Weeks 3 & 4: SQL

The SQL Query Language

Select Statements

Joins, Aggregate and Nested Queries

Insertions, Deletions and Updates

Assertions, Views, Triggers and Access Control

SQL as a Query Language

- SQL expresses queries in declarative way — queries specify the properties of the result, not the way to obtain it.
- Queries are translated by the query optimizer into the procedural language internal to the DBMS.
- The programmer focuses on **readability**, not on efficiency.

SQL Queries

- SQL queries are expressed by the select statement.
- Syntax:
  select *AttrExpr* [[**as**] *Alias*] {, *AttrExpr* [[**as**] *Alias*] }
  from *Table* [[**as**] *Alias*] {, [[**as**] *Alias*] }
  [**where** *Condition*]
- The three parts of the query are usually called: *target list*, **from** *clause*, **where** *clause*.
- The query first builds the Cartesian product of the tables in the **from** clause, then selects only the rows that satisfy the condition in the **where** clause and for each row evaluates the attribute expressions in the target list.

Example Database

EMPLOYEE

FirstName	Surname	Dept	Office	Salary	City
Mary	Brown	Administration	10	45	London
Charles	White	Production	20	36	Toulouse
Gus	Green	Administration	20	40	Oxford
Jackson	Neri	Distribution	16	45	Dover
Charles	Brown	Planning	14	80	London
Laurence	Chen	Planning	7	73	Worthing
Pauline	Bradshaw	Administration	75	40	Brighton
Alice	Jackson	Production	20	46	Toulouse

DEPARTMENT

DeptName	Address	City
Administration	Bond Street	London
Production	Rue Victor Hugo	Toulouse
Distribution	Pond Road	Brighton
Planning	Bond Street	London
Research	Sunset Street	San José

Simple SQL Query

→"Find the salaries of employees named Brown":

```
select Salary as Remuneration
from Employee
where Surname = 'Brown'
```

→Result:

Remuneration
45
80

* in the Target List

→"Find all the information relating to employees named Brown":

```
select *
from Employee
where Surname = 'Brown'
```

→Result:

FirstName	Surname	Dept	Office	Salary	City
Mary	Brown	Administration	10	45	London
Charles	Brown	Planning	14	80	London

Attribute Expressions

→Find the monthly salary of the employees named White:

```
select Salary / 12 as
MonthlySalary
from Employee
where Surname = 'White'
```

→Result:

MonthlySalary
3.00

Simple Join Query

→"Find the names of employees and their cities of work":

```
select Employee.FirstName,
   Employee.Surname, Department.City
from Employee, Department
where Employee.Dept = Department.DeptName
```

→Result:

FirstName	Surname	City
Mary	Brown	London
Charles	White	Toulouse
Gus	Green	London
Jackson	Neri	Brighton
Charles	Brown	London
Laurence	Chen	London
Pauline	Bradshaw	London
Alice	Jackson	Toulouse

Table Aliases

→"Find the names of employees and the cities where they work" (using an alias):

```
select FirstName, Surname, D.City
from Employee, Department D
where Dept = DeptName
```

→Result:

FirstName	Surname	City
Mary	Brown	London
Charles	White	Toulouse
Gus	Green	London
Jackson	Neri	Brighton
Charles	Brown	London
Laurence	Chen	London
Pauline	Bradshaw	London
Alice	Jackson	Toulouse

Predicate Conjunction

→"Find the first names and surnames of employees who work in office number 20 of the Administration department":

```
select FirstName, Surname
from Employee
where Office = '20' and
      Dept = 'Administration'
```

→Result:

FirstName	Surname
Gus	Green

Predicate Disjunction

→"Find the first names and surnames of employees who work in either the Administration or the Production department":

```
select FirstName, Surname
from Employee
where Dept = 'Administration' or
      Dept = 'Production'
```

→Result:

FirstName	Surname
Mary	Brown
Charles	White
Gus	Green
Pauline	Bradshaw
Alice	Jackson

Complex Logical Expressions

→"Find the first names of employees named Brown who work in the Administration department or the Production department":

```
select FirstName
from Employee
where Surname = 'Brown' and
      (Dept = 'Administration' or
       Dept = 'Production')
```

