# Chapter 4: Intermediate SQL[1]

Abu Raihan Mostofa Kamal

Professor, CSE Department
Islamic University of Technology (IUT)

September 26, 2024

[1]This is based on Textbook, its companion slide and other sources

# Chapter Outline



# Joins: 2 types

- Inner Join (also called Natural join)
- Outer Join

# Joins: 2 types

- Inner Join (also called Natural join)
- Outer Join

# Inner Join

Combines two tables in a natural way

It splits one large table into two. It removes both **Redundancy** and **Inconsistency**.

| *ID* | *name* | *salary* | *dept_name* | *building* | *budget* |
|---|---|---|---|---|---|
| 22222 | Einstein | 95000 | Physics | Watson | 70000 |
| 12121 | Wu | 90000 | Finance | Painter | 120000 |
| 32343 | El Said | 60000 | History | Painter | 50000 |
| 45565 | Katz | 75000 | Comp. Sci. | Taylor | 100000 |
| 98345 | Kim | 80000 | Elec. Eng. | Taylor | 85000 |
| 76766 | Crick | 72000 | Biology | Watson | 90000 |
| 10101 | Srinivasan | 65000 | Comp. Sci. | Taylor | 100000 |
| 58583 | Califieri | 62000 | History | Painter | 50000 |
| 83821 | Brandt | 92000 | Comp. Sci. | Taylor | 100000 |
| 15151 | Mozart | 40000 | Music | Packard | 80000 |
| 33456 | Gold | 87000 | Physics | Watson | 70000 |
| 76543 | Singh | 80000 | Finance | Painter | 120000 |

# Inner Join

## Combines two tables in a natural way

It splits one large table into two. It removes both **Redundancy** and **Inconsistency**.

| *ID* | *name* | *salary* | *dept_name* | *building* | *budget* |
|---|---|---|---|---|---|
| 22222 | Einstein | 95000 | Physics | Watson | 70000 |
| 12121 | Wu | 90000 | Finance | Painter | 120000 |
| 32343 | El Said | 60000 | History | Painter | 50000 |
| 45565 | Katz | 75000 | Comp. Sci. | Taylor | 100000 |
| 98345 | Kim | 80000 | Elec. Eng. | Taylor | 85000 |
| 76766 | Crick | 72000 | Biology | Watson | 90000 |
| 10101 | Srinivasan | 65000 | Comp. Sci. | Taylor | 100000 |
| 58583 | Califieri | 62000 | History | Painter | 50000 |
| 83821 | Brandt | 92000 | Comp. Sci. | Taylor | 100000 |
| 15151 | Mozart | 40000 | Music | Packard | 80000 |
| 33456 | Gold | 87000 | Physics | Watson | 70000 |
| 76543 | Singh | 80000 | Finance | Painter | 120000 |

# Inner Join (Cont.)

So, we will create two separate tables as such:

```
1
2
3
4
5
6
```

```
CREATE TABLE STUDENTS(
ID NUMBER PRIMARY KEY,
NAME VARCHAR2(30),
SALARY NUMBER(8,0)
);
```

```
1
2
3
4
5
6
```

```
CREATE TABLE DEPTS(
DEPT_NAME VARCHAR2(6)
PRIMARY KEY,
BUILDING VARCHAR2(20),
BUDGET NUMBER(10,0)
);
```

Is this splitting **OK**?

**No**. Since there is no linking attribute from the first table to the second one.

# Inner Join (Cont.)

So, we will create two separate tables as such:

```
1
2
3
4
5
6
```

```
CREATE TABLE STUDENTS(
ID NUMBER PRIMARY KEY,
NAME VARCHAR2(30),
SALARY NUMBER(8,0)
);
```

```
1
2
3
4
5
6
```

```
CREATE TABLE DEPTS(
DEPT_NAME VARCHAR2(6)
PRIMARY KEY,
BUILDING VARCHAR2(20),
BUDGET NUMBER(10,0)
);
```

Is this splitting **OK**?

**No**. Since there is no linking attribute from the first table to the second one.

# Inner Join (Cont.)

So, we will create two separate tables as such:

```
1  CREATE TABLE STUDENTS(
2  ID NUMBER PRIMARY KEY,
3  NAME VARCHAR2(30),
4  SALARY NUMBER(8,0)
5  );
6
```

```
1  CREATE TABLE DEPTS(
2  DEPT_NAME VARCHAR2(6)
   PRIMARY KEY,
3  BUILDING VARCHAR2(20),
4  BUDGET NUMBER(10,0)
5  );
6
```

Is this splitting **OK**?

