

## Introduction to Databases

---

### Querying Relational Databases using SQL Part--4

---

**Cong Gao**

Professor

School of Computer Science and Engineering  
Nanyang Technological University, Singapore

# Summary and roadmap

- Introduction to SQL
- SELECT  
FROM  
WHERE
- Eliminating duplicates
- Renaming attributes
- Expressions in SELECT Clause
- Patterns for Strings
- Ordering
- Joins
- Subquery
- Aggregations
- UNION, INTERSECT, EXCEPT
- NULL
- Outerjoin
- Insert/Delete tuples
- Create/Alter/Delete tables
- Constraints (primary key)

- Next

- More constraints
  - FOREIGN KEY
  - CHECK
  - ASSERTION
  - Trigger
- Views
- Indexes

# Views -- Motivation

Author		Wrote	
<u>PenName</u>	RealName	<u>PenName</u>	<u>Article</u>
...	...	...	...

RealName	<u>Article</u>
...	...

- `SELECT RealName, Article`  
`FROM Author, Wrote`  
`WHERE Author.PenName = Wrote.PenName`
- Assume that we frequently need to check the real names of the authors of some articles
- Can we somehow make the above join results more easily accessible?

# Views

**Author**

<u>PenName</u>	RealName
...	...

**Wrote**

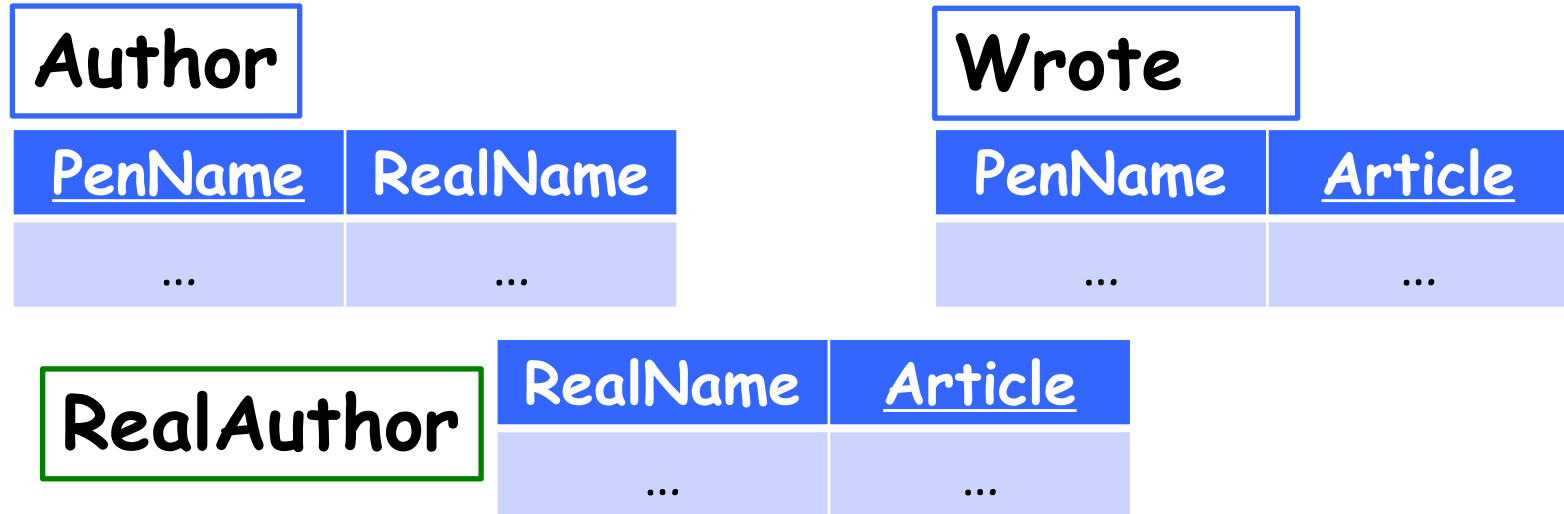
<u>PenName</u>	<u>Article</u>
...	...

**RealAuthor**

RealName	<u>Article</u>
...	...

