DATABASE VIEWS

1

VIEWS FOR CUSTOMIZATION

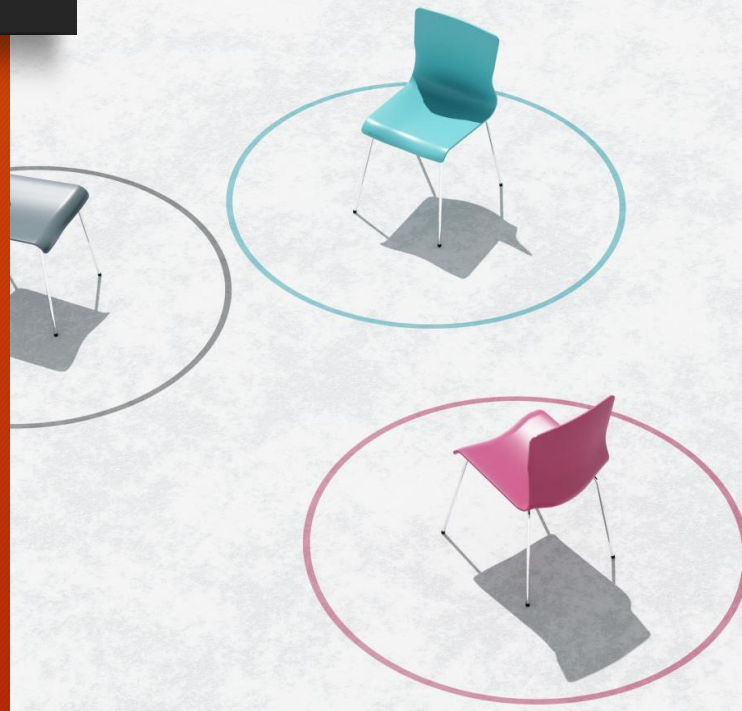
- Consider database(s) describing university's activities
  - Academic institution
    - Students, professors, classes
    - Grades, transcripts
    - Admissions, convocations
    - Alumni
  - Corporate institution
    - Finances, human resources
    - Board of Governors
    - Capital assets
  - Charitable institution
    - Donors, fundraising activities
  - Research institution
    - Granting agencies, industrial/non-profits/academic partners
    - Grants and contracts, intellectual property, licensing
- Each user group provided appropriate "subset" of the data
  - e.g., some financial/scheduling info relevant to most groups; other info confidential
  - Underlying data *shared, not silo'd.*
- Updates must be seen by all affected users.

VIEWS (VIRTUAL TABLES)

- Consider again the query

```
SELECT title, year, genre
FROM Film
WHERE director = 'Steven
Spielberg' AND year > 1990;
```

  - Returns all matching films currently in the database
  - If re-run after updates, will give revised table of matches
- A **view** is an *unexecuted query* that can be run on demand.
  - Single table derived from other table(s)
  - A virtual table



USING VIEWS IN SQL

- **`CREATE VIEW`** command
  - View name and a query to specify the contents of the view

```
CREATE VIEW Big_Earners AS
    SELECT E.Ssn AS Ssn, E.Lname AS Name,
           E.Salary AS Salary, M.Lname AS Manager
    FROM EMPLOYEE E, EMPLOYEE M
    WHERE E.Super_ssn = M.Ssn and E.Salary > M.Salary;
```

- Queries can use view as if it were a base table.

```
SELECT *
FROM Big_Earners
WHERE Salary < 100000;
```

- View always up-to-date
  - (Re-)evaluated whenever a query uses the view
  - Keeping it up-to-date is responsibility of the DBMS and not the user
- **`DROP VIEW`** command
  - Dispose of a view

UPDATING A VIEW

- What if an update is applied to a view as if it were a base table?

```
CREATE VIEW Big_Earners AS
   SELECT E.Ssn AS Ssn, E.Lname AS Name,
          E.Salary AS Salary, M.Lname AS Manager
   FROM EMPLOYEE E, EMPLOYEE M
   WHERE E.Super_ssn = M.Ssn and E.Salary > M.Salary;

UPDATE Big_Earners
SET Salary = 100000
WHERE Name = 'Smith';
```

  - Change corresponding tuple(s) in base table(s)
  - Tuple might disappear from view!
    - **`WITH CHECK OPTION`** clause at end of view definition ensures new and updated tuples match view definition (else error)
  - Deleting tuple from view might require update to base table instead of deletion from base table
    - e.g., deletion from CS338 view $\stackrel{?}{=}$ deletion from UW database?
  - Not all views are updateable.
    - What if `Salary` defined as sum of two base attributes or as aggregate such as `SUM` or `AVG`?
    - What if `Big_Earners` defined as a `UNION` of two tables?