**No**. Since there is no linking attribute from the first table to the second one.

# Inner Join (Cont.)

So, we will the right splitting with a **foreign key**:

```
1        CREATE TABLE STUDENTS(
2        ID NUMBER PRIMARY KEY,
3        NAME VARCHAR2(30),
4        SALARY NUMBER(8,0),
5        DEPT VARCHAR2(6),
6        CONSTRANTS FKSTU FOREIGN KEY
     (DEPT)
7        REFERENCING DEPTS
8        );
9
```

```
1
2        CREATE TABLE DEPTS(
3        DEPT_NAME VARCHAR2(6)
   PRIMARY KEY,
4        BUILDING VARCHAR2(20),
5        BUDGET NUMBER(10,0)
6        );
7
8
```

# Inner Joins (Cont.)

**How to retrieve values?**

Lets reproduce the large big table: Find out the name of the employees along with their ID, salary, their dept name, dept location and dept budget.

By a Natural Join

```
SELECT S.ID, S.NAME, S.SALARY, D.DEPT_NAME, D.BUILDING, D.BUDGET
FROM STUDENTS S, DEPTS D
WHERE S.DEPT=D.DEPT_NAME;
```

You can any valid condition you like in the SQL.

# Inner Joins (Cont.)

**How to retrieve values?**

Lets reproduce the large big table: Find out the name of the employees along with their ID, salary, their dept name, dept location and dept budget.

## By a Natural Join

```
1  SELECT S.ID, S.NAME, S.SALARY, D.DEPT_NAME, D.BUILDING, D.BUDGET
2  FROM STUDENTS S, DEPTS D
3  WHERE S.DEPT=D.DEPT_NAME;
4
```

You can any valid condition you like in the SQL.

# Inner Joins (Cont.)

**How to retrieve values?**

Lets reproduce the large big table: Find out the name of the employees along with their ID, salary, their dept name, dept location and dept budget.

## By a Natural Join

```
1  SELECT S.ID, S.NAME, S.SALARY, D.DEPT_NAME, D.BUILDING, D.BUDGET
2  FROM STUDENTS S, DEPTS D
3  WHERE S.DEPT=D.DEPT_NAME;
4
```

You can any valid condition you like in the SQL.

# Inner Joins (Cont.)or Natural Join Syntax

- List the names of instructors along with the course ID of the courses that they taught

```
select name, course_id
from students, takes
where student.ID = takes.ID;
```

- Same query in SQL with **natural join** construct

```
select name, course_id
from student natural join takes;
```

# Danger of Natural Join Construct

- Beware of unrelated attributes with same name which get equated incorrectly:
  In older style we write: (it is more comprehensive and used by the programmer)

```
1       select name, course_id
2       from students, takes
3       where student.ID = takes.ID;
4
```

- Same query in SQL with **natural join** construct

```
1       select name, course_id
2       from student natural join takes;
3
```

It assumes that both tables have ID in common which corresponds the same attribute (i.e. Student ID). But it may happen that one ID implies book ID!!!

# Danger of Natural Join Construct

- Beware of unrelated attributes with same name which get equated incorrectly:
  In older style we write: (it is more comprehensive and used by the programmer)

```
1       select name, course_id
2       from students, takes
3       where student.ID = takes.ID;
4
```

- Same query in SQL with **natural join** construct

```
1       select name, course_id
2       from student natural join takes;
3
```

It assumes that both tables have ID in common which corresponds the same attribute (i.e. Student ID). But it may happen that one ID implies book ID!!!

# Outer Join: Left, Right and Full

- An extension of the join operation that avoids loss of information. only matched records are selected
- Sometimes it is necessary to know both matched and non-matched records.
- The non-matched values are replaced by NULL
- This type of joining is termed as "Outer Join"
- 3 types of Outer Joins:
  1. Left outer join
  2. Right outer join
  3. Full outer join

# Outer Join: Left, Right and Full

- An extension of the join operation that avoids loss of information. only matched records are selected
- Sometimes it is necessary to know both matched and non-matched records.
- The non-matched values are replaced by NULL
- This type of joining is termed as "Outer Join"
- 3 types of Outer Joins:
  1. Left outer join
  2. Right outer join
  3. Full outer join

# Outer Join: Left, Right and Full

- An extension of the join operation that avoids loss of information. only matched records are selected
- Sometimes it is necessary to know both matched and non-matched records.
- The non-matched values are replaced by NULL
- This type of joining is termed as "Outer Join"
- 3 types of Outer Joins:
  1. Left outer join
  2. Right outer join
  3. Full outer join