- **CREATE VIEW** RealAuthor **AS**  
SELECT RealName, Article  
FROM Author, Wrote  
WHERE Author.PenName = Wrote.PenName;
- This view can then be used as a table
- SELECT Article FROM RealAuthor  
WHERE RealName = 'Chris'

# Tables vs. Views



- Tables are physically stored in the database
- Views are NOT physically stored but are computed on the fly

# Queries on Views

Wrote
-------

Author	<u>PenName</u>	RealName
--------	----------------	----------

PenName	<u>Article</u>
---------	----------------

RealAuthor
------------

RealName	<u>Article</u>
----------	----------------

- We have a view as follows:  
**CREATE VIEW** RealAuthor **AS**  
SELECT RealName, Article  
FROM Author, Wrote  
WHERE Author.PenName = Wrote.PenName;

- We have a query like this:  
SELECT Article FROM RealAuthor  
WHERE RealName = 'Chris'

- What the DBMS would do:  
SELECT RealName, Article  
FROM Author, Wrote  
WHERE Author.PenName = Wrote.PenName  
AND RealName = 'Chris';

Query Rewriting

# Deleting Views

Author	<u>PenName</u>	RealName
--------	----------------	----------

Wrote
-------

PenName	<u>Article</u>
---------	----------------

- DROP VIEW RealAuthor;

RealAuthor
------------

RealName	<u>Article</u>
----------	----------------

# Summary on Views

## What is view?

A **view** is a query over the **base relations** to produce another relation.

## View is virtual

It is considered **virtual** because it does not usually exist physically.

## To the user

A view appears just like any other table and can be present in any SQL query where a table is present.



# View Materialization

## The idea

Physically creating and keeping a temporary table  
Syntax: CREATE MATERIALIZED VIEW ... AS..

## Pro and Con

Pro: Other queries defined on the view can be answered efficiently

Con: Maintaining correspondence between the base table and the view when the base table is updated

# Views vs. Temporary Views

**Author**

<u>PenName</u>	RealName
...	...

**Wrote**

<u>PenName</u>	<u>Article</u>
...	...

- **CREATE VIEW** RealAuthor **AS**  
SELECT RealName, Article  
FROM Author, Wrote  
WHERE Author.PenName = Wrote.PenName;
- **WITH** RealAuthor **AS**  
(SELECT RealName, Article  
FROM Author, Wrote  
WHERE Author.PenName = Wrote.PenName)  
SELECT Article  
FROM RealAuthor  
WHERE RealAuthor.RealName = 'Chris';

# Views vs. Temporary Views

- A view persists after its creation
- It will be there unless you **explicitly** delete it
- **CREATE VIEW** RealAuthor **AS**  
SELECT RealName, Article  
FROM Author, Wrote  
WHERE Author.PenName =  
Wrote.PenName;
- **DROP VIEW** RealAuthor;

# Views vs. Temporary Views

- A temporary view only persists during the execution of a query
- It will not be there once the query is done
- **WITH** RealAuthor **AS**  
(SELECT RealName, Article  
FROM Author, Wrote  
WHERE Author.PenName = Wrote.PenName)  
SELECT Article  
FROM RealAuthor  
WHERE RealAuthor.RealName = 'Cedric';
- Here, the temporary view RealAuthor is gone once the query is finished

# SELECT INTO vs. VIEWS

- `SELECT RealName, Article  
INTO RealAuthor  
FROM Author, Wrote  
WHERE Author.PenName =  
Wrote.PenName`

A Table is  
created  
(Physically)

- `CREATE VIEW RealAuthor AS  
SELECT RealName, Article  
FROM Author, Wrote  
WHERE Author.PenName =  
Wrote.PenName;`

A Definition  
is created  
(Virtually)

# Summary and roadmap

- Introduction to SQL
- SELECT  
FROM  
WHERE
- Eliminating duplicates
- Renaming attributes
- Expressions in SELECT Clause
- Patterns for Strings
- Ordering
- Joins
- Subquery
- Aggregations
- UNION, INTERSECT, EXCEPT
- NULL
- Outerjoin
- Insert/Delete tuples
- Create/Alter/Delete tables
- Constraints: primary key
- Views

## • Next

- More constraints
  - FOREIGN KEY
  - CHECK
  - ASSERTION
  - Trigger
- Views
- Indexes

# Constraints

- A constraint is a requirement on tuples/tables that the DBMS needs to enforce
- Examples
  - PRIMARY KEY
  - UNIQUE
  - NOT NULL
- More to be discussed
  - FOREIGN KEY
  - CHECK
  - ASSERTION
  - Trigger

# FOREIGN KEY Example

Beers

Name

manf

Sells

Bar

Beer

Price

- Any value in Sells.Beer should appear in Beer.Name
  - values for certain values must make sense
- How to ensure this when we create Sells?