MATERIALIZED VIEWS

- If the base tables don't change, neither does the view instance.
  - Re-executing view definition each time view is used is wasteful if base data has not been updated.
- Solution: **view materialization**
  - Create a temporary view table when the view is first queried
  - Keep view table on the assumption that more queries using the view will follow
  - Use *materialized* view (if it exists) to answer future query
  - Requires efficient strategy for *automatically updating view table* when the base tables are updated

    Options when any base table is updated:

    1. Delete the materialized view
    2. Rematerialize the view
    3. Incrementally update the view
       - DBMS determines what new tuples must be inserted, deleted, or modified in materialized view

TRIGGERS

TRIGGERS

A database trigger is a stored PL/SQL program unit associated with a specific database table. DBMS executes (fires) a database trigger automatically when a given SQL operation (like INSERT, UPDATE or DELETE) affects the table. Unlike a procedure, or a function, which must be invoked explicitly, database triggers are invoked implicitly.

TRIGGERS

Database triggers can be used to perform any of the following:

- Audit data modification
- Log events transparently
- Enforce complex business rules
- Derive column values automatically
- Implement complex security authorizations
- Maintain replicate tables

TRIGGERS

- You can associate up to 12 database triggers with a given table. A database trigger has three parts: a **triggering event**, an **optional trigger constraint**, and a **trigger action**.
- When an event occurs, a database trigger is fired, and an predefined PL/SQL block will perform the necessary action.

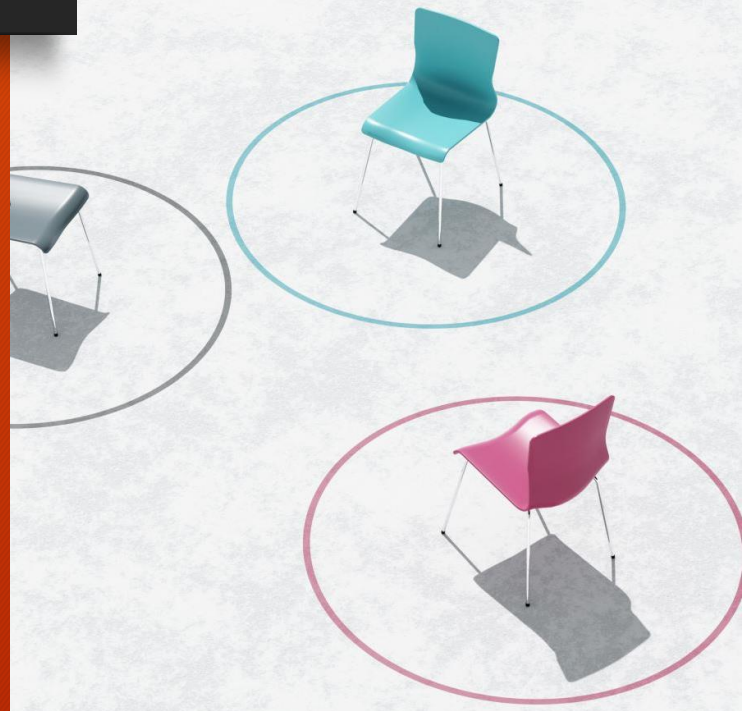
TRIGGERS

SYNTAX:

```
CREATE [OR REPLACE] TRIGGER trigger_name
{BEFORE|AFTER} triggering_event ON
      table_name
[FOR EACH ROW]
[WHEN condition]
DECLARE
Declaration statements
BEGIN
Executable statements
EXCEPTION
Exception-handling statements
END;
```

TRIGGERS

- The trigger_name references the name of the trigger.
- BEFORE or AFTER specify when the trigger is fired (before or after the triggering event).
- The triggering_event references a DML statement issued against the table (e.g., INSERT, DELETE, UPDATE).
- The table_name is the name of the table associated with the trigger.
- The clause, FOR EACH ROW, specifies a trigger is a row trigger and fires once for each modified row.
- A WHEN clause specifies the condition for a trigger to be fired.
- Bear in mind that if you drop a table, all the associated triggers for the table are dropped as well.



