# Views

Makes development **simpler**
Often used for **security**
**Not instantiated** - makes updates tricky

CREATE VIEW *view_name*
AS *select_statement*

```
CREATE VIEW Reds
AS SELECT  B.bid,  COUNT (*) AS scount
    FROM Boats B, Reserves R
    WHERE  R.bid=B.bid AND   B.color='red'
     GROUP BY  B.bid
```

# An illustration

```
CREATE VIEW Reds
AS SELECT  B.bid,  COUNT (*) AS scount
    FROM Boats B, Reserves R
    WHERE  R.bid=B.bid AND   B.color='red'
     GROUP BY  B.bid
```

| bid | bname | color |
|---|---|---|
| 101 | Interlake | blue |
| 102 | Interlake | red |
| 103 | Clipper | green |
| 104 | Marine | red |

| b.bid | scount |
|---|---|
| 102 | 1 |

Reds

# Views Instead of Relations in Queries

```
CREATE VIEW Reds
AS SELECT  B.bid,  COUNT (*) AS scount
    FROM Boats B, Reserves R
    WHERE  R.bid=B.bid AND   B.color='red'
     GROUP BY  B.bid
```

```
SELECT  bname, scount
    FROM Reds R, Boats B
    WHERE  R.bid=B.bid
        AND scount < 10
```

| b.bid | scount |
|---|---|
| 102 | 1 |

Reds

# Views vs INTO

(1)
```
SELECT bname, bcity
FROM   branch
INTO   branch2
```

vs

(2)
```
CREATE VIEW branch2 AS
SELECT  bname, bcity
FROM    branch
```

(1) creates a new table that gets stored on disk

(2) creates a “virtual table” (materialized when needed)

Therefore: **changes** in branch are **seen** in (2) but **not** in (1)

Assertions and Triggers

# CONSTRAINTS

# Integrity Constraints

- predicates on the database
- must always be true (checked whenever db gets updated)

There are the following 4 types of IC's:

**Key** constraints (1 table)

e.g., 2 accts can't share the same acct_no

**Attribute** constraints (1 table)

e.g., 2 accts must have nonnegative balance

**Referential Integrity** constraints ( 2 tables)

E.g. bnames associated w/ loans must be names of real branches

**Global Constraints** (n tables)

E.g., a loan must be carried by at least 1 customer with a svngs acct

# Global Constraints

Idea: two kinds

1) **single relation** (constraints spans multiple columns)

   E.g.: CHECK (total = svngs + check) declared in the CREATE TABLE

2) **multiple relations**: CREATE ASSERTION

SQL examples:

1) **single relation**: All BOSTON branches must have assets > 5M

```
CREATE TABLE branch (
        .....
        bcity  CHAR(15),
        assets INT,
        CHECK (NOT(bcity = 'BOS') OR assets > 5M))
```

**Affects**:

**insertions** into branch

**updates** of bcity or **assets** in branch

# Global Constraints

SQL example:

2) **Multiple relations**: every loan has a borrower with a savings account

```
CHECK (NOT EXISTS (
                SELECT   *
                FROM      loan AS L
                WHERE  NOT EXISTS(
                          SELECT   *
                          FROM borrower B, depositor D, account A
                          WHERE B.cname = D.cname  AND
                                    D.acct_no = A.acct_no  AND  L.lno  = B.lno)))
```

Problem: Where to put this constraint? At depositor? Loan? ....

Ans: None of the above:

```
CREATE ASSERTION loan-constraint
    CHECK(  ..... )
```

Checked with EVERY DB update!
very expensive.....

# Global Constraints

Issues:

1) How does one decide what global constraint to impose?
2) How does one minimize the cost of checking the global constraints?

Ans: Semantics of application and Functional dependencies.

# Summary: Integrity Constraints

| Constraint Type | Where declared | Affects... | Expense |
|---|---|---|---|
| Key Constraints | CREATE TABLE<br>(PRIMARY KEY, UNIQUE) | Insertions, Updates | Moderate |
| Attribute Constraints | CREATE TABLE<br>CREATE DOMAIN<br>(Not NULL, CHECK) | Insertions, Updates | Cheap |
| Referential Integrity | Table Tag<br>(FOREIGN KEY ....<br>REFERENCES ....) | 1.Insertions into referencing rel'n<br>2. Updates of referencing rel'n of relevant attrs<br>3. Deletions from referenced rel'n<br>4. Update of referenced rel'n | 1,2: like key constraints. Another reason to index/sort on the primary keys<br>3,4: depends on<br>a. update/delete policy chosen<br>b. existence of indexes on foreign key |
| Global Constraints | Table Tag (CHECK)<br>or<br>outside table<br>(CREATE ASSERTION) | 1. For single rel'n constraint, with insertion, deletion of relevant attrs<br>2. For assesrtions w/ every db modification | 1. cheap<br>2. very expensive |