# Example

Requirement: Beers.Name **must be** declared as either PRIMARY KEY or UNIQUE

```
CREATE TABLE    Beers (  
    name        CHAR(30)    PRIMARY KEY,  
    manf        VARCHAR(50)  
);
```

```
CREATE TABLE    Sells (  
    bar         CHAR(20),  
    beer        CHAR(30),  
    price       REAL,  
    FOREIGN KEY beer REFERENCES Beers(name)  
);
```

# Foreign Key

## What it is?

- Consider **Sells(Bar, Beer, Price)**
- The beer value might be a “real” beer
  - Can be found in the **Beers** relation
- A constraint that requires a beer in **Sells** to be a beer in **Beers** is called a **foreign key constraint**

## Expressing Foreign Key

- Use the keyword **REFERENCES**. Two methods:
  1. Within the declaration of an attribute, when FK has only one attribute
  2. An separate element of Create Table: **FOREIGN KEY (<attribute list>) REFERENCES <relation> (<attributes>)**
- **Referenced attributes** must be declared as **Primary Key** or **UNIQUE** in the other **relation**

# FOREIGN KEY (cont.)

Visits

Name

Shop

Purchase

Name

Shop

Product

- Any value combination in Purchase(Name, Shop) should appear in Visits(Name, Shop)
- CREATE TABLE Purchase(  
Name VARCHAR(30),  
Shop VARCHAR(30),  
Product VARCHAR(30),  
**FOREIGN KEY** (Name, Shop) **REFERENCES**  
Visits(Name, Shop) );
- Requirement: Visits(Name, Shop) is declared as either PRIMARY KEY or UNIQUE

# Foreign Key Constraints

## Beers

<u>Name</u>	Brand
B1	Tiger

## Sells

<u>Bar</u>	<u>Beer</u>	Price
Lotus	B1	10

- Sells(Beer) is a foreign key referencing Beer(Name)
- INSERT INTO Sells VALUES( 'Lotus', B2, 20)
- This insertion will be rejected

**INSERT** or **UPDATE** a **Sells** tuple so it refers to a nonexistent beer

Always rejected

# Foreign Key Constraints (cont.)

## Beers

<u>Name</u>	Brand
B1	Tiger

## Sells

<u>Bar</u>	<u>Beer</u>	Price
Lotus	B1	10

- Sells(Beer) is a foreign key referencing Beer(Name)
- DELETE FROM Beers  
WHERE Name = 'B1'
- What is going to happen?
- Three possibilities
  - Default: Reject the deletion
  - Cascade: Delete the corresponding tuple in Sells
  - Set NULL: Set the corresponding values to NULL

# Default

Beers	<u>Name</u>	Brand
	B1	Tiger

## Sells

<u>Bar</u>	<u>Beer</u>	Price
Lotus	B1	10

- Sells(Beer) is a foreign key referencing Beer(Name)
- DELETE FROM Beers  
WHERE Name = 'B1' Rject!
- UPDATE Beers  
SET Name = 'B2'  
Where Name = 'B1' Rject!