→Result:

FirstName
Mary

Operator `like`

→"Find employees with surnames that have 'r' as the second letter and end in 'n'":

```
select *
from Employee
where Surname like '_r%n'
```

0 or more chars

exactly 1 char

→Result:

FirstName	Surname	Dept	Office	Salary	City
Mary	Brown	Administration	10	45	London
Gus	Green	Administration	20	40	Oxford
Charles	Brown	Planning	14	80	London

Management of Null Values

→Null values may mean that:
- ✓a value is not applicable
- ✓a value is applicable but unknown
- ✓it is unknown if a value is applicable or not

→SQL-89 uses a two-valued logic
- ✓a comparison with *null* returns FALSE

→SQL-2 uses a three-valued logic
- ✓a comparison with *null* returns UNKNOWN

→To test for null values:

Attribute **is [not] null**

Algebraic Interpretation of SQL Queries

→The generic query:

select $T_1.Attr_{11}, \ldots, T_h.Attr_{hm}$
from $Table_1\ T_1, \ldots, Table_n\ T_n$
where *Condition*

corresponds to the relational algebra query:

$$\pi_{T_1.Attr_{11},\ldots,T_h.Attr_{hm}}(\sigma_{Condition}(Table_1 \times \ldots \times Table_n))$$

Duplicates

→In the relational algebra and calculus the results of queries do not contain duplicates.

→In SQL, tables may have identical rows.

→Duplicates can be removed using the keyword ***distinct***:

```
select City
from Department
```

City
London
Toulouse
Brighton
London
San José

```
select distinct City
from Department
```

City
London
Toulouse
Brighton
San José

Joins in SQL-2

→SQL-2 introduced an alternative syntax for the representation of joins, representing them explicitly in the *from* clause:

select *AttrExpr* [[**as**] *Alias*] {, *AttrExpr* [[**as**] *Alias*
from *Table* [[**as**] *Alias*]
{[*JoinType*] **join** *Table*
[[**as**] *Alias*] **on** *JoinConditions* }
[**where** *OtherCondition*]

→*JoinType* can be any of ***inner, right [outer], left [outer] or full [outer].***

→The keyword ***natural*** may precede *JoinType* (rarely implemented).

Inner Join in SQL-2

→"Find the names of the employees and the cities in which they work":

```
select FirstName, Surname, D.City
from Employee inner join Department as D
     on Dept = DeptName
```

→Result:

FirstName	Surname	City
Mary	Brown	London
Charles	White	Toulouse
Gus	Green	London
Jackson	Neri	Brighton
Charles	Brown	London
Laurence	Chen	London
Pauline	Bradshaw	London
Alice	Jackson	Toulouse

Another Example: Drivers and Cars

DRIVER

FirstName	Surname	DriverID
Mary	Brown	VR 2030020Y
Charles	White	PZ 1012436B
Marco	Neri	AP 4544442R

AUTOMOBILE

CarRegNo	Make	Model	DriverID
ABC 123	BMW	323	VR 2030020Y
DEF 456	BMW	Z3	VR 2030020Y
GHI 789	Lancia	Delta	PZ 1012436B
BBB 421	BMW	316	MI 2020030U

Left Join

→"Find all drivers and their cars, if any":

```
select FirstName,Surname,
 Driver.DriverID,CarRegNo,Make,Model
from Driver left join Automobile on
 (Driver.DriverID =
 Automobile.DriverID)
```

→Result:

FirstName	Surname	DriverID	CarRegNo	Make	Model
Mary	Brown	VR 2030020Y	ABC 123	BMW	323
Mary	Brown	VR 2030020Y	DEF 456	BMW	Z3
Charles	White	PZ 1012436B	GHI 789	Lancia	Delta
Marco	Neri	AP 4544442R	NULL	NULL	NULL

Full Join

→"Find all possible drivers and their cars":

```
select
 FirstName,Surname,Driver.DriverID
    CarRegNo, Make, Model
from Driver full join Automobile on
 (Driver.DriverID =
 Automobile.DriverID)
```

→Result:

FirstName	Surname	DriverID	CarRegNo	Make	Model
Mary	Brown	VR 2030020Y	ABC 123	BMW	323
Mary	Brown	VR 2030020Y	DEF 456	BMW	Z3
Charles	White	PZ 1012436B	GHI 789	Lancia	Delta
Marco	Neri	AP 4544442R	NULL	NULL	NULL
NULL	NULL	NULL	BBB 421	BMW	316

Table Variables

→Table aliases may be interpreted as table variables. These correspond to the renaming operator ρ.