# Outer Join: Left, Right and Full

- An extension of the join operation that avoids loss of information. only matched records are selected
- Sometimes it is necessary to know both matched and non-matched records.
- The non-matched values are replaced by NULL
- This type of joining is termed as "Outer Join"
- 3 types of Outer Joins:
  1. Left outer join
  2. Right outer join
  3. Full outer join

# Outer Join: Left, Right and Full

- An extension of the join operation that avoids loss of information. only matched records are selected
- Sometimes it is necessary to know both matched and non-matched records.
- The non-matched values are replaced by NULL
- This type of joining is termed as "Outer Join"
- 3 types of Outer Joins:
  1. Left outer join
  2. Right outer join
  3. Full outer join

# Outer Join: Left, Right and Full

- An extension of the join operation that avoids loss of information. only matched records are selected
- Sometimes it is necessary to know both matched and non-matched records.
- The non-matched values are replaced by NULL
- This type of joining is termed as "Outer Join"
- 3 types of Outer Joins:
  1. Left outer join
  2. Right outer join
  3. Full outer join

# Outer Join: Left, Right and Full

- An extension of the join operation that avoids loss of information. only matched records are selected
- Sometimes it is necessary to know both matched and non-matched records.
- The non-matched values are replaced by NULL
- This type of joining is termed as "Outer Join"
- 3 types of Outer Joins:
  1. Left outer join
  2. Right outer join
  3. Full outer join

# Outer Join: Left, Right and Full

- An extension of the join operation that avoids loss of information. only matched records are selected
- Sometimes it is necessary to know both matched and non-matched records.
- The non-matched values are replaced by NULL
- This type of joining is termed as "Outer Join"
- 3 types of Outer Joins:
  1. Left outer join
  2. Right outer join
  3. Full outer join

# Outer Join: Left, Right and Full (Cont.)

1. **Left outer join.** All records from the first table and only the matched part from the second one.
2. **Right outer join.** All records from the second table and only the matched part from the first one.
3. **Full outer join.** All records from both tables. (merge into one)

# Outer Join: Left, Right and Full (Cont.)

1. **Left outer join.** All records from the first table and only the matched part from the second one.
2. **Right outer join.** All records from the second table and only the matched part from the first one.
3. **Full outer join.** All records from both tables. (merge into one)

# Outer Join: Left, Right and Full (Cont.)

1. **Left outer join.** All records from the first table and only the matched part from the second one.
2. **Right outer join.** All records from the second table and only the matched part from the first one.
3. **Full outer join.** All records from both tables. (merge into one)

# Outer Join: Left, Right and Full (Cont.)

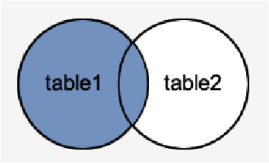



Figure: Left Outer Join

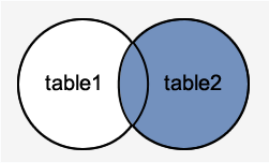



Figure: Right Outer Join

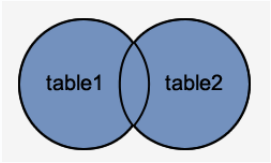



Figure: Full Outer Join

# Outer Joins Syntax

```
1   //DDLs
2   create table passports
3   ( pid number primary key ,
4   name varchar2 (10) ,
5   issued_on date ,
6   expired_on date
7   );
8
9   create table driving_lic
10  ( did number primary key ,
11  name varchar2 (10) ,
12  issued_on date ,
13  expired_on date ,
14  vehicle varchar2 (10) ,
15  passport number ,
16  constraints fk_driv foreign key ( passport ) references passports
17  );
18
```

# Outer Joins Syntax Cont.

```
1    // Left outer join: All from passport(T1)
2
3   select p . pid , p . name , p . issued_on , d . did ,
4   d . issued_on , d . vehicle
5   from passports p , driving_lic d
6   where p . pid = d . passport (+) ;
7
8   //Right outer join: All from driving lic (T2)
9
10  select p . pid , p . name , p . issued_on , d . did ,
11  d . issued_on , d . vehicle
12  from passports p , driving_lic d
13  where p . pid (+) = d . passport ;
14
```

# Outer Joins Syntax Cont.

```
1   // Full outer join: All from both (T1,T2)
2
3   select p.pid, p.name, p.issued_on, d.did, d.issued_on, d
        .vehicle
4   from passports p
5   full outer join driving_lic d
6   on p.pid = d.passport;
7
```

# Views

**What is a view?**

Normally the relations are **stored** in the database. SQL allows a **virtual relation** to be defined by a query, and the relation conceptually contains the result of the query. The virtual relation is **not precomputed and stored**, but instead is **computed by executing the query whenever the virtual relation is used**.