# Cascade

Beers	<u>Name</u>	Brand
	B1	Tiger

Sells		
<u>Bar</u>	<u>Beer</u>	Price
Lotus	B1	10

- Sells(Beer) is a foreign key referencing Beer(Name)
- DELETE FROM Beers  
WHERE Name = 'B1'
- Delete **all tuples from sells where Beer = 'B1'**
- UPDATE Beers  
SET Name = 'B2'  
Where Name = 'B1'
- Update Sells: for any tuple with Beer = 'B1', set its Beer value to 'B2'

# Set NULL

Beers	<u>Name</u>	Brand
	B1	Tiger

Sells		
<u>Bar</u>	<u>Beer</u>	Price
Lotus	B1	10

- Sells(Beer) is a foreign key referencing Beer(Name)
- DELETE FROM Beers  
WHERE Name = 'B1'
- Change all 'B1' Beer values in Sells to NULL
- UPDATE Beers  
SET Name = 'B2'  
Where Name = 'B1'
- Change all 'B1' Beer values in Sells to NULL



# Choosing a Policy

Sells

Beers

<u>Name</u>	Brand
B1	Tiger

<u>Bar</u>	<u>Beer</u>	Price
Lotus	B1	10

- We can specify whether to default, cascade, or set null when we declare a foreign key
- Example:  

```
CREATE TABLE Sells(  
  Bar      VARCHAR(30),  
  Beer     VARCHAR(30),  
  Price    FLOAT,  
  FOREIGN KEY (Beer) REFERENCES Beers(Name)  
  ON DELETE SET NULL  
  ON UPDATE CASCADE );
```
- Default is taken when SET NULL and CASCADE are absent

# Summary and roadmap

- Introduction to SQL
- SELECT  
FROM  
WHERE
- Eliminating duplicates
- Renaming attributes
- Expressions in SELECT Clause
- Patterns for Strings
- Ordering
- Joins
- Subquery
- Aggregations
- UNION, INTERSECT, EXCEPT
- NULL
- Outerjoin
- Insert/Delete tuples
- Create/Alter/Delete tables
- Constraints: primary key
- Views
- Constraints: Foreign key

## • Next

- More constraints
  - CHECK
  - ASSERTION
  - Trigger
- Indexes

# attribute-based check

We want to make sure that any beer does not sell over 100 dollars. How?

```
CREATE TABLE      Sells (  
    bar            CHAR(20),  
    beer           CHAR(30) ,  
    price          REAL      CHECK (price <= 100),  
);
```

- **CHECK (<condition>)** must be added to the declaration for the attribute
- An attribute-based check is checked only when a value for that attribute is **inserted** or **updated** (but not deleted)

# Tuple-Based Check

- **CHECK (<condition>)** may be added as separate element in a table declaration

- The condition can refer to any attribute of the relation,

Example: Only Lotus can sell beer for more than \$10

```
CREATE TABLE    Sells (  
    bar          CHAR(20),  
    beer         CHAR(30),  
    price        REAL,  
    CHECK (bar = 'Lotus' OR price <= 10.00)  
);
```

- Checked whenever a tuple is **inserted** or **updated**

# SQL Assertions

## What is it?

- Database schema constraints
  - Not available with major DBMS, but in SQL standard
- Condition can refer to any relation or attribute in the database schema
- We must check every **ASSERTION** after every modification to any relation of the database

## Syntax

```
CREATE ASSERTION <name>  
CHECK (condition);
```

# Assertions

Beers	<u>Name</u>	Brand
	B1	Tiger

Sells		
<u>Bar</u>	<u>Beer</u>	Price
Lotus	B1	10

- A more flexible type of checks
  - But not available with most of DBMS
- Example: There should not be more bars than beers
- **CREATE ASSERTION** MoreBars **CHECK**  
(  
    (SELECT COUNT(\*) FROM Beers)  
    >=  
    (SELECT COUNT(DISTINCT Bar) FROM Sells)  
)

# Example

## Constraint

In **Sells(Bar, Beer, Price)** no bar may charge an average of more than \$10.

```
CREATE ASSERTION      NoRipOffBars CHECK (  
    NOT EXISTS (  
        SELECT      bar  
        FROM        Sells  
        GROUP BY    bar  
        HAVING       10.00 < AVG(price)  
    ));
```

# Motivation for Trigger

## Attributes and Tuple-based checks

Limited capabilities

## Assertions

- Sufficiently general for most constraint applications
- Hard to implement efficiently!
- We must check every **ASSERTION** after every modification to any relation of the database



