr, min, s, ms.

DATEADD → add an interval to the date.
Select DATEADD (day, 20, '2017-01-14')
op - 2017-02-03 00:00:00.000

Configuration and conversion function

@@SERVERNAME - Provides name of local server that is running SQL.
Select @@SERVERNAME as 'server'

CAST/CONVERT → To perform any conversions that do not occur implicitly.
CAST(Hiredate as varchar(20)) As 'Cast' → 2003-02-04
uses default style YYYY-MM-DD
CONVERT (varchar, HireDate, 3) As 'convert' → 04/02/03
- uses date & time style you specify. 3 → dd/mm/yy

Logical and Mathematical fn.
CHOOSE - returns an item from list of values based on position
Select CHOOSE (2, 'A', 'B', 'C') → op - B.

IFF → conditional - ternary a = x ? a : b

IFF (SalesYTD > 20000, 'A', 'B') as bonus.

≡ SalesYTD > 20000 ? A : B.

SIGN

select SIGN(-20) → -1 -1 for -ve
SIGN(20) → +1 +1 for +ve

POWER

Select POWER(3,3) → 3^3 = 9.
POWER(50,3) → 50^3 = 125000

Views - A view can filter some rows from base table or project only some columns from it.

```
Create view view_name AS
  Select id, fname, salary
  from Employees E
  where hiredate > date '2015-01-01';

select * from view_name
```

Complex views

A view can be really complex query (aggregation, join, subquery etc) just be sure to add column names for everything you select.

```
Create view dept_income AS
  Select d.name as Deptname, sum(e.salary)
  From Employees e
  JOIN Department d
   on e.deptid = d.id
  Group by d.name;
```

Materialized view:

a view whose results are physically stored and must be periodically refreshed in order to remain current.

- Therefore useful in storing results of complex queries when realtime results are not required.
- Can be created in oracle & postgre.
- Other db offer similar functionality - SQL indexed views etc.

Section 48.1: PostgreSQL example

similar functionality, such as SQL Server's indexed views or DB2's materialized query tables.

Section 48.1: PostgreSQL example

```
CREATE TABLE mytable (number INT);
INSERT INTO mytable VALUES (1);

CREATE MATERIALIZED VIEW myview AS SELECT * FROM mytable;

SELECT * FROM myview;

number
--------
1
(1 row)


INSERT INTO mytable VALUES(2);

SELECT * FROM myview;

number
--------
1
(1 row)


REFRESH MATERIALIZED VIEW myview;
SELECT * FROM myview;

number
--------
1
2
(2 rows)
```

Comments :

Single Line -

```
Select *
from emp      -- This is a comment.
```

Multi line.

```
/* This is a
   multiline comment */
Select *
from emp
```

Foreign Keys

- ensure data integrity by enforcing that values in one table must match values in other table.
- The datatype of FK must match datatype of referenced key.

A few tips using FK -

* must reference a UNIQUE (or Primary) key in parent table.
* Entering a NULL value in FK does not raise an error
* FK constraints can reference tables within same db
* FK constraints can refer to another column in same table (Self reference)

Creating a table with FK

```
Create table HeroPOwers
(
    ID INT ~~AUT~~ NOTNULL ,
   Herold int references Superheros(ID)
);
```

SEQUENCE

```
Create Sequence orders.seq
Start with 1000
Increment by 1;
```

→ create a sequence with a starting value of 1000 and increment by 1.

General -

```
Create Sequence seqname
Start with initial_val
Incrementby value
MINVALUE min
MAXVALUE max
CYCE/ NOCYCLE;
```

→ minimum value of sequence

→ max ——— " ———

→ cycle- when reach its set limit start from beginning.

nocycle → An exception will be thrown if exceeds max. value.

Using Sequences

seq_name.NEXTVAL → returns next value in sequence.

Can be used for Insert, Update, Select

```
Insert into Orders values (orders_seq.NEXTVAL, 1032);

Select order_seq.NEXTVAL from dual;

Update Orders
  set Order_UID = orders_seq.NEXTVAL
  where custid = 581;
```

Subqueries:

Subquery in FROM clause-

Acts similar to a temporary table that is generated during the execution of the query and lost afterwards.

```
Select Managers.Id, Employees.Salary
 FROM (
   Select Id from Employees where ManagerId is NULL) AS Managers
 JOIN Employees on Managers.Id = Employee.Id
```

Subquery in select clause.

```
Select id, fname, lname,
      (Select count(*) FROM Cars where Cars.customerID =
                  Customers.id ) AS NumCars
 FROM Customers
```

Subquery in where clause.

```
Select * From Employees
  where Salary = (Select max(Salary) from Employees)
```

Correlated Subqueries

Aka Synchronized or Cordinated subqueries are nested queries that make references to the current row of outer query

```
Select EmployeeId
 FROM Employee AS eOuter
  WHERE Salary >
       (Select Avg(Salary)
         FROM Employee eInner
          where eInner.deptId = eOuter.deptId
       )
```

Filter query results using query on different table

select * from employees
where employeeId not in (select empId from supervisor)
→ All employees not in supervisor table.