Functions of a view:

1. Controls **security and access control** of sensitive information.
2. Helps to **reuse the code** (its calculated on the fly, avoiding writing complex query each time)

# Views

**What is a view?**

Normally the relations are **stored** in the database. SQL allows a **virtual relation** to be defined by a query, and the relation conceptually contains the result of the query. The virtual relation is **not precomputed and stored**, but instead is **computed by executing the query whenever the virtual relation is used**.

Functions of a view:

1. Controls **security and access control** of sensitive information.
2. Helps to **reuse the code** (its calculated on the fly, avoiding writing complex query each time)

# How to create a view?

```
1
2   create view faculty as
3   select ID, name, dept name
4   from instructor;
5
6
7
```

# DML through a view

Most SQL implementations allow updates only on simple views:

- The from clause has **only one database relation**.
- The select clause contains only attribute names of the relation, and does not have any expressions, **aggregates**, or distinct specification.
- Any attribute not listed in the select clause can be **set to null**.

The condition in general:

It suggests that when a view is essentially a subset of a single table, each attribute of the view has a 1-to-1 correspondence of its underlying base table's attribute and all the not-null attributes are included in the view.

# DML through a view

Most SQL implementations allow updates only on simple views:

- The from clause has **only one database relation**.
- The select clause contains only attribute names of the relation, and does not have any expressions, **aggregates**, or distinct specification.
- Any attribute not listed in the select clause can be **set to null**.

The condition in general:

It suggests that when a view is essentially a subset of a single table, each attribute of the view has a 1-to-1 correspondence of its underlying base table's attribute and all the not-null attributes are included in the view.

# DML through a view

Most SQL implementations allow updates only on simple views:

- The from clause has **only one database relation**.
- The select clause contains only attribute names of the relation, and does not have any expressions, **aggregates**, or distinct specification.
- Any attribute not listed in the select clause can be **set to null**.

The condition in general:

It suggests that when a view is essentially a subset of a single table, each attribute of the view has a 1-to-1 correspondence of its underlying base table's attribute and all the not-null attributes are included in the view.

# DML through a view

Most SQL implementations allow updates only on simple views:

- The from clause has **only one database relation**.
- The select clause contains only attribute names of the relation, and does not have any expressions, **aggregates**, or distinct specification.
- Any attribute not listed in the select clause can be **set to null**.

The condition in general:

It suggests that when a view is essentially a subset of a single table, each attribute of the view has a 1-to-1 correspondence of its underlying base table's attribute and all the not-null attributes are included in the view.

# DML through a view

Most SQL implementations allow updates only on simple views:

- The from clause has **only one database relation**.
- The select clause contains only attribute names of the relation, and does not have any expressions, **aggregates**, or distinct specification.
- Any attribute not listed in the select clause can be **set to null**.

## The condition in general:

It suggests that when a view is essentially a subset of a single table, each attribute of the view has a 1-to-1 correspondence of its underlying base table's attribute and all the not-null attributes are included in the view.

# Materialized Views

## Definition

Database systems allow view relations to be **stored**, but they make sure that, if the actual relations used in the view definition change, the view is **kept up-to-date**. Such views are called materialized views. Materialized views are updated through an **efficient batch process**. Used in **data warehousing** technology.

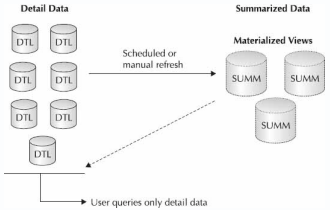



Figure: Materialized View

Benefits:

- For efficient and smooth running of the operational database (hence applications).
- Quick access time. (Local copy access is faster)

# Materialized Views

## Definition

Database systems allow view relations to be **stored**, but they make sure that, if the actual relations used in the view definition change, the view is **kept up-to-date**. Such views are called materialized views. Materialized views are updated through an **efficient batch process**. Used in **data warehousing** technology.

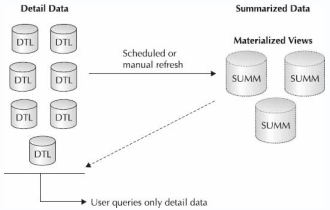



Figure: Materialized View

Benefits:

- For efficient and smooth running of the operational database (hence applications).
- Quick access time. (Local copy access is faster)

# Transactions

- Unit of work. **All or nothing** property.
- Normally **commit** and **rollback** used.