# Trigger

Beers	<u>Name</u>	Brand
	B1	Tiger

Sells		
<u>Bar</u>	<u>Beer</u>	Price
Lotus	B1	10

- Every time there is a new beer from Tiger, the Lotus Bar would sell it at 10 dollars.
- How to capture this?
- we can use **triggers**
  - Triggering event
  - Action

# Trigger (**AFTER INSERT**)

Beers	Name	Brand
	B1	Tiger

Bar	Beer	Price
Lotus	B1	10

Sells

- Every time there is a new beer from Tiger, the Lotus Bar would sell it at 10 dollars.
- **CREATE TRIGGER** BeerTrig  
**AFTER INSERT ON** Beers  
**REFERENCING NEW ROW AS** NewTuple  
**FOR EACH ROW**  
**WHEN** (NewTuple.Brand = 'Tiger')  
**INSERT INTO** Sells  
VALUES ('Loctus', NewTuple.Name, 10);

# Trigger (AFTER DELETE)

Sells

Beers	Name	Brand
	B1	Tiger

Bar	Beer	Price
Lotus	B1	10

- Every time the Lotus Bar stops selling a beer, the Cheetah Bar would sell it at its last price
- CREATE TRIGGER BeerTrig  
AFTER DELETE ON Sells  
REFERENCING OLD ROW AS OldTuple  
FOR EACH ROW  
WHEN (OldTuple.Bar = 'Lotus')  
INSERT INTO Sells  
VALUES ('Cheetah', OldTuple.Beer,  
OldTuple.Price);

# Trigger (AFTER UPDATE ON) Sells

Beers	Name	Brand
	B1	Tiger

Bar	Beer	Price
Lotus	B1	10

- Every time the some bar raises the price of a beer by over 10 dollars, the Cheetah Bar would sell it at its old price
- CREATE TRIGGER BeerTrig  
AFTER UPDATE ON Sells  
REFERENCING  
    OLD ROW AS OldTuple  
    NEW ROW AS NewTuple  
FOR EACH ROW  
WHEN (OldTuple.Price + 10 < NewTuple.Price)  
INSERT INTO Sells  
VALUES ('Cheetah', OldTuple.Beer,  
OldTuple.Price);

---

duplicate beers at the Cheetah Bar. How to avoid?

# Trigger (cont.)

## Sells

Beers	Name	Brand
	B1	Tiger

Bar	Beer	Price
Lotus	B1	10

- Every time the some bar raises the price of a beer by over 10 dollars, the Cheetah Bar would sell it at its old price. (Note: Avoid duplicate beers at the CheeTah Bar)
- CREATE TRIGGER BeerTrig  
AFTER UPDATE ON Sells  
REFERENCING  
    OLD ROW AS OldTuple  
    NEW ROW AS NewTuple  
FOR EACH ROW  
WHEN (OldTuple.Price + 10 < NewTuple.Price)  
BEGIN  
    DELETE FROM Sells  
    WHERE Bar = 'Cheetah' AND Beer = OldTuple.Beer;  
    INSERT INTO Sells  
        VALUES ('CheeTah', OldTuple.Beer, OldTuple.Price);  
END

# A summary of Trigger

- A triggering event can be the execution of an SQL **INSERT, DELETE, and UPDATE** statement
- **WHEN** clause is optional
  - If it is missing, then the action is executed whenever the trigger is awakened
- If the When condition of the trigger is satisfied, the action associated with the trigger is performed by the DBMS

## Types of actions

- An SQL query, a DELETE, INSERT, UPDATE, ROLLBACK, SQL/PSM
- There can be **more than one** SQL statements
- Queries make no sense in an action
  - So only DB modification

# Exercise (1)

Beers	Name	Brand
	B1	Tiger

Sells		
Bar	Beer	Price
Lotus	B1	10

- Every time the Lotus Bar sells a new bear, the Cheetah Bar would sell it with 1 dollars less
- CREATE TRIGGER BeerTrig  
AFTER INSERT ON Sells  
REFERENCING  
    NEW ROW AS NewTuple  
FOR EACH ROW  
WHEN (NewTuple.Bar = 'Lotus')  
BEGIN  
    DELETE FROM Sells  
    WHERE Bar = 'Cheetah' AND Beer = NewTuple.Beer;  
    INSERT INTO Sells  
        VALUES ('CheeTah', NewTuple.Beer,  
                NewTuple.Price - 1);  
END

