



Chapter 4 : Intermediate SQL

Database System Concepts, 7th Ed.

©Silberschatz, Korth and Sudarshan

See www.db-book.com for conditions on re-use



Chapter 4: Intermediate SQL

- Join Expressions
- Views
- Transactions
- Integrity Constraints
- SQL Data Types and Schemas
- Index Definition in SQL
- Authorization



Problem?

Employee	
<u>LastName</u>	<u>DepartmentName</u>
Rafferty	Sales
Jones	Engineering
Heisenberg	Engineering
Robinson	Clerical
Smith	Clerical
Williams	NULL

Redundant Data!!!



Solution

Employee	
<u>LastName</u>	<u>DepartmentName</u>
Rafferty	Sales
Jones	Engineering
Heisenberg	Engineering
Robinson	Clerical
Smith	Clerical
Williams	NULL

Employee Table

Department Table

<u>LastName</u>
Rafferty
Jones
Heisenberg
Robinson
Smith
Williams

<u>DepartmentName</u>
Sales
Engineering
Engineering
Clerical
Clerical
NULL



Solution

Employee	
<u>LastName</u>	<u>DepartmentName</u>
Rafferty	Sales
Jones	Engineering
Heisenberg	Engineering
Robinson	Clerical
Smith	Clerical
Williams	NULL

Employee Table

Department Table

<u>LastName</u>
Rafferty
Jones
Heisenberg
Robinson
Smith
Williams

<u>DepartmentName</u>
Sales
Engineering
Clerical
Marketing

Delete Redundant Data!



Solution

Add primary key (ID)

Employee	
<u>LastName</u>	<u>DepartmentName</u>
Rafferty	Sales
Jones	Engineering
Heisenberg	Engineering
Robinson	Clerical
Smith	Clerical
Williams	NULL

Employee Table

Department Table

<u>LastName</u>
Rafferty
Jones
Heisenberg
Robinson
Smith
Williams

Department	
<u>DepartmentID</u>	<u>DepartmentName</u>
31	Sales
33	Engineering
34	Clerical
35	Marketing



Solution

Employee	
<u>LastName</u>	<u>DepartmentName</u>
Rafferty	Sales
Jones	Engineering
Heisenberg	Engineering
Robinson	Clerical
Smith	Clerical
Williams	NULL

Employee Table

Department Table

How to make a relationship
between two tables?

Foreign Key

<u>LastName</u>
Rafferty
Jones
Heisenberg
Robinson
Smith
Williams

Department	
<u>DepartmentID</u>	<u>DepartmentName</u>
31	Sales
33	Engineering
34	Clerical
35	Marketing



Solution

Employee	
<u>LastName</u>	<u>DepartmentName</u>
Rafferty	Sales
Jones	Engineering
Heisenberg	Engineering
Robinson	Clerical
Smith	Clerical
Williams	NULL

Employee Table

Department Table

Employee	
<u>LastName</u>	<u>DepartmentID</u>
Rafferty	
Jones	
Heisenberg	
Robinson	
Smith	
Williams	

Department	
<u>DepartmentID</u>	<u>DepartmentName</u>
31	Sales
33	Engineering
34	Clerical
35	Marketing



Solution

Employee	
<u>LastName</u>	<u>DepartmentName</u>
Rafferty	Sales
Jones	Engineering
Heisenberg	Engineering
Robinson	Clerical
Smith	Clerical
Williams	NULL

Employee Table

Department Table

Employee	
<u>LastName</u>	<u>DepartmentID</u>
Rafferty	31
Jones	33
Heisenberg	33
Robinson	34
Smith	34
Williams	NULL

Department	
<u>DepartmentID</u>	<u>DepartmentName</u>
31	Sales
33	Engineering
34	Clerical
35	Marketing



Structured Query Language (SQL)

SELECT *<attribute list>*

FROM *<table list >*

WHERE *<condition>*

- Language for constructing a new table from argument table(s).
 - **FROM** indicates source tables
 - **WHERE** indicates which *rows* to retain
 - It acts as a filter
 - **SELECT** indicates which *columns* to extract from retained rows
 - Projection
- The result is a table.