# Triggers (Active database)

- Trigger: A procedure that starts automatically if specified changes occur to the DBMS
- Analog to a "daemon" that monitors a database for certain events to occur
- Three parts:
  - Event (activates the trigger)
  - Condition (tests whether the triggers should run) [Optional]
  - Action (what happens if the trigger runs)

- Semantics:
  - When event occurs, and condition is satisfied, the action is performed.

# An example of Trigger

```
CREATE TRIGGER minSalary BEFORE INSERT ON Professor
FOR EACH ROW
WHEN (new.salary < 100,000)
BEGIN
    RAISE_APPLICATION_ERROR (-20004, 'Violation of Minimum Professor Salary');
END;
```

Conditions can refer to **old/new** values of tuples modified by the statement activating the trigger.

# Triggers – Event, Condition, Action

Events could be :

```
BEFORE|AFTER INSERT|UPDATE|DELETE ON <tableName>
```

e.g.: `BEFORE INSERT ON Professor`

Condition is SQL expression or even an SQL query (query with non-empty result means TRUE)

Action can be many different choices :

- SQL statements, and even DDL and transaction-oriented statements like "commit".

# Example Trigger

Assume our DB has a relation schema :

Professor (pNum, pName, salary)

We want to write a trigger that :

Ensures that any new professor inserted has salary >= 70000

# Example Trigger

```
CREATE TRIGGER minSalary BEFORE INSERT ON Professor

     for what context  ?

BEGIN

    check for violation here ?

END;
```

# Example Trigger

```
CREATE TRIGGER minSalary BEFORE INSERT ON Professor
     FOR EACH ROW
BEGIN
    check for violation here ?
END;
```

# Example Trigger

```
CREATE TRIGGER minSalary BEFORE INSERT ON Professor

      FOR EACH ROW

BEGIN

  IF (:new.salary < 70000)
     THEN RAISE_APPLICATION_ERROR (-20004,
     'Violation of Minimum Professor Salary');
  END IF;

END;
```

# Details of Trigger Example

BEFORE INSERT ON Professor

- This trigger is checked before the tuple is inserted

FOR EACH ROW

- specifies that trigger is performed for each row inserted

:new

- refers to the new tuple inserted

If (:new.salary < 70000)

- then an application error is raised and hence the row is not inserted; otherwise the row is inserted.

Use error code: -20004;

- this is in the valid range

# Example Trigger Using Condition

```
CREATE TRIGGER minSalary BEFORE INSERT ON Professor
   FOR EACH ROW
WHEN (new.salary < 70000)
BEGIN
   RAISE_APPLICATION_ERROR (-20004,
   'Violation of Minimum Professor Salary');
END;
```

Conditions can refer to **old/new** values of tuples modified by the statement activating the trigger.

# Triggers: REFERENCING

```
CREATE TRIGGER minSalary BEFORE INSERT ON Professor

REFERENCING NEW as newTuple

FOR EACH ROW

WHEN (newTuple.salary < 70000)

BEGIN
   RAISE_APPLICATION_ERROR (-20004,
   'Violation of Minimum Professor Salary');
END;
```

# Example Trigger

```
CREATE TRIGGER updSalary
     BEFORE UPDATE ON Professor
REFERENCING OLD AS oldTuple NEW as newTuple
FOR EACH ROW
WHEN (newTuple.salary < oldTuple.salary)
BEGIN
   RAISE_APPLICATION_ERROR (-20004, 'Salary
   Decreasing !!');
END;
```

Ensure that salary does not decrease

# Another Trigger Example (SQL:99)

```
CREATE TRIGGER  youngSailorUpdate
   AFTER  INSERT ON SAILORS
REFERENCING NEW TABLE AS NewSailors
FOR EACH STATEMENT
   INSERT
        INTO YoungSailors(sid, name, age, rating)
        SELECT sid, name, age, rating
        FROM NewSailors N
        WHERE N.age <= 18
```

# Row vs Statement Level Trigger

- Row level:  activated once per modified tuple
- Statement level: activate once per SQL statement

- Row level triggers can access new data, statement level triggers cannot always do that (depends on DBMS).