→"Find all first names and surnames of employees who have the same surname and different first names with someone in the Administration department":

```
select E1.FirstName, E1.Surname
from Employee E1, Employee E2
where E1.Surname = E2.Surname and
      E1.FirstName <> E2.FirstName and
      E2.Dept = 'Administration'
```

→Result:

FirstName	Surname
Charles	Brown

The `order by` Clause

→***order by*** — appearing at the end of a query — orders the rows of the result; syntax:

order by *OrderingAttribute* [***asc*** | ***desc***]
{, OrderingAttribute [***asc*** | ***desc*** *] }*

→Extract the content of the **`Automobile`** table in descending order with respect to make and model:

```
select *
from Automobile
order by Make desc, Model desc
```

→Result:

CarRegNo	Make	Model	DriverID
GHI 789	Lancia	Delta	PZ 1012436B
DEF 456	BMW	Z3	VR 2030020Y
ABC 123	BMW	323	VR 2030020Y
BBB 421	BMW	316	MI 2020030U

Aggregate Queries

→Aggregate queries cannot be represented in relational algebra.

→The result of an aggregate query depends on functions that take as an argument a set of tuples.

→SQL-2 offers five aggregate operators:

- ✓**`count`**
- ✓**`sum`**
- ✓**`max`**
- ✓**`min`**
- ✓**`avg`**

Operator count

- **count** returns the number of elements (or, distinct elements) of its argument:

  count(< * | [**distinct** | **all**] *AttributeList* >)

- "Find the number of employees":

```
select count(*)from Employee
```

- "Find the number of different values on attribute Salary for all tuples in Employee":

```
select count(distinct Salary)
       from Employee
```

- "Find the number of tuples in Employee having non-null values on the attribute Salary":

```
select count(all Salary) from Employee
```


Sum, Average, Maximum and Minimum

- Syntax:

  < **sum** | **max** | **min** | **avg** > ([**distinct** | **all**] *AttributeExpr*)

- "Find the sum of all salaries for the Administration department":

```
select sum(Salary) as SumSalary
from Employee
where Dept = 'Administration'
```

- Result:

SumSalary
125

Aggregate Queries with Join

- "Find the maximum salary among the employees who work in a department based in London":

```
select max(Salary) as MaxLondonSal
from Employee, Department
where Dept = DeptName and
      Department.City = 'London'
```

- Result:

MaxLondonSal
80

Aggregate Queries and Target List

- Incorrect query:

```
select FirstName,Surname,max(Salary)
from Employee, Department
where Dept = DeptName and
      Department.City = 'London'
```

  (Whose name? The target list must be homogeneous!)

- Find the maximum and minimum salaries among all employees:

```
select max(Salary) as MaxSal,
       min(Salary) as MinSal
from Employee
```

- Result:

MaxSal	MinSal
80	36

Group by Queries

- Queries may apply aggregate operators to subsets of rows.
- "Find the sum of salaries of all the employees of the same department":

```
select Dept, sum(Salary) as TotSal
from Employee
group by Dept
```

- Result:

Dept	TotSal
Administration	125
Distribution	45
Planning	153
Production	82

Semantics of group by Queries - I

- First, the query is executed without **group by** and without aggregate operators:

```
select Dept, Salary
from Employee
```

Dept	Salary
Administration	45
Production	36
Administration	40
Distribution	45
Planning	80
Planning	73
Administration	40
Production	46

Semantics of group by Queries - II

- … then the query result is divided in subsets characterized by the same values for the attributes appearing as argument of the **group by** clause (in this case attribute Dept):
- Finally, the aggregate operator is applied separately to each subset