Example

```
SELECT Name
FROM Student
WHERE Id > 4999;
```

<i>Id</i>	<i>Name</i>	<i>Address</i>	<i>Status</i>
1234	John	123 Main	fresh
5522	Mary	77 Pine	senior
9876	Bill	83 Oak	junior

Student

<i>Name</i>
Mary
Bill

Result



Examples

```
SELECT Id, Name FROM Student;
```

```
SELECT Id, Name FROM Student  
WHERE Status = 'senior';
```

```
SELECT * FROM Student  
WHERE Status = 'senior';
```

*result is a table
with one column
and one row*

```
SELECT COUNT(*) FROM Student  
WHERE Status = 'senior';
```



More Complex Example

- **Goal:** table in which each row names a senior and gives a course taken and grade
- Combines information in two tables:
 - Student: *Id, Name, Address, Status*
 - Transcript: *StudId, CrsCode, Semester, Grade*

```
SELECT Name, CrsCode, Grade
FROM   Student, Transcript
WHERE  StudId = Id AND Status = 'senior';
```



Join

SELECT $a1, b1$
FROM T1, T2
WHERE $a2 = b2$

T1			T2	
$a1$	$a2$	$a3$	$b1$	$b2$
A	1	xyy	3.2	17
B	17	rst	4.8	17

FROM T1, T2
yields:

$a1$	$a2$	$a3$	$b1$	$b2$
A	1	xyy	3.2	17
A	1	xyy	4.8	17
B	17	rst	3.2	17
B	17	rst	4.8	17

WHERE $a2 = b2$
yields:

B	17	rst	3.2	17
B	17	rst	4.8	17

SELECT $a1, b1$
yields result:

B	3.2
B	4.8



Modifying Tables

```
UPDATE Student  
SET Status = 'soph'  
WHERE Id = 111111111;
```

```
INSERT INTO Student (Id, Name, Address, Status)  
VALUES (999999999, 'Bill', '432 Pine', 'senior')
```

```
DELETE FROM Student  
WHERE Id = 111111111
```



Practice

Find the titles of courses in the Comp. Sci. department that have 3 credits.

```
select title  
from course  
where dept name = 'Comp. Sci.'  
and credits = 3
```

Find the highest salary of any instructor.

```
select max(salary)  
from instructor
```

Find all instructors earning the highest salary (there may be more than one with the same salary).

```
select ID, name  
from instructor  
where salary = (select max(salary) from instructor)
```




Practice

write a query that finds departments whose names contain the string “Sci” as a substring.

```
select dept_name  
from department  
where dept_name like '%Sci%'
```

Find all instructors who do not work for Computer Science department.
(Assume that all people work for exactly one department).

```
select name  
from instructor  
where dept_name <> 'Comp. Sci.'
```



Practice

Modify the database so that Kim now teaches in Biology.
(Assume that each person has only one tuple in the *instructor* relation)

```
update instructor  
set dept_name = 'Biology'  
where name = 'Kim'
```

Increase the salary of each instructor in the Comp. Sci. department by 10%.

```
update instructor  
set salary = salary * 1.10  
where dept name = 'Comp. Sci.'
```



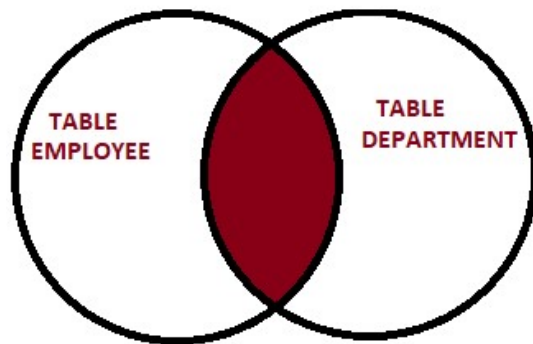
Joined Relations

- **Join operations** take two relations and return as a result another relation.
- A join operation is a Cartesian product which requires that tuples in the two relations match (under some condition). It also specifies the attributes that are present in the result of the join
- The join operations are typically used as subquery expressions in the **from** clause
- Three types of joins:
 - Natural join
 - Inner join
 - Outer join