Execution blocks

Using BEGIN .. END

```
BEGIN
    UPDATE Employees SET Ph = '55723147' where Id = 1;
    UPDATE emp SET salary = 1000 where Id = 2;
END
```

Stored Procedure.

- A prepared SQL that you can save so that the code can be reused over and over again.

```
Create PROCEDURE procedure-name
    @LastName nvarchar(50),
    @FirstName nvarchar(50)
AS
SELECT FirstName, LastName, Dept
FROM tablename
where firstname = @FirstName
    and lastname = @lastname
AND ENDDATE IS NULL.
```

(@LastName, @FirstName: } parameters)

Calling the procedure:

```
EXECUTE procedure-name 'Smith', 'Jones' ;
```

('Smith', 'Jones': params)

or

```
EXEC procedure-name @LastName = 'Jones', @FirstName = 'Sm';
GO
```

TRIGGERS

- are stored programs which are automatically executed when some events occur.

Triggers can be on response of following event

DML – Delete, Insert, Update
DDL – Create, Alter, Drop
DB operation – servererror, logon, logoff, startup, shutdown

Benefits–

- Generating some derived column values automatically

- enforce referential integrity
- event logging, and storing info on table access.
- Auditing
- Synchronous replication of tables.
- Imposing security authorizations
- Prevent invalid transactions.

Syntax -

```
CREATE [OR REPLACE] TRIGGER trigger_name
{ BEFORE | AFTER | INSTEAD OF }
{INSERT [OR] | UPDATE [OR] | DELETE }
[OF colname]
ON table_name
[REFERENCING OLD AS o NEW AS n]
[FOR EACH ROW]
WHEN (condition)
DECLARE
      Declaration statements
BEGIN
      Executable statements
EXCEPTION
      Exception handling statements
END;
```

eg -

```
CREATE OR REPLACE TRIGGER salary_chg
BEFORE DELETE OR INSERT OR UPDATE
ON CUSTOMERS
FOR EACH ROW
when (NEW.ID > 0)
DECLARE
     sal_diff_no;
BEGIN
    sal_diff_no := :NEW.salary - :OLD_SALARY;
    dbms_output.put_line ('Old sal: ' || :OLD.salary);
END;
```

eg -

```
create TRIGGER mytrigger
  ON mytable
  AFTER INSERT
AS
BEGIN
     INSERT INTO myaudit (mytableid, user)
         (Select mytableid, current_user from inserted)
```

```
END
```

Transactions

Simple Transaction

```
BEGIN TRANSACTION
    INSERT INTO deletedemp (empId, date, User)
        ( Select 123, getdate(), curr_user);
    DELETE FROM emp where id = 123;
COMMIT TRANSACTION.
```

Rollback Transaction

```
BEGIN TRY
    BEGIN TRANSACTION
        INSERT INTO Users [id, name, age]
        Values (1, 'Bob', 24)

        DELETE FROM User where name = 'Todd'
    COMMIT TRANSACTION
END TRY
BEGIN CATCH
    ROLLBACK TRANSACTION
END CATCH
```

Savepoint - -you can rollback to a savepoint

```
savepoint <savepoint_name>;
Rollback savepoint-name;
```

Autocommit

```
SET AUTOCOMMIT ON;
    —a———— OFF;
```

SQL keywords are not case sensitive however it is common practice to write them in uppercase

SQL Injection

- is an attempt to access a website's database tables by injecting SQL in form field.
- If webserver does not protect against SQL injection attacks, a hacker can trick the db into running SQL code. which can access all user data, modify, delete db.

Userid : `105 OR 1=1`

Select * from User where Userid = 105 OR 1=1;

will give all users data.

Use SQL parameters for protection.

SQL engine checks params to ensure it is correct for its column and are treated literally and not as a part of SQL to be executed.

```
txtname = getRequestString("UserId");
txtSQL = "SELECT * FROM Users WHERE UserId = @0";
db.execute(txtSQL, txtname)
```

PLSQL.

PL/SQL is a combination of SQL along with procedural features of programming languages.