Dept	Salary
Administration	45
Administration	40
Administration	40
Distribution	45
Planning	80
Planning	73
Production	36
Production	46

Dept	TotSal
Administration	125
Distribution	45
Planning	153
Production	82

group by Queries and Target List

- Incorrect query:

```
select Office from Employee
group by Dept
```

- Incorrect query:

```
select DeptName, count(*), D.City
from Employee E join Department D
     on (E.Dept = D.DeptName)
group by DeptName
```

- Correct query:

```
select DeptName,count(*),D.City
from Employee E join Department D
     on (E.Dept = D.DeptName)
group by DeptName, D.City
```

Group Predicates

→When conditions are defined on the result of an aggregate operator, it is necessary to use the **having** clause

→"Find which departments spend more than 100 on salaries":

```
select Dept
from Employee
group by Dept
having sum(Salary) > 100
```

→Result:

Dept
Administration
Planning

where or having?

→Only predicates containing aggregate operators should appear in the argument of the **having** clause

→"Find the departments where the average salary of employees working in office number 20 is higher than 25":

```
select Dept
from Employee
where Office = '20'
group by Dept
having avg(Salary) > 25
```


Syntax of an SQL Query ...so far!

→Considering all clauses discussed so far, the syntax of an SQL query is:

select *TargetList*
from *TableList*
[**where** *Condition*]
[**group by** *GroupingAttributeList*]
[**having** *AggregateCondition*]
[**order by** *OrderingAttributeList*]

Set Queries

→A single select statement cannot represent any set operation.

→Syntax:

SelectSQL { <**union** | **intersect** | **except** >
[**all**] *SelectSQL* }

→"Find all first names and surnames of employees":

```
select FirstName as Name from Employee
 union
select Surname as Name from Employee
```

→Duplicates are removed (unless the **all** option is used)

Intersection

→"Find surnames of employees that are also first names":

```
select FirstName as Name
from Employee
 intersect
select Surname as Name
from Employee
```

(equivalent to:

```
select E1.FirstName as Name
from Employee E1, Employee E2
where E1.FirstName = E2.Surname  )
```


Difference

→"Find the surnames of employees that are not first names":

```
select Surname as Name
from Employee
 except
select FirstName as Name
 from Employee
```

→Can also be represented with a nested query (see later.)

Nested Queries

→A **where** clause may include predicates that:

✓Compare an attribute (or attribute expression) with the result of an SQL query;

syntax: *ScalarValue Op* <**any** | **all**> *SelectSQL*

any – the predicate is true if at least one row returned by *SelectSQL* satisfies the comparison

all – predicate is true if all rows satisfy comparison;

✓Use the existential quantifier on an SQL query;

syntax: **exists** *SelectSQL*

the predicate is true if *SelectSQL* is non-empty.

→The query appearing in the **where** clause is called a ***nested query***.

Simple Nested Query

→"Find the employees who work in departments in London":

```
select FirstName, Surname
from Employee
where Dept = any (select DeptName
    from Department
    where City = 'London')
```

(Equivalent to:

```
select FirstName, Surname
from Employee, Department D
where Dept = DeptName and
      D.City = 'London'    )
```

...Another...

- "Find employees of the Planning department, having the same first name as a member of the Production department":
  - (with a nested query)

```
select FirstName,Surname from Employee
where Dept = 'Plan' and FirstName = any
       (select FirstName from Employee
            where Dept = 'Prod')
```

  - (without nested query)

```
select E1.FirstName,E1.Surname
from Employee E1, Employee E2
where E1.FirstName=E2.FirstName and
       E2.Dept='Prod' and E1.Dept='Plan'
```


Negation with Nested Queries

- "Find departments where there is no one named Brown":

```
select DeptName
from Department
where DeptName <>
     all (select Dept from Employee
        where Surname = 'Brown')
```

- (Alternatively:)

```
select DeptName from Department
   except
select Dept from Employee
where Surname = 'Brown'
```


Operators in and not in

- Operator **in** is a shorthand for **= any**

```
select FirstName, Surname
from Employee
where Dept in (select DeptName
               from Department
               where City = 'London')
```