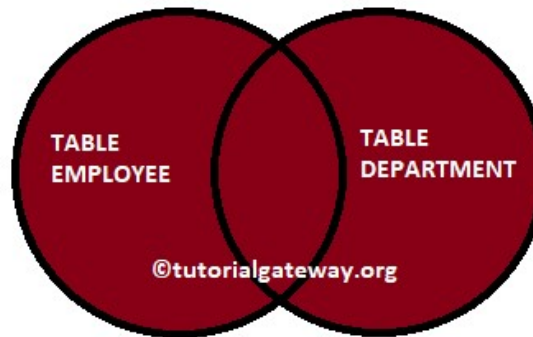


Join

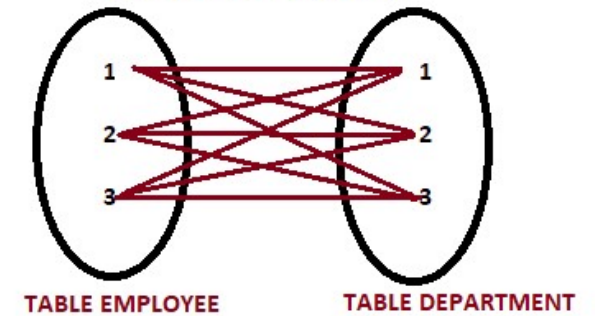
INNER JOIN EXAMPLE



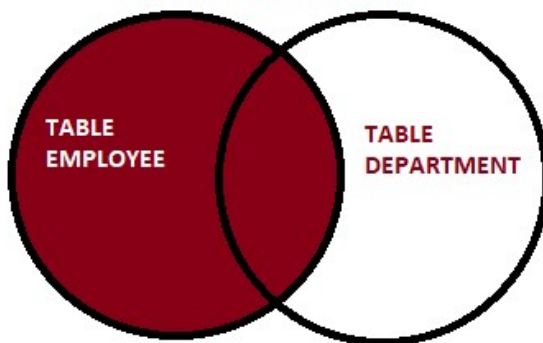
FULL JOIN EXAMPLE



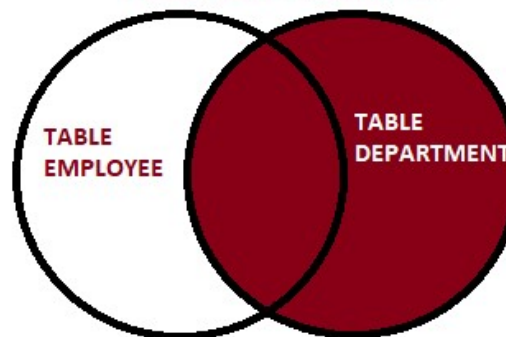
CROSS JOIN EXAMPLE



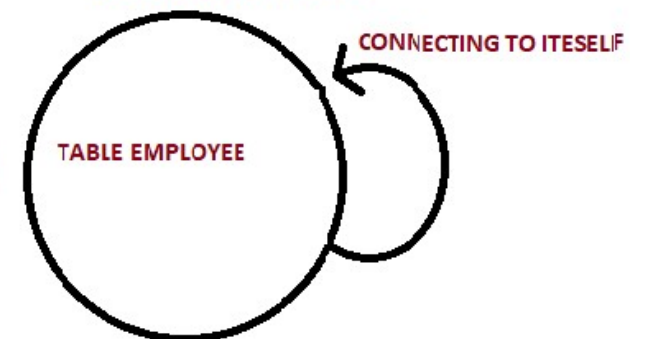
LEFT JOIN EXAMPLE



RIGHT JOIN EXAMPLE



SELF JOIN EXAMPLE





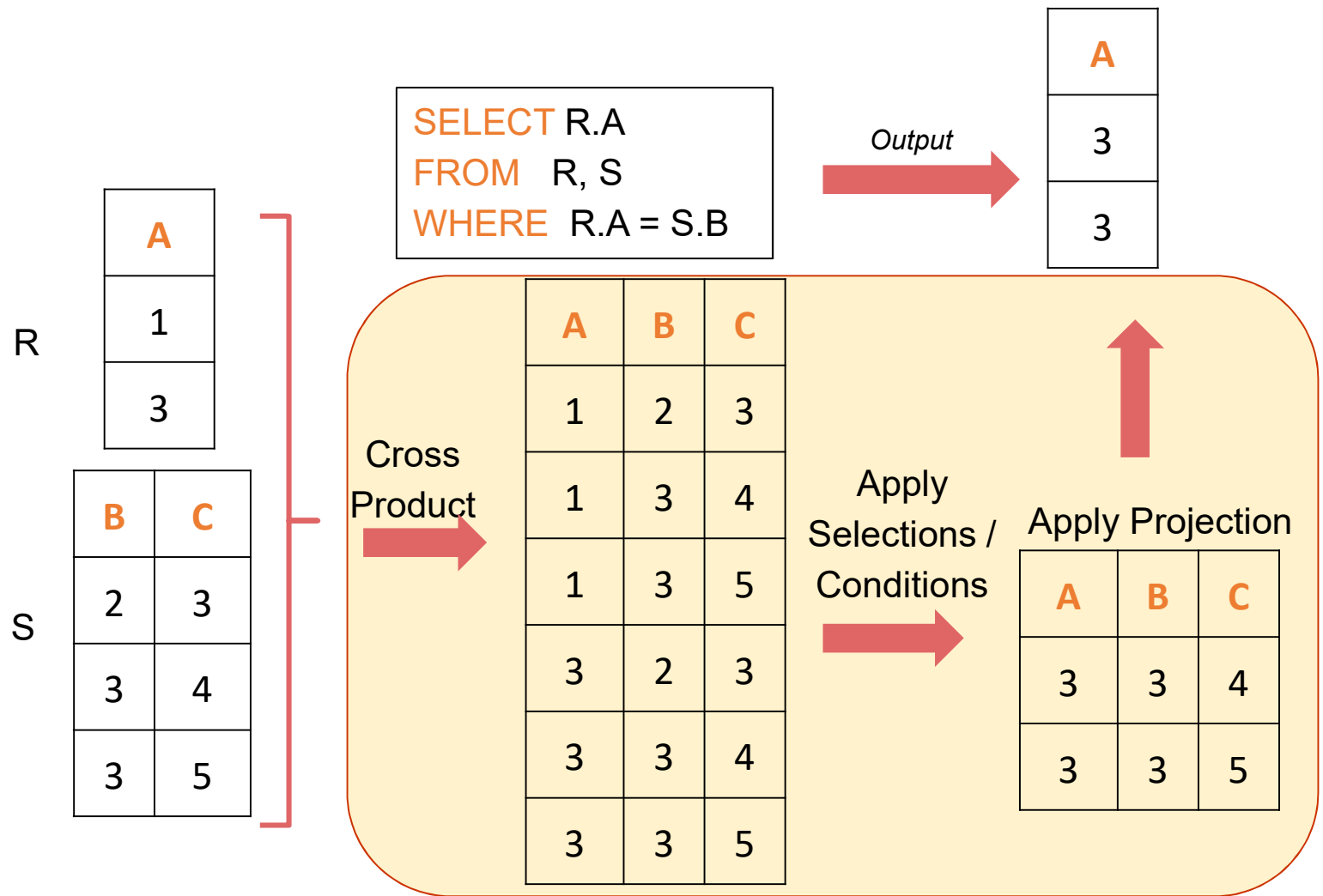
Semantics of JOINS

```
SELECT  $x_1.a_1, x_1.a_2, \dots, x_n.a_k$   
FROM  $R_1$  AS  $x_1, R_2$  AS  $x_2, \dots, R_n$  AS  $x_n$   
WHERE  $\text{Conditions}(x_1, \dots, x_n)$ 
```

```
Answer = {}  
for  $x_1$  in  $R_1$  do  
  for  $x_2$  in  $R_2$  do  
    ....  
    for  $x_n$  in  $R_n$  do  
      if  $\text{Conditions}(x_1, \dots, x_n)$   
        then Answer = Answer  $\cup \{(x_1.a_1, x_1.a_2, \dots, x_n.a_k)\}$   
return Answer
```