TYPES OF TRIGGERS

Triggers may be called BEFORE or AFTER the following events:

INSERT, UPDATE and DELETE.

The before/after options can be used to specify when the trigger body should be fired with respect to the triggering statement. If the user indicates a BEFORE option, then Oracle fires the trigger before executing the triggering statement. On the other hand, if an AFTER is used, Oracle fires the trigger after executing the triggering statement.

TYPES OF TRIGGERS

- A trigger may be a ROW or STATEMENT type. If the statement FOR EACH ROW is present in the CREATE TRIGGER clause of a trigger, the trigger is a row trigger. A row trigger is fired for each row affected by an triggering statement.
- A statement trigger, however, is fired only once for the triggering statement, regardless of the number of rows affected by the triggering statement

TYPES OF TRIGGERS

Example: statement trigger

```
CREATE OR REPLACE TRIGGER mytrig1 BEFORE DELETE OR INSERT OR
      UPDATE ON employee
BEGIN
IF      (TO_CHAR(SYSDATE,      'day')      IN      ('sat',      'sun'))      OR
      (TO_CHAR(SYSDATE,'hh:mi') NOT BETWEEN '08:30' AND '18:30')
      THEN                  RAISE_APPLICATION_ERROR(-20500, 'table is
      secured');
END IF;
END;
/
```

The above example shows a trigger that limits the DML actions to the employee table to weekdays from 8.30am to 6.30pm. If a user tries to insert/update/delete a row in the EMPLOYEE table, a warning message will be prompted.

Example: ROW Trigger

```
CREATE OR REPLACE TRIGGER mytrig2
AFTER DELETE OR INSERT OR UPDATE ON employee
FOR EACH ROW
BEGIN
IF DELETING THEN
INSERT INTO xemployee (emp_ssn, emp_last_name,emp_first_name, deldate)
VALUES (:old.emp_ssn, :old.emp_last_name,:old.emp_first_name, sysdate);
ELSIF INSERTING THEN
 INSERT INTO nemployee (emp_ssn, emp_last_name,emp_first_name, adddate)
VALUES (:new.emp_ssn, :new.emp_last_name,:new.emp_first_name, sysdate);
 ELSIF UPDATING('emp_salary') THEN
INSERT INTO cemployee (emp_ssn, oldsalary, newsalary, up_date)
VALUES (:old.emp_ssn,:old.emp_salary, :new.emp_salary, sysdate);     ELSE
INSERT INTO uemployee (emp_ssn, emp_address, up_date)
VALUES (:old.emp_ssn, :new.emp_address, sysdate);
END IF;
END;
/
```

TYPES OF TRIGGERS

Example: ROW Trigger

- The previous trigger is used to keep track of all the transactions performed on the employee table. If any employee is deleted, a new row containing the details of this employee is stored in a table called xemployee. Similarly, if a new employee is inserted, a new row is created in another table called nemployee, and so on.
- Note that we can specify the old and new values of an updated row by prefixing the column names with the :OLD and :NEW qualifiers.

TYPES OF TRIGGERS

```
SQL>  DELETE FROM  employee WHERE emp_last_name =
      'Joshi';
1 row deleted.
SQL> SELECT * FROM xemployee;
```

EMP_SSN	EMP_LAST_NAME	EMP_FIRST_NAME	DELDATE
999333333	Ali	Khan	02-MAY-24

ENABLING, DISABLING, DROPPING TRIGGERS

SQL>ALTER TRIGGER trigger_name DISABLE;

SQL>ALTER TABLE table_name DISABLE ALL TRIGGERS;

To enable a trigger, which is disabled, we can use the following syntax:

SQL>ALTER TABLE table_name ENABLE trigger_name;

All triggers can be enabled for a specific table by using the following command

SQL> ALTER TABLE table_name ENABLE ALL TRIGGERS;

SQL> DROP TRIGGER trigger_name

Thank you