PL/SQL programming language was developed by

The PL/SQL programming language was developed by Oracle Corporation in the late 1980s as procedural extension language for SQL and the Oracle relational database. Following are certain notable facts about PL/SQL –

- PL/SQL is a completely portable, high-performance transaction-processing language.
- PL/SQL provides a built-in, interpreted and OS independent programming environment.
- PL/SQL can also directly be called from the command-line **SQL*Plus interface**.
- Direct call can also be made from external programming language calls to database.
- PL/SQL's general syntax is based on that of ADA and Pascal programming language.
- Apart from Oracle, PL/SQL is available in **TimesTen in-memory database** and **IBM DB2**.

Features of PL/SQL

PL/SQL has the following features –

- PL/SQL is tightly integrated with SQL.
- It offers extensive error checking.
- It offers numerous data types.
- It offers a variety of programming structures.
- It supports structured programming through functions and procedures.
- It supports object-oriented programming.
- It supports the development of web applications and server pages.

Advantages of PL/SQL

PL/SQL has the following advantages –

- SQL is the standard database language and PL/SQL is strongly integrated with SQL. PL/SQL supports both static and dynamic SQL. Static SQL supports DML operations and transaction control from PL/SQL block. In Dynamic SQL, SQL allows embedding DDL

Advantages of PL/SQL

PL/SQL has the following advantages –

- SQL is the standard database language and PL/SQL is stron
  PL/SQL supports both static and dynamic SQL. Static SQL supp
  transaction control from PL/SQL block. In Dynamic SQL, SQL
  statements in PL/SQL blocks.
- PL/SQL allows sending an entire block of statements to the da
  reduces network traffic and provides high performance for the appl
- PL/SQL gives high productivity to programmers as it can query, tr
  in a database.
- PL/SQL saves time on design and debugging by strong feat
  handling, encapsulation, data hiding, and object-oriented data type
- Applications written in PL/SQL are fully portable.
- PL/SQL provides high security level.
- PL/SQL provides access to predefined SQL packages.
- PL/SQL provides support for Object-Oriented Programming.
- PL/SQL provides support for developing Web Applications and Ser

PL SQL is a block structured language ie - PLSQL programs are divided into logical blocks of code

Each block consists of three sub parts -

Declarations - DECLARE keyword.
- Optional section, defines all variables, cursors, subprograms and other elements to be used in the program.

Executable commands - BEGIN ... END. in btw these.
- mandatory section.
- should have atleast 1 executable line of code which may be NULL to indicate nothing should be executed

Exception Handling → starts with EXCEPTION keyword.
- optional.

```
DECLARE
    <declaration section>
BEGIN.
    <executable command(s)>
EXCEPTION
    <exception handling>
END;
```

eg.
```
DECLARE
    message VARCHAR(20):='Hello World';
BEGIN
    dbms_output.put_line(message);
END;
/
```
→ to run code from command line, you need to type / at beginning of first blank line after last line of code.

PL SQL Identifiers -
constants, variables, exceptions, procedures, cursors, reserved words.
- not case sensitive.
- cant use reserved keyword as identifier.

Delimiters :- a symbol with special meaning.

+, -, *, /	add, subtract, mult, divide
%,	Attribute indicator.
'	char string delimiter

	component selector.
(,)	Expression or list delimiter.
:	Host variable indicator.
,	item separator
"	Quote identifier delimiter.
=	Relational operator.
@	Remote access indicator
;	statement terminator
:=	assignment operator
=>	Association operator.
||	concat
**	exponent
<<, >>	label delimiter (begin and end)
/*, */	multiline comment
--	singleline comment
..	range operator
<, >, <=, >=	relational operator
<>, !=, ~=, ^=	not equal to.

Data types

1) Scalar → single values with no internal components, such as number, Date, Boolean.

Numeric - PLS-Integer, Binary_Integer, Binary_float, Binary_double
Number(prec,scale), Decimal(prec,scale), Numeric(prec,scale)
float, int, integer, smallint, real.

Character - char, varchar2, raw, nchar, nvarchar2, bng, longraw, rowid, Urowid

Boolean - True, false, ~~Null~~

Datetime.

2) LOB - large object (text, image, video clips, sound waveforms).
Bfile, Blob, clob, nclob,

3) Composite: collections, records.

4) Reference. pointers to other data items

Null → nothing.
- can be assigned but cannot be equated with anything, including itself.