# Transactions

- Unit of work. **All or nothing** property.
- Normally **commit** and **rollback** used.

# Integrity Constraints

- These are the **set of restrictions and rules**. Integrity constraints guard against accidental damage to the database, by ensuring that authorized changes to the database do not result in a loss of data consistency.
- Integrity constraints are introduced **while designing the database schema**. The constraints are specified within the **SQL DDL** command like 'create table' and 'alter table' command.
- Note: It is possible to impose similar restrictions from the UI design, but it is generally recommended to impose them from the back-end (i.e. database) since it ensures better functionality of the system.

Some examples:

- A student must be attached to a department
- Phone number must be fulled up.

# Integrity Constraints

- These are the **set of restrictions and rules**. Integrity constraints guard against accidental damage to the database, by ensuring that authorized changes to the database do not result in a loss of data consistency.
- Integrity constraints are introduced **while designing the database schema**. The constraints are specified within the **SQL DDL** command like 'create table' and 'alter table' command.
- Note: It is possible to impose similar restrictions from the UI design, but it is generally recommended to impose them from the back-end (i.e. database) since it ensures better functionality of the system.

Some examples:

- A student must be attached to a department
- Phone number must be fulled up.

# Integrity Constraints

- These are the **set of restrictions and rules**. Integrity constraints guard against accidental damage to the database, by ensuring that authorized changes to the database do not result in a loss of data consistency.
- Integrity constraints are introduced **while designing the database schema**. The constraints are specified within the **SQL DDL** command like 'create table' and 'alter table' command.
- Note: It is possible to impose similar restrictions from the UI design, but it is generally recommended to impose them from the back-end (i.e. database) since it ensures better functionality of the system.

Some examples:

- A student must be attached to a department
- Phone number must be fulled up.

# Integrity Constraints

- These are the **set of restrictions and rules**. Integrity constraints guard against accidental damage to the database, by ensuring that authorized changes to the database do not result in a loss of data consistency.
- Integrity constraints are introduced **while designing the database schema**. The constraints are specified within the **SQL DDL** command like 'create table' and 'alter table' command.
- Note: It is possible to impose similar restrictions from the UI design, but it is generally recommended to impose them from the back-end (i.e. database) since it ensures better functionality of the system.

Some examples:

- A student must be attached to a department
- Phone number must be fulled up.

# Integrity Constraints

- These are the **set of restrictions and rules**. Integrity constraints guard against accidental damage to the database, by ensuring that authorized changes to the database do not result in a loss of data consistency.
- Integrity constraints are introduced **while designing the database schema**. The constraints are specified within the **SQL DDL** command like 'create table' and 'alter table' command.
- Note: It is possible to impose similar restrictions from the UI design, but it is generally recommended to impose them from the back-end (i.e. database) since it ensures better functionality of the system.

Some examples:

- A student must be attached to a department
- Phone number must be fulled up.

# Integrity Constraints on a Single Relation

- not null
- primary key
- unique
- check (P), where P is a predicate

# Example: Integrity Constraints

```
1  CREATE TABLE EMP
2  (ID NUMBER PRIMARY KEY,
3  NAME VARCHAR2(20) NOT NULL,
4  SALARY NUMBER(10,2),
5  SHORTNAME VARCHAR2(10),
6  CONSTRAINT SAL_CHECK CHECK(SALARY>1000)
7  CONSTRAINT UNIQ_NAME UNIQUE(SHORTNAME)
8  );
9
```

# Referential Integrity

Note: This refers to the Integrity Constraints on **Multiple Relations**

- Foreign Key
- Cascading actions (mostly supported in deletion, not in updating)