# Exercise (2)

Beers	Name	Brand
	B1	Tiger

Sells		
Bar	Beer	Price
Lotus	B1	10

- Every time the Lotus Bar cut the price of a beer by  $x$  dollars, the Cheetah Bar would cut the price by  $2 * x$  dollars
- CREATE TRIGGER BeerTrig  
AFTER UPDATE ON Sells  
REFERENCING  
    OLD ROW AS OldTuple  
    NEW ROW AS NewTuple  
FOR EACH ROW  
WHEN (OldTuple.Price > NewTuple.Price AND OldTuple.Bar = 'Lotus')  
BEGIN  
    UPDATE Sells  
    SET Price = Price - 2 \* (OldTuple.Price - NewTuple.Price)  
    WHERE Bar = 'Cheetah' AND Beer = OldTuple.Beer;  
END



# Exercise (3)

## Sells

Beers	Name	Brand
	B1	Tiger

Bar	Beer	Price
Lotus	B1	10

- Every time the Lotus Bar stops selling a beer, if the Cheetah bar sells the beer, it would set the price of the beer to the average price of the Tiger beers sold at the Lotus Bar
- CREATE TRIGGER BeerTrig  
AFTER DELETE ON Sells  
REFERENCING  
    OLD ROW AS OldTuple  
FOR EACH ROW  
WHEN (OldTuple.Bar = 'Lotus')  
BEGIN  
    UPDATE Sells  
    SET Price = (SELECT AVG(Price)  
                  FROM Beers, Sells  
                  WHERE Bar = 'Lotus' AND Brand = Tiger and  
                        name = beer)  
    WHERE Bar = 'Cheetah' AND Beer = OldTuple.Beer;  
END

# Question

- All customers must have an account balance of at least \$1000 in their account.
  - Do you use Check or Trigger ?
- All new customers opening an account must have a balance of \$1000
  - Do you use Check or Trigger ?

# Other syntax of triggers

- Before { [ INSERT ] [ , ] [ UPDATE ] [ , ] [ DELETE ] }
  - WHEN condition is tested before the triggering event is executed
- Instead of { [ INSERT ] [ , ] [ UPDATE ] [ , ] [ DELETE ] }
  - It overrides Insert, Update, Delete
  - Mostly used for updating a view---to update base tables of view

# BEFORE - Trigger Execution (Example)

**Employee**

First_Name	Last_Name	Phone
...	...	...

```
CREATE TRIGGER    Last_Name_Upper
BEFORE INSERT ON Employee
REFERENCING NEW ROW AS N
FOR EACH ROW
    N.Last_Name = Upper(N.Last_Name);
```

- No When Clause. The trigger action is executed.
- Then the triggering event ( Insert N into Employee) is executed

# BEFORE - Trigger Execution

1. **WHEN** condition is tested **before** the triggering event
  - If the condition is true then the action of the trigger is executed
2. Next, the **triggering event** is executed, regardless of whether the condition is true

## ■ Action:

- update or validate record values (the same record in the triggering event) before they are saved to the database
- Not allowed to modify the database

# INSTEAD OF - Trigger Execution (Example)

**Likes**

Drinker	Beer
John	A1

**Frequent**

Drinker	Bar
John	B2

**Sells**

Bar	Beer	Price
Lotus	B1	10

CREATE VIEW  
 SELECT  
 FROM  
 WHERE

*Synergy AS*

*Likes.drinker, Likes.beer, Sells.bar*

*Likes, Sells, Frequent*

*Likes.drinker = Frequent.drinker AND*

*Likes.beer = Sells.beer AND*

*Sells.bar = Frequent.bar*

view *Synergy* has (drinker, beer, bar) triples: the bar serves the beer and the drinker frequents the bar and likes the beer

Use a **INSTEAD OF** trigger to turn a (drinker, beer, bar) triple into three insertions of projected pairs

# INSTEAD OF - Trigger Execution (Example)

```
CREATE TRIGGER                                ViewTrig
INSTEAD OF INSERT ON                          Synergy
REFERENCING
        NEW ROW AS                            n
FOR EACH ROW
BEGIN
    INSERT INTO Likes                          VALUES(n.drinker, n.beer);
    INSERT INTO Sells(bar,beer)                VALUES(n.bar, n.beer);
    INSERT INTO Frequents                     VALUES(n.drinker, n.bar);
END;
```

- Semantic: Trigger defined on view. instead of triggering event, we implement action
- Generally it is impossible to modify views because it doesn't exist physically.