Variables:
variable_name [CONSTANT] datatype [NOT NULL] [:= / DEFAULT initial_value]
eg pi CONSTANT double precision := 3.1415;
eg name varchar(25)

Operator precedence

**	exponent
+, -	identity, negation
*, /	mult/div
+, -, ||	add/subtract, concat
comparison	
NOT	
AND	
OR	

Loops

Basic Loop

```
Declare
    x number := 10
BEGIN
    LOOP
        x := x+10
        If x > 50 Then
            exit;
        END IF;
    END LOOP;
END;
/
```

While Loop

```
while condition LOOP
        seq. of statements
END LOOP;
```

eg.

```
Declare c
    a number := 10;
BEGIN
    while a<20 LOOP
        a: a+1;
    END LOOP;
END;
```

For

```
FOR counter IN initial-val..final-val LOOP
        seq. of statements
END LOOP;
```

```
DECLARE
    a number := 10
BEGIN
    FOR a in 10..20 LOOP   // FOR a in REVERSE 10..20 loop
        dbms.out.putline(a)
    END LOOP;
END;
```

Conditions

```
IF condition THEN
    S;
END IF
```

eg
```
if (a<=20) THEN
    c:=c+1
END IF
```

```
IF CONDITION THEN
    S1;
```

```
ELSE
    S2;
END IF
```

```
if-then-elif
  IF condition THEN
      S1;
  ELSIF Condition2 THEN
      S2;
  ELSIF cond3 THEN
      S3;
  ELSE
      S4;
END IF;
```

Case

```
CASE selector
    WHEN val1 THEN S1;
    WHEN val2 THEN S2;
    :
    ELSE  Sn;  -- default case.
END CASE;
```

Strings

Hello that's you → 'Hello that''s you'

Arrays.

Varray

- each element has index, size can be changed dynamically.

Create or replace Type varry-type-name IS varray(n) of <element type>

(n) → no. of elements.

valid attribute name.

element type → data type

Create or Replace TYPE namearray AS varry(3) of Varchar2(3)

OR

Type namearray IS varray(5) of Integer;

```
DECLARE
    type namesarray is varray(5) of integer;
    marks namesarray;
BEGIN
    marks := namearray(1, 2, 3, 4, 90, 95)
    total := marks.count
    FOR i in 1 ... total LOOP
        dbms.putline(marks(i))
```

```
    END LOOP;
END;
/
```

functions and Procedures

return a single value, mainly used to compute and return a value	do not return a value directly. mainly used to perform an action

Procedure

```
create [or Replace] Procedure procedure_name
[(parameter_name [IN|OUT|IN OUT] type [,...])]
{IS|As}
BEGIN
      //Body
END procedure_name;
```

As - standalone procedures.

```
create or replace procedure greet
As
BEGIN
      output.putline('Hello')
END;
/
```

Call -

```
EXECUTE greet;
   OR
BEGIN
      greet;
END;
```

Drop procedure

```
DROP procedure procedurename;
      ——"——  greet;
```

function

```
create or Replace function fn_name
[(parameter_name [IN|OUT|INOUT] type [,...])]
RETURN return_datatype
{IS|As}
BEGIN
      //Body
```

END [function-name];

Call —

```
BEGIN
    c := totalcustomers( )
END;
```

CURSORS

Oracle creates a memory area called context area to process SQL statement, which contains all info needed for processing, eg. no of rows processed etc.

A cursor is a pointer to context area.
PL/SQL controls context area through cursor
- The set of rows the cursor holds is called active set

There are 2 types of cursors

Implicit
- automatically created when SQL is executed.
- We cant control implicit cursors

SQL%.ROWCOUNT => no of rows affected

Explicit
- programmer defined
- should be defined in declaration section of PL/SQL Block.
- It is created on select statements which returns more than one row.

Explicit cursor

Declaring
```
CURSOR c-customers IS
    select id, name, addr from customers;
```

Opening the cursor
```
OPEN c-customers.
```

Fetching the cursor, accessing one row at a time
```
FETCH c-customers INTO c-id, c-name, c-addr;
```

Closing the cursor
```
CLOSE c-customers
```

DECLARE

```
DECLARE
   c_id customers.id%type;
   c_name customers.name%type;
   c_addr customers.address%type;
   CURSOR c_customers is
      SELECT id, name, address FROM customers;
BEGIN
   OPEN c_customers;
   LOOP
   FETCH c_customers into c_id, c_name, c_addr;
      EXIT WHEN c_customers%notfound;
      dbms_output.put_line(c_id || ' ' || c_name || ' ' || c_addr);
   END LOOP;
   CLOSE c_customers;
END;
/
```

Records - later

Exceptions - an error condition during a program execution.

System defined User defined.