# Referential Integrity

Note: This refers to the Integrity Constraints on **Multiple Relations**

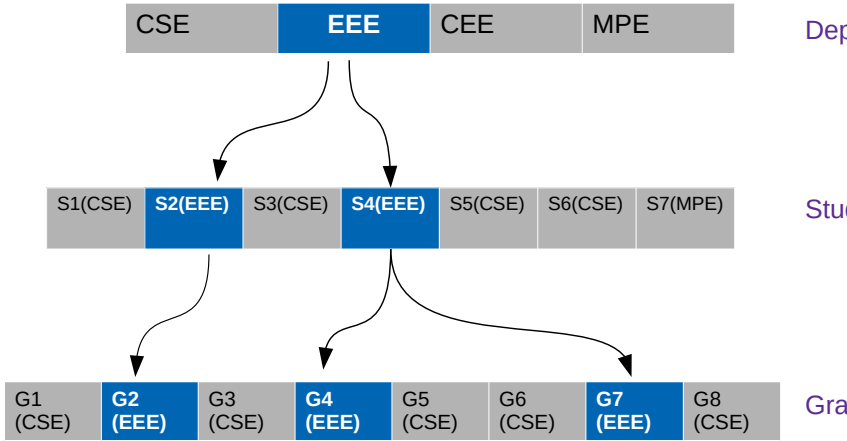
- Foreign Key
- Cascading actions (mostly supported in deletion, not in updating)

# An example:

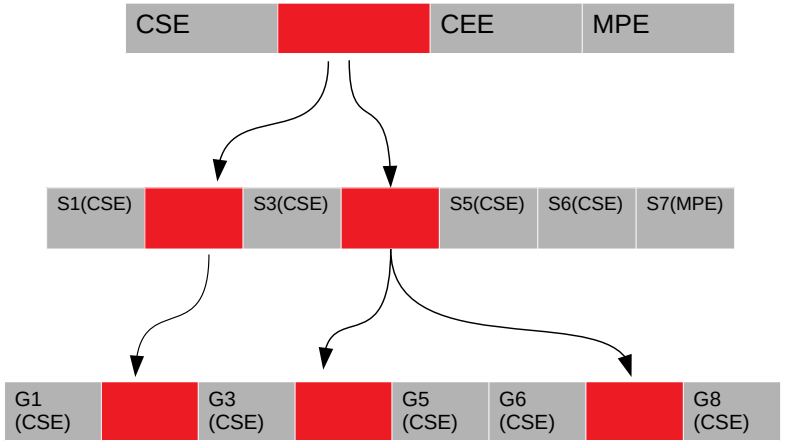
```
create table depts
(id number primary key,
name varchar2(10)
);
```

```
create table students
(sid number primary key,
name varchar2(10),
cgpa number(4,2),
dept number,
constraints fk_stu foreign key(dept)
references depts on delete cascade
);
```

# Cascading Delete : Explained



# Cascading Delete : Explained



# Demo of Cascading Action

Is it always good?

# Integrity Constraint Violation During Transactions

- Consider:

```
create table person (
ID char(10) primary key,
name char(40),
mother char(10),
father char(10),
foreign key father references person,
foreign key mother references person)
```

- How to insert a tuple without causing constraint violation?
  - ✓ Insert father and mother of a person before inserting person
  - ✓ OR, set father and mother to null initially, update after inserting all persons (run a process to it..)
  - ✓ OR defer constraint checking

# Date and Time Types in SQL

2 types of date related basic datatype: 1) Date 2)Timestamp. We will discuss only Date only. See details of Timestamp for your further interest.

## DATE

The DATE data type allows you to store point-in-time values that include both date and time with a precision of one second. It includes Year, Month, Hour, Minute and Second. Default format is DD-MON-YY, example: 20-SEP-18.

# Date (Cont.): *TO_CHAR()* function

TO_CHAR() function is used to change the default format:

| Format Specifier | Meaning |
|---|---|
| YYYY | 4-digit year |
| YY | 2-digit year |
| MONTH | Month name (January - December) |
| MM | Month (1 - 12) |
| DD | Day (1 - 31) |
| DY | Abbreviated day (Sun - Sat) |
| Day | Day (Sunday, Monday) |
| HH24 | Hour (0 - 23) |
| HH or HH12 | Hour (1 - 12) |
| MI | Minutes (0 - 59) |
| SS | Seconds (0 - 59) |

```
1        SELECT TO_CHAR(SYSDATE, '
   YYYY-MM-DD')
2        FROM dual;
3
4        Output:  2012-07-19
5
```

# Date (Cont.): ToDate function

Is used to a convert a string to a DATE datatype.
Example:

```
SELECT TO_DATE( '5 Jan 2017', 'DD MON YYYY' )
FROM dual;

Output: 05-JAN-17

--Now you can use any date function on it:

select to_date('5 Feb 2018','DD MON YYYY')+100
FROM DUAL;

Output: 16-MAY-18
```