# Trigger - Syntax

```
CREATE TRIGGER triggerName
{BEFORE | AFTER | INSTEAD OF}
  {INSERT | DELETE | UPDATE [OF column-name-list]}
ON table-name
  [ REFERRING [ OLD ROW AS var-to-refer-to-old-tuple
               [ NEW ROW AS var-to-refer-to-new-tuple]
               [ OLD TABLE AS name-to-refer-to-new-table]
               [ NEW TABLE AS name-to-refer-to-old-table]
  [ FOR EACH { ROW | STATEMENT } ]
  [ WHEN (precondition) ]
[BEGIN]
statement-list;
[END];
```

**Granularity**



# Trigger Execution Granularity (Example)

```
CREATE TRIGGER  
AFTER INSERT ON  
REFERENCING NEW ROW AS  
FOR EACH ROW  
WHEN (newTuple.beer  
  
INSERT INTO  
VALUES (newTuple.beer);
```

```
BeerTrig  
Sells  
newTuple  
  
NOT IN  
(SELECT name FROM Beers))  
Beers(name)
```

```
CREATE TRIGGER  
AFTER UPDATE OF  
FOR EACH STATEMENT  
INSERT INTO  
VALUES (Current_Date, SELECT AVG(Salary) FROM Employee))
```

```
RecordNewAvg  
Salary ON Employee  
  
Log
```

# Trigger Execution Granularity

## Row level

- **FOR EACH ROW** indicates **row-level**
- **Row-level** triggers are executed **once for each modified (inserted, updated, or deleted) tuple/ row**

## Statement level

- Executed **once for an SQL statement**, regardless of the number of tuples modified
  - even if 0 tuple is modified: An **UPDATE** statement that makes no changes (condition in the **WHERE** clause does not affect any tuples)
- **If neither is specified, default is “Statement level”**

# Trigger - Syntax

```
CREATE TRIGGER triggerName
{BEFORE | AFTER| INSTEAD OF}
  {INSERT | DELETE | UPDATE [OF column-name-list]}
ON table-name
```

## Referencing

```
[ REFERENCING [ OLD ROW AS var-to-refer-to-old-tuple ]
               [ NEW ROW AS var-to-refer-to-new-tuple]
               [ OLD TABLE AS name-to-refer-to-new-table]
               [ NEW TABLE AS name-to-refer-to-old-table]
```

```
[ FOR EACH { ROW | STATEMENT } ]
[ WHEN (precondition) ]
[BEGIN]
statement-list;
[END];
```

# Trigger Referencing a table

```
CREATE TRIGGER FLIGHTS_DELETE
AFTER DELETE ON FLIGHTS
REFERENCING OLD TABLE AS DELETED_FLIGHTS
FOR EACH STATEMENT
BEGIN
DELETE FROM FLIGHT_AVAILABILITY
WHERE FLIGHT_ID IN
(SELECT FLIGHT_ID
FROM DELETED_FLIGHTS);
END
```

- The **OLD TABLE** maps those rows affected by the triggering event (i.e., only deleted rows of FLIGHT due to the execution of the triggering SQL statement).

**❑ NOT** the old version of FLIGHT table before deletion

# Summary and roadmap

- Introduction to SQL
- SELECT  
FROM  
WHERE
- Eliminating duplicates
- Renaming attributes
- Expressions in SELECT Clause
- Patterns for Strings
- Ordering
- Joins
- Subquery
- Aggregations
- UNION, INTERSECT, EXCEPT
- NULL
- Outerjoin
- Insert/Delete tuples
- Create/Alter/Delete tables
- Constraints: primary key
- Views
- Constraints:
  - Foreign key
  - CHECK
  - ASSERTION
  - Trigger
- Next
  - Indexes