- Statement level triggers will be more efficient if we do not need to make row-specific decisions

# Row vs Statement Level Trigger

Example: Consider a relation schema

Account (num, amount)

where we will allow creation of new accounts only during normal business hours.

# Example: Statement level trigger

```
CREATE TRIGGER MYTRIG1
BEFORE INSERT ON Account
FOR EACH STATEMENT                  --- is default
BEGIN
   IF (TO_CHAR(SYSDATE,'dy') IN ('sat','sun'))
   OR
   (TO_CHAR(SYSDATE,'hh24:mi') NOT BETWEEN '08:00' AND '17:00')
    THEN
      RAISE_APPLICATION_ERROR(-20500,'Cannot   create new account now !!');
   END IF;
END;
```

# When to use BEFORE/AFTER

Based on efficiency considerations or semantics.

Suppose we perform statement-level after insert,

➔all the rows are inserted first,

➔if the condition fails ➔ all inserts must be "rolled back"

Not very efficient !!

# Combining multiple events into one trigger

```
CREATE TRIGGER salaryRestrictions
AFTER INSERT OR UPDATE ON Professor
FOR EACH ROW
BEGIN
IF (INSERTING AND :new.salary < 70000) THEN
  RAISE_APPLICATION_ERROR (-20004, 'below min salary');
END IF;
IF (UPDATING AND :new.salary < :old.salary) THEN
  RAISE_APPLICATION_ERROR (-20004, 'Salary Decreasing !!');
END IF;
END;
```

# Summary : Trigger Syntax

```
CREATE TRIGGER <triggerName>
BEFORE|AFTER   INSERT|DELETE|UPDATE
   [OF <columnList>] ON <tableName>|<viewName>
   [REFERENCING [OLD AS <oldName>] [NEW AS <newName>]]
[FOR EACH ROW] (default is “FOR EACH STATEMENT”)
[WHEN (<condition>)]
<PSM body>;
```

# Constraints versus Triggers

- Constraints are useful for database consistency
  - Use IC when sufficient
  - More opportunity for optimization
  - Not restricted into insert/delete/update

- Triggers are flexible and powerful
  - Alerters
  - Event logging for auditing
  - Security enforcement
  - Analysis of table accesses (statistics)
  - Workflow and business intelligence …

But can be **hard** to understand ……
- Several triggers (Arbitrary order → unpredictable!)
- Chain triggers (When to stop ?)
- Recursive triggers (Termination?)

# Links for Examples

Schema is available at:
https://gist.github.com/manathan1984/35b189ae92fd996cce7816e2d7f9e40f

Lightweight online SQL frontend:
http://sqlfiddle.com/

# CS660 Fall 2024

## Lab 2: SQL

BOSTON UNIVERSITY

DiSC lab

# The Movies Database

MotionPicture (**id**, name, rating, production, budget)

User (**email**, name, age)

Likes (**uemail, mpid**)

Movie (**mpid**, boxoffice_collection)

Series (**mpid**, season_count)

People (**id**, name, nationality, dob, gender)

Role (**mpid, pid**, role_name)