Syntax

```
DECLARE
    <declaration section>
BEGIN
    <executable cmd>.
EXCEPTION
    WHEN exception1 THEN
        // handling statement
    WHEN exception2 THEN
        //
    .
    .
    .
    WHEN others THEN        <- default
        // - - -
```

Raising Exceptions

exceptions are raised automatically by db server but user can also raise.

```
DECLARE
    exception_name EXCEPTION;        <- user defined exception.
BEGIN.
    IF condition THEN
        RAISE exception_name;
    END IF
EXCEPTION
    WHEN exception name THEN
     statement ;
END;
```

Date time

ADD_MONTHS(x, y) Add y months to x

LAST_DAY(x) - last day of month.

MONTHS_BETWEEN(x, y)

Next_day(x, day). - returns datetime of next day after x.

New time.

sysDate() → current datetime
CURDate() → ⇒
CURRENT_TIMESTAMP() →

Normalization → reduce redundancy
— eliminate insert, update, delete anomalies.
-structure logically.

FULL NAMES

FULL NAMES	PHYSICAL ADDRESS	MOVIES
Janet Jones	First Street Plot No 4	Pirates Caribbe the Tita
Robert Phil	3rd Street 34	Forgetti Marsha Little Gi
Robert Phil	5th Avenue	Clash of

1NF — Atomic (no multivalued attributes)
- each cell should contain single value
- each record needs to be unique

FULL NAMES

FULL NAMES	PHYSICAL ADDRESS	MOVIES RENTED
Janet Jones	First Street Plot No 4	Pirates of the Caribbean
Janet Jones	First Street Plot No 4	Clash of the Tita
Robert Phil	3rd Street 34	Forgetting Sarah Marshal
Robert Phil	3rd Street 34	Daddy's Little Gi
Robert Phil	5th Avenue	Clash of the Tita

2NF → 1NF

(Stid, courn, courn fee) → dependency

→ No functional dependency.

→ Single column PK that does not functionally depend on any subset of candidate key relation

Rule 1- Be in 1NF

2NF (Second Normal Form) Rules

- Rule 1- Be in 1NF
- Rule 2- Single Column Primary Key that does not functionally dependant on any subset of candidate key relation

It is clear that we can't move forward to make our simple database in 2nd Normalization form unless we partition the table above.

MEMBERSHIP ID	FULL NAMES	PHYSICAL ADDRESS	SALUTATION
1	Janet Jones	First Street Plot No 4	Ms.
2	Robert Phil	3rd Street 34	Mr.
3	Robert Phil	5th Avenue	Mr.

MEMBERSHIP ID	MOVIES RENTED
1	Pirates of the Caribbean
1	Clash of the Titans
2	Forgetting Sarah Marshal
2	Daddy's Little Girls
3	Clash of the Titans

We have divided our 1NF table into two tables viz. Table 1 and Table2. Table 1 contains member information. Table 2 contains information on movies rented.

We have introduced a new column called Membership_id which is the primary key for table 1. Records can be uniquely identified in Table 1 using membership id

Database – Foreign Key

Transitive dependency - changing a non key column may cause any of the other non key column to change

Id	name	Salutation
1	Robert	Mr
2	Janet	Mrs
3	Robert	Mr

Rebeca — Mrs.

A → B
B → C
} transitive dependency.

name → state

state → country.

st, state state, country.

3NF → 2NF

→ no transitive dependency.

NF Example

NF Example

elow is a 3NF example in SQL database:

Membership ID	Full Names	Physical Address	Salutation ID
1	Janet Jones	First Street Plot No 4	2
2	Robert Phil	3rd Street 34	1
3	Robert Phil	5th Avenue	1

Membership ID	Movies Rented
1	Pirates of the Caribbean
1	Clash of the Titans
2	Forgetting Sarah Marshal
2	Daddy's Little Girls
3	Clash of the Titans

Table 2

Salutation ID	Salutation
1	Mr.
2	Ms.
3	Mrs.
4	Dr.

We have again divided our tables and created a new table which stores Salutations.

There are no transitive functional dependencies, and hence our table is in 3NF

In Table 3 Salutation ID is primary key, and in Table 1 Salutation ID is foreign to primary key in Table 3

Now our little example is at a level that cannot further be decomposed to attain higher normal form types of normalization in DBMS. In fact, it is already in higher normalization forms. Separate efforts for moving into ne

BCNF - Boyce Codd Normal Form. (3.5NF).
Even when DB is in 3NF there would still be anomalies if it has more than one candidate key

Keys -

-1 Super key - combo of keys.
-2 Candidate key - set of cols that uniquely identify a record.
-3 PK - no nulls, unique
-4 FK
-5 Alternate key - candidate key, not selected as PK currently
-6 Composite key - eg - PK(id, name)
-7 Unique key - 1 null allowed, unique

Possible Candidate Keys