- Operator **not in** is a shorthand for **<> all**

```
select DeptName
from Department
where DeptName not in
   (select Dept from Employee
       where Surname = 'Brown')
```


max and min within a Nested Query

- Queries using the aggregate operators **max** and **min** can be expressed with nested queries
- "Find the department of the employee earning the highest salary":
  - with **max**:

```
select Dept from Employee
where Salary in (select max(Salary)
                 from Employee)
```

  - with a nested query:

```
select Dept from Employee
where Salary >= all (select Salary
                     from Employee)
```

A Complex Nested Query

- A nested query may use variables of the outer query ('transfer of bindings').
- Semantics: the nested query is evaluated for each row of the outer query.
- "Find all persons who have the same first name and surname with someone else ("synonyms"), but different tax codes":

```
select * from Person P
where exists (select * from Person P1
  where P1.FirstName = P.FirstName
    and P1.Surname = P.Surname
    and P1.TaxCode <> P.TaxCode)
```


...Another...

- "Find all persons who have no synonyms":

```
select * from Person P
where not exists
 (select * from Person P1
   where P1.FirstName = P.FirstName
     and P1.Surname = P.Surname
     and P1.TaxCode <> P.TaxCode)
```


Tuple Constructors

- The comparison within a nested query may involve several attributes bundled into a tuple.
- A tuple constructor is represented in terms of a pair of angle brackets.
- The previous query can also be expressed as:

```
select * from Person P
where <FirstName,Surname> not in
  (select FirstName,Surname
     from Person P1
      where P1.TaxCode <> P.TaxCode)
```


Comments on Nested Queries

- The use of nested queries may produce less declarative queries, but often results in improved readability.
- Complex queries can become very difficult to understand.
- The use of variables must respect scoping conventions: a variable can be used only within the query where it is defined, or within a query that is recursively nested in the query where it is defined.

Scope of Variables

→Incorrect query:

```
select * from Employee
where Dept in
  (select DeptName from Department D1
    where DeptName = 'Production') or
      Dept in (select DeptName
        from Department D2
            where D2.City = D1.City)
```

→What's wrong?

Data Modification in SQL

→Modification statements include:

- ✓Insertions (**insert**);
- ✓Deletions (**delete**);
- ✓Updates of attribute values (**update**).

→All modification statements operate on a set of tuples (no duplicates.)

→In the ***condition*** part of an update statement it is possible to access other relations.

Insertions

→Syntax:

insert into *TableName* [(*AttributeList*)]
 < **values** (*ListOfValues*) | *SelectSQL* >

→Using **values**:

```
insert into Department(DeptName,City)
  values('Production','Toulouse')
```

→Using a subquery:

```
insert into LondonProducts
  (select Code, Description
   from Product
   where ProdArea = 'London')
```


Notes on Insertions

→The ordering of attributes (if present) and of values is meaningful -- first value for the first attribute, etc.

→If *AttributeList* is omitted, all the relation attributes are considered, in the order they appear in the table definition.

→If *AttributeList* does not contain all the relation attributes, left-out attributes are assigned default values (if defined) or the null value.

Deletions

→Syntax:

delete from *TableName* [**where** Condition]

→"Remove the Production department":

```
delete from Department
  where DeptName = 'Production'
```

→"Remove departments with no employees":

```
delete from Department
  where DeptName not in
     (select Dept from Employee)
```


Notes on Deletions

→The **delete** statement removes from a table all tuples that satisfy a condition.

→The removal may produce deletions from other tables — if a referential integrity constraint with **cascade** policy has been defined.

→If the **where** clause is omitted, **delete** removes all tuples. For example, to remove all tuples from **Department** (keeping the table schema):

```
delete from Department
```

→To remove table **Department** completely (content and schema):

```
drop table Department cascade
```


Updates

→Syntax:

update *TableName*
set *Attribute* = < *Expression* | *SelectSQL* | **null** | **default** >
{, *Attribute* = < *Expression* | *SelectSQL* | **null** | **default** >}
[**where** *Condition*]

→Examples:

```
update Employee set Salary = Salary + 5
  where RegNo = 'M2047'
update Employee set Salary = Salary * 1.1
   where Dept = 'Administration'
```


Notes on Updates

→As with any side effect statement, the order of updates is important:

```
update Employee
  set Salary = Salary * 1.1
  where Salary <= 30
update Employee
  set Salary = Salary * 1.15
  where Salary > 30
```

→In this example, some employees may get a double raise! How can we fix this?