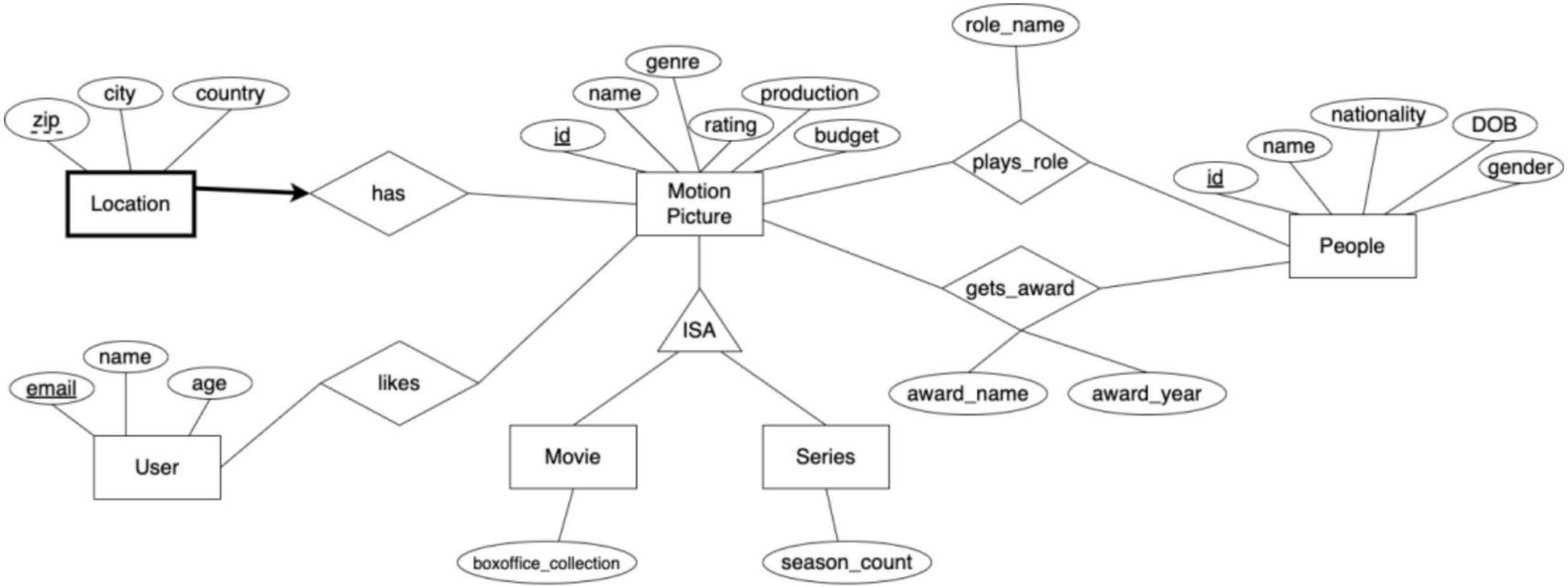
Award (**mpid**, **pid, award_name**, **award_year**)

Genre (**mpid**, genre_name)

Location (**mpid**, **zip**, city, country)

Primary keys are **underlined** and foreign keys are in **blue**

# ER Diagram



# Exercises

**Q1. List all directors who have directed a TV series at a specific zip code "02215".**

*You should list the director's name and TV series name only, without any duplicates.*

**Q2. List the people who have played multiple roles in a motion picture whose rating is more than 8.0.**

*You should list the person's name, motion picture name, and count of roles for that motion picture.*

**Q3. Find the actors who share the same birthday.**

*List the actors' names and their common birthday.*

# Movies Database

MotionPicture (**id**, name, rating, production, budget)

User (**email**, name, age)

Likes (**uemail, mpid**)

Movie (**mpid**, boxoffice_collection)

Series (**mpid**, season_count)

People (**id**, name, nationality, dob, gender)

Role (**mpid, pid**, role_name)

Award (**mpid**, **pid**, **award_name**, **award_year**)

Genre (**mpid**, genre_name)

Location (**mpid**, **zip**, city, country)

# Exercise 1

**List all directors who have directed a TV series at a specific zip code “02215”.**

*You should list the director's name and TV series name only, without any duplicates.*

# Exercise 1

**List all directors who have directed a TV series at a specific zip code “02215”.**

*You should list the director's name and TV series name only, without any duplicates.*

```
SELECT DISTINCT P.name, M.name
FROM Location L, MotionPicture M,
Role R, People P, Series S
WHERE L.mpid = M.id
AND M.id = R.mpid
AND R.pid = P.id
AND M.id = S.mpid
AND R.role_name = 'director'
AND L.zipcode = '02215'
```

# Exercise 2

**List the people who have played multiple roles in a motion picture whose rating is more than 8.0.**

*You should list the person's name, motion picture name, and count of roles for that motion picture.*

# Exercise 2

**List the people who have played multiple roles in a motion picture whose rating is more than 8.0.**

*You should list the person's name, motion picture name, and count of roles for that motion picture.*

```
SELECT P.name, M.name, COUNT(*)
FROM MotionPicture M, Role R,
People P
WHERE M.id = R.mpid
AND P.id = R.pid
AND M.rating > 8.0
GROUP BY R.mpid, R.pid, P.name,
M.name HAVING COUNT(*) > 1
```

# Exercise 3

**Find the actors who share the same birthday.**

*List the actors' names and their common birthday.*

# Exercise 3

**Find the actors who share the same birthday.**

*List the actors' names and their common birthday.*

```
SELECT P1.name, P2.name FROM
      (SELECT P.id, P.name, P.dob
            FROM People P, Role R
            WHERE P.id = R.pid
            AND
            R.role_name='Actor') P1
 INNER JOIN
      (SELECT P.id, P.name, P.dob
            FROM People P, Role R
            WHERE P.id = R.pid AND
            R.role_name='Actor') P2
ON P1.dob=P2.dob
WHERE P1.id > P2.id;
```