# Built-in (major) functions on Date

- ADD_MONTHS(date, n):
  SELECT ADD_MONTHS(SYSDATE,13) FROM DUAL;
  Output: 04-AUG-21
- NEXT_DAY(date,NameOfDay):
  SELECT NEXT_DAY(SYSDATE,'FRIDAY') FROM DUAL;
  Output: 10-JUL-20
- MONTHS_BETWEEN(date1, date2):
  SELECT MONTHS_BETWEEN('18-FEB-92','14-MAY-89')FROM DUAL;
  Output: 33.1290323

# Large-Object Types

- Large objects (photos, videos, CAD files, etc.) are stored as a large object.
- blob: binary large object – object is a large collection of uninterpreted binary data (whose interpretation is left to an application outside of the database system)
- clob: character large object – object is a large collection of character data.
- When a query returns a large object, a pointer is returned rather than the large object itself. Special programming is needed to manipulate it.

# Large-Object Types

- Large objects (photos, videos, CAD files, etc.) are stored as a large object.
- blob: binary large object – object is a large collection of uninterpreted binary data (whose interpretation is left to an application outside of the database system)
- clob: character large object – object is a large collection of character data.
- When a query returns a large object, a pointer is returned rather than the large object itself. Special programming is needed to manipulate it.

# Large-Object Types

- Large objects (photos, videos, CAD files, etc.) are stored as a large object.
- blob: binary large object – object is a large collection of uninterpreted binary data (whose interpretation is left to an application outside of the database system)
- clob: character large object – object is a large collection of character data.
- When a query returns a large object, a pointer is returned rather than the large object itself. Special programming is needed to manipulate it.

# Large-Object Types

- Large objects (photos, videos, CAD files, etc.) are stored as a large object.
- blob: binary large object – object is a large collection of uninterpreted binary data (whose interpretation is left to an application outside of the database system)
- clob: character large object – object is a large collection of character data.
- When a query returns a large object, a pointer is returned rather than the large object itself. Special programming is needed to manipulate it.

# Create Table Extensions: (4.5.6)

- Need similar table structure often.
- We seek a shortcut to lend the structure from existing one.
- structure can be copied as well as data.
- Default it copies both structure and data.
- To exclude data use a clever trick in where clause.

Syntax:

```
1
2   CREATE TABLE NEW_STUDENTS
3   AS SELECT ID, NAME, GPA FROM STUDENTS;
4
5
```

It creates a table similar to the selected attributes and copy all data into it.

# Create Table Extensions: (4.5.6)

- Need similar table structure often.
- We seek a shortcut to lend the structure from existing one.
- structure can be copied as well as data.
- Default it copies both structure and data.
- To exclude data use a clever trick in where clause.

Syntax:

```
1
2   CREATE TABLE NEW_STUDENTS
3   AS SELECT ID, NAME, GPA FROM STUDENTS;
4
5
```

It creates a table similar to the selected attributes and copy all data into it.

# Create Table Extensions: (4.5.6)

- Need similar table structure often.
- We seek a shortcut to lend the structure from existing one.
- structure can be copied as well as data.
- Default it copies both structure and data.
- To exclude data use a clever trick in where clause.

Syntax:

```
1
2   CREATE TABLE NEW_STUDENTS
3   AS SELECT ID, NAME, GPA FROM STUDENTS;
4
5
```

It creates a table similar to the selected attributes and copy all data into it.

# Create Table Extensions: (4.5.6)

- Need similar table structure often.
- We seek a shortcut to lend the structure from existing one.
- structure can be copied as well as data.
- Default it copies both structure and data.
- To exclude data use a clever trick in where clause.

Syntax:

```
1
2   CREATE TABLE NEW_STUDENTS
3   AS SELECT ID, NAME, GPA FROM STUDENTS;
4
5
```

It creates a table similar to the selected attributes and copy all data into it.

# Create Table Extensions: (4.5.6)

- Need similar table structure often.
- We seek a shortcut to lend the structure from existing one.
- structure can be copied as well as data.
- Default it copies both structure and data.
- To exclude data use a clever trick in where clause.

Syntax:

```
1
2   CREATE TABLE NEW_STUDENTS
3   AS SELECT ID, NAME, GPA FROM STUDENTS;
4
5
```

It creates a table similar to the selected attributes and copy all data into it.

# Create Table Extensions: (4.5.6)

- Need similar table structure often.
- We seek a shortcut to lend the structure from existing one.
- structure can be copied as well as data.
- Default it copies both structure and data.
- To exclude data use a clever trick in where clause.

Syntax:

```
1
2   CREATE TABLE NEW_STUDENTS
3   AS SELECT ID, NAME, GPA FROM STUDENTS;
4
5
```

It creates a table similar to the selected attributes and copy all data into it.