An example of SQL semantics





Practice

MySQL supports the following types of joins:

Cross join

Inner join

Left join

Right join

MySQL Tutorial:

<http://www.mysqltutorial.org/>

[https://en.wikipedia.org/wiki/Join_\(SQL\)](https://en.wikipedia.org/wiki/Join_(SQL))

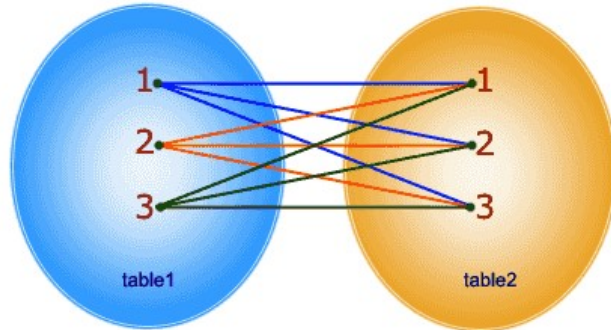


Example

T1	
<i>col1</i>	<i>col2</i>
1	11
2	22

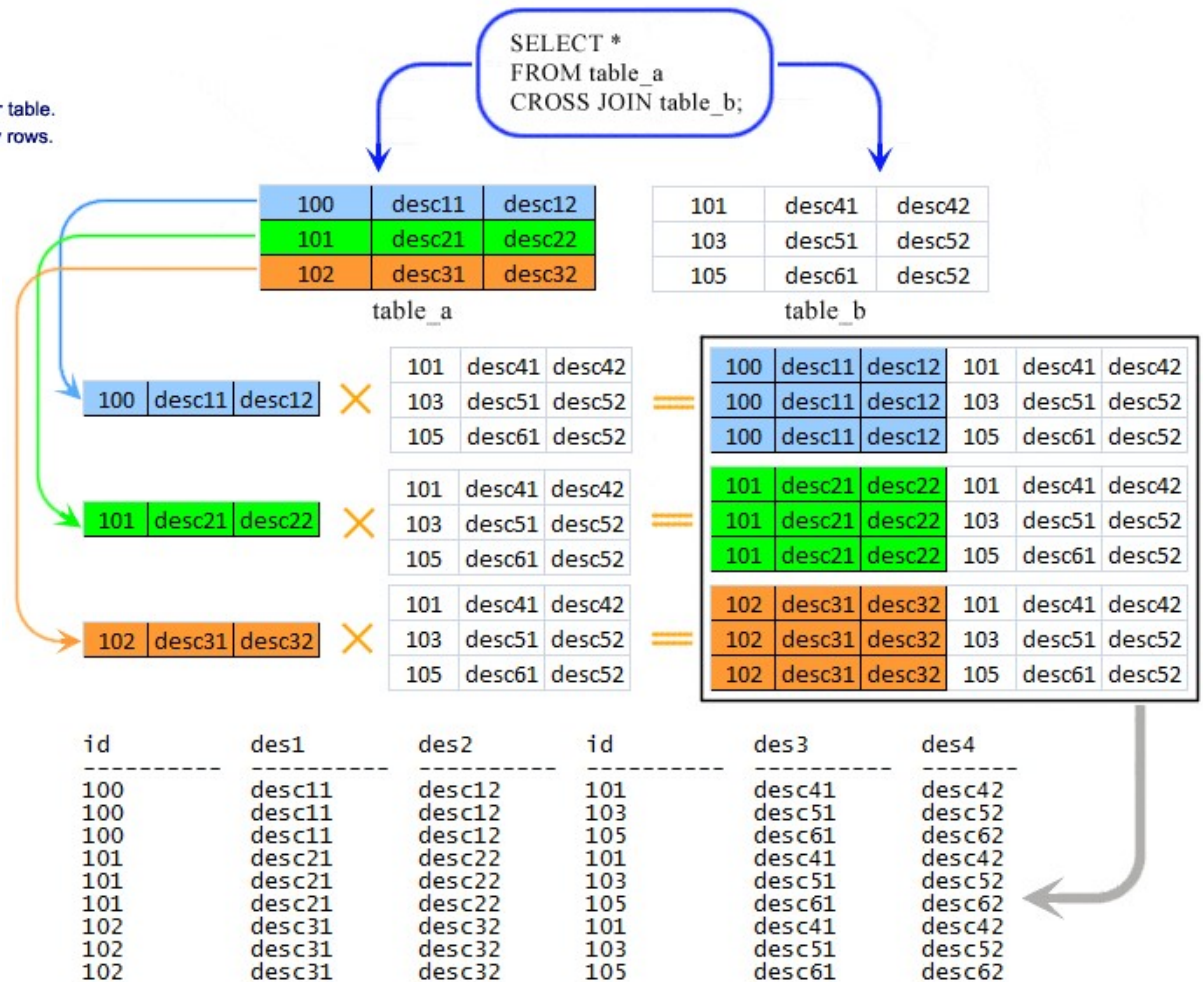
T2	
<i>col1</i>	<i>col3</i>
10	101
2	202