Generic Integrity Constraints

- The **check** clause can be used to express arbitrary constraints during schema definition.
- Syntax:

  check (*Condition*)
- *Condition* is what can appear in a **where** clause — including nested queries.
- For example, the definition of an attribute **Superior** in the schema of table **Employee**:

```
Superior character(6)
  check (RegNo like “1%” or
     Dept = (select Dept from Employee E
        where E.RegNo = Superior)
```


Assertions

- Assertions permit the definition of constraints independently of table definitions.
- Assertions are useful in many situations -- e.g., to express generic inter-relational constraints.
- An assertion associates a name to a **check** clause; syntax:

create assertion *AssertName* **check** (*Condition*)

- "There must always be at least one tuple in table **Employee**":

```
create assertion AlwaysOneEmployee
  check (1 <= (select count(*)
                 from Employee))
```


Views

- Views are "virtual tables" whose rows are computed from other tables (*base relations*).
- Syntax:

  create view *ViewName* [(*AttributeList*)] **as** *SelectSQL* [**with** [**local|cascaded**] **check option**]
- Examples:

```
create view AdminEmployee
   (RegNo,FirstName,Surname,Salary) as
 select RegNo,FirstName,Surname,Salary
 from Employee
 where Dept = ‘Admin’ and Salary > 10
create view JuniorAdminEmployee as
 select * from AdminEmployee
 where Salary < 50 with check option
```


Notes on Views

- SQL views cannot be mutually dependent (no recursion).
- **check option** executes when a view is updated.
- Views can be used to formulate complex queries -- views decompose a problem and produce more readable solutions.
- Views are sometimes necessary to express certain queries:
  - ✓ Queries that combine and nest several aggregate operators;
  - ✓ Queries that make fancy use of the union operator.

Views and Queries

→"Find the department with highest salary expenditures" (without using a view):

```
select Dept from Employee
 group by Dept
 having sum(Salary) >= all
   (select sum(Salary) from
Employee
      group by Dept)
```

→This solution may not work with all SQL systems.

Views and Queries

→"Find the department with highest salary expenditures" (using a view):

```
create view SalBudget
(Dept,SalTotal) as
 select Dept,sum(Salary)
  from Employee group by Dept
select Dept from SalBudget
 where SalTotal =
  (select max(SalTotal) from
SalBudget)
```


Views and Queries

→"Find the average number of offices per department":

Incorrect solution (SQL does not allow a cascade of aggregate operators):

```
select avg(count(distinct Office))
from Employee group by Dept
```

Correct solution (using a view):

```
create view
  DeptOff(Dept,NoOfOffices) as
   select Dept,count(distinct Office)
   from Employee group by Dept
select avg(NoOfOffices)
  from DeptOffice
```


Access Control

→Every element of a schema can be protected (tables, attributes, views, domains, etc.)

→The owner of a resource (the creator) assigns privileges to the other users.

→A predefined user `_system` represents the database administrator and has access to all resources.

→A privilege is characterized by:
- ✓a resource;
- ✓the user who grants the privilege;
- ✓the user who receives the privilege;
- ✓the action that is allowed on the resource;
- ✓whether or not the privilege can be passed on to other users.

Types of Privileges

→SQL offers six types of privilege:

- ✓**insert:** to insert a new object into the resource;
- ✓**update:** to modify the resource content;
- ✓**delete:** to remove an object from the resource;
- ✓**select:** to access the resource content;
- ✓**references:** to build a referential integrity constraint with the resource;
- ✓**usage:** to use the resource in a schema definition (e.g., a domain)

grant and revoke

→To grant a privilege to a user:

grant < *Privileges* | **all privileges** > **on** *Resource*
to *Users* [**with grant option**]

→**grant option** specifies whether the privilege can be propagated to other users.