# Create Table Extensions: (4.5.6)

- Need similar table structure often.
- We seek a shortcut to lend the structure from existing one.
- structure can be copied as well as data.
- Default it copies both structure and data.
- To exclude data use a clever trick in where clause.

Syntax:

```
1
2     CREATE TABLE NEW_STUDENTS
3     AS SELECT ID, NAME, GPA FROM STUDENTS;
4
5
```

It creates a table similar to the selected attributes and copy all data into it.

# Authorization (4.6)

- Role-based Access Control.
- It is used to distribute a large system into a number of users with different access controls.
- Normally SELECT, UPDATE, DELETE, INSERT are the privileges on a database for tables or views.
- Additionally, EXECUTE is another privilege for sub-program (will be covered later).
- Role is created and customized (it is like a package of actions)
- Then the owner grant (opposite is itermrevoke) the created role to a specific user.
- Roles can be defined using another role.

# Authorization (4.6)

- Role-based Access Control.
- It is used to distribute a large system into a number of users with different access controls.
- Normally SELECT, UPDATE, DELETE, INSERT are the privileges on a database for tables or views.
- Additionally, EXECUTE is another privilege for sub-program (will be covered later).
- Role is created and customized (it is like a package of actions)
- Then the owner grant (opposite is itermrevoke) the created role to a specific user.
- Roles can be defined using another role.

# Authorization (4.6)

- Role-based Access Control.
- It is used to distribute a large system into a number of users with different access controls.
- Normally SELECT, UPDATE, DELETE, INSERT are the privileges on a database for tables or views.
- Additionally, EXECUTE is another privilege for sub-program (will be covered later).
- Role is created and customized (it is like a package of actions)
- Then the owner grant (opposite is itermrevoke) the created role to a specific user.
- Roles can be defined using another role.

# Authorization (4.6)

- Role-based Access Control.
- It is used to distribute a large system into a number of users with different access controls.
- Normally SELECT, UPDATE, DELETE, INSERT are the privileges on a database for tables or views.
- Additionally, EXECUTE is another privilege for sub-program (will be covered later).
- Role is created and customized (it is like a package of actions)
- Then the owner grant (opposite is itermrevoke) the created role to a specific user.
- Roles can be defined using another role.

# Authorization (4.6)

- Role-based Access Control.
- It is used to distribute a large system into a number of users with different access controls.
- Normally SELECT, UPDATE, DELETE, INSERT are the privileges on a database for tables or views.
- Additionally, EXECUTE is another privilege for sub-program (will be covered later).
- Role is created and customized (it is like a package of actions)
- Then the owner grant (opposite is itermrevoke) the created role to a specific user.
- Roles can be defined using another role.

# Authorization (4.6)

- Role-based Access Control.
- It is used to distribute a large system into a number of users with different access controls.
- Normally SELECT, UPDATE, DELETE, INSERT are the privileges on a database for tables or views.
- Additionally, EXECUTE is another privilege for sub-program (will be covered later).
- Role is created and customized (it is like a package of actions)
- Then the owner grant (opposite is itermrevoke) the created role to a specific user.
- Roles can be defined using another role.

# Authorization (4.6)

- Role-based Access Control.
- It is used to distribute a large system into a number of users with different access controls.
- Normally SELECT, UPDATE, DELETE, INSERT are the privileges on a database for tables or views.
- Additionally, EXECUTE is another privilege for sub-program (will be covered later).
- Role is created and customized (it is like a package of actions)
- Then the owner grant (opposite is itermrevoke) the created role to a specific user.
- Roles can be defined using another role.

# Granting privileges: Syntax

```
1  grant <privilege list>
2  on <relation name or view name>
3  to <user/role list>;
4
5
6  --privilege list is:
7  SELECT, UPDATE, INSERT, DELETE, (AND EXECUTE)
8
```

# Role-Based Access Control: Overview

**Given Scenario:** Result Processing System (RPS)

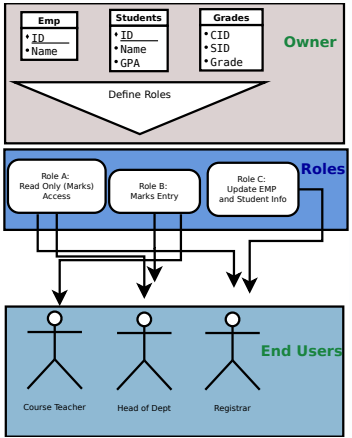



Figure: Role-based Access Control: Concept

Thank You.