SELECT * FROM table1 CROSS JOIN table2;



In CROSS JOIN, each row from 1st table joins with all the rows of another table.
If 1st table contain x rows and y rows in 2nd one the result set will be $x * y$ rows.

Cross Join Cartesian Product





Cross Join

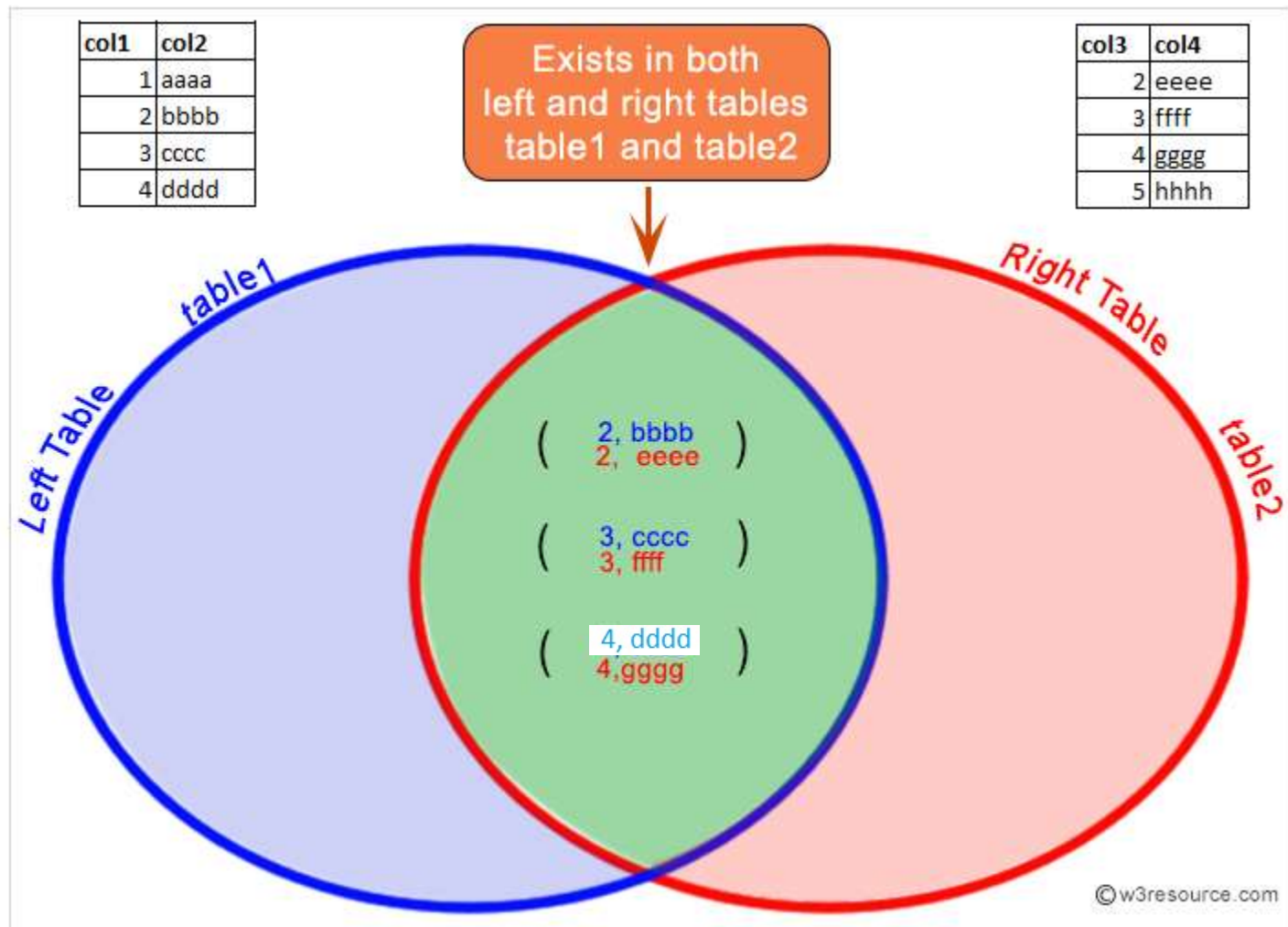
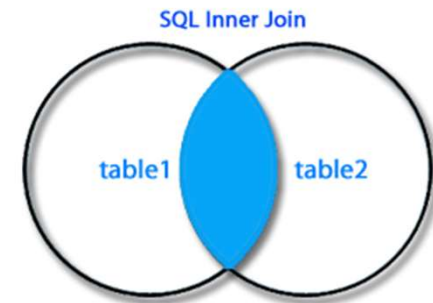
Employee	
<u>LastName</u>	<u>DepartmentID</u>
Rafferty	31
Jones	33
Heisenberg	33
Robinson	34
Smith	34
Williams	NULL