→For example,

```
grant select on Department to
 Stefano
```

→To take away privileges:

revoke *Privileges* **on** *Resource* **from** *Users*
[**restrict** | **cascade**]

Database Triggers

→Triggers (also known as ECA rules) are element of the database schema.

→General form:

on <*event*> **when** <*condition*> **then** <*action*>

- ✓*Event-* request to execute database operation
- ✓*Condition* - predicate evaluated on database state
- ✓*Action* – execution of procedure that might involve database updates

→Example:
on "updating maximum enrollment limit"
if "# registered > new max enrollment limit "
then "deregister students using LIFO policy"

Trigger Details

→Activation — occurrence of the *event* that activates the trigger.

→Consideration — the point, after activation, when *condition* is evaluated; this can be *immediate* or *deferred*.

- ✓*Deferred* means that *condition* is evaluated when the database operation (*transaction*) currently executing requests to commit.

→*Condition* might refer to both the state before and the state after *event* occurs.

Trigger Execution

- This is the point when the *action* part of the trigger is carried out.
- With deferred consideration, execution is also deferred.
- With immediate consideration, execution can occur immediately after consideration or it can be deferred
  - ✓ If execution is immediate, execution can occur before, after, or instead of triggering event.
  - ✓ Before triggers adapt naturally to maintaining integrity constraints: violation results in rejection of event.

Event Granularity

Event granularity can be:

- ***Row-level***: the event involves change of a single row,
  - ✓ This means that a single **`update`** statement might result in multiple events;
- ***Statement-level***: here events result from the execution of a whole statement; for example, a single **`update`** statement that changes multiple rows constitutes a single event.

Multiple Trigger Executions

- Should we allow multiple triggers to be activated by a single event?
- If so, how do we handle trigger execution?
  - ✓ Evaluate one condition at a time and if true immediately execute action; or
  - ✓ Evaluate all conditions, then execute all associated actions.
- The execution of an action can affect the truth of a subsequently evaluated condition so the choice is significant.

Triggers in SQL-3

- Events: **`insert`**, **`delete`**, or **`update`** statements or changes to individual rows caused by these statements.
- Condition: Anything allowed in a **`where`** clause.
- Action: An individual SQL statement or a program written in the language of Procedural Stored Modules (PSM) -- which can contain embedded SQL statements.

Triggers in SQL-3

- →Consideration = *immediate* – condition can refer to both the state of the affected row or table before *and* after the event occurs.
- →Execution = *immediate* – can be before or after the execution of the triggering event
- →Note that the action of a before-trigger cannot modify the database.
- →Granularity: Both *row-level* and *statement-level.*

Before-Trigger with Row Granularity

```
CREATE TRIGGER Max_EnrollCheck
  BEFORE INSERT ON Transcript
     REFERENCING NEW AS N   --row to be added
  FOR EACH ROW
  WHEN
  ((SELECT COUNT (T.StudId) FROM Transcript T
     WHERE T.CrsCode = N.CrsCode
           AND T.Semester = N.Semester)
   >=
   (SELECT C.MaxEnroll FROM Course C
     WHERE C.CrsCode = N.CrsCode ))
  THEN ABORT TRANSACTION
```

Check that enrollment ≤ limit

Action

After-Trigger with Row Granularity

```
CREATE TRIGGER LimitSalaryRaise
  AFTER UPDATE OF Salary ON Employee
  REFERENCING OLD AS O
              NEW AS N
  FOR EACH ROW
  WHEN (N.Salary - O.Salary > 0.05 * O.Salary)
  THEN UPDATE Employee       -- action
     SET Salary = 1.05 * O.Salary
     WHERE Id = O.Id
```

No salary raises greater than 5%

[Note: The action itself is a triggering event; however, in this case a chain reaction is not possible.]

After-Trigger with Statement Granularity

```
CREATE TRIGGER RecordNewAverage
  AFTER UPDATE OF Salary ON Employee
  FOR EACH STATEMENT
  THEN INSERT INTO Log
    VALUES (CURRENT_DATE,
            SELECT AVG (Salary)
            FROM Employee)
```

Keep track of salary averages in the log