Department	
<u>DepartmentID</u>	<u>DepartmentName</u>
31	Sales
33	Engineering
34	Clerical
35	Marketing

Employee.LastName	Employee.DepartmentID	Department.DepartmentName	Department.DepartmentID
Rafferty	31	Sales	31
Jones	33	Sales	31
Heisenberg	33	Sales	31
Smith	34	Sales	31
Robinson	34	Sales	31
Williams	NULL	Sales	31
Rafferty	31	Engineering	33
Jones	33	Engineering	33
Heisenberg	33	Engineering	33
Smith	34	Engineering	33
Robinson	34	Engineering	33
Williams	NULL	Engineering	33
Rafferty	31	Clerical	34
Jones	33	Clerical	34
Heisenberg	33	Clerical	34
Smith	34	Clerical	34
Robinson	34	Clerical	34
Williams	NULL	Clerical	34
Rafferty	31	Marketing	35
Jones	33	Marketing	35
Heisenberg	33	Marketing	35
Smith	34	Marketing	35
Robinson	34	Marketing	35
Williams	NULL	Marketing	35



Inner Join





Join Condition

- The **on** condition allows a general predicate over the relations being joined.
- This predicate is written like a **where** clause predicate except for the use of the keyword **on**.

- Query example

select *

from *student* **join** *takes* **on** *student_ID* = *takes_ID*

- The **on** condition above specifies that a tuple from *student* matches a tuple from *takes* if their *ID* values are equal.

- Equivalent to:

select *

from *student* , *takes*

where *student_ID* = *takes_ID*



Inner join

Employee	
<u>LastName</u>	<u>DepartmentID</u>
Rafferty	31
Jones	33
Heisenberg	33
Robinson	34
Smith	34
Williams	NULL

Department	
<u>DepartmentID</u>	<u>DepartmentName</u>
31	Sales
33	Engineering
34	Clerical
35	Marketing

Employee.LastName	Employee.DepartmentID	Department.DepartmentName
Robinson	34	Clerical
Jones	33	Engineering
Smith	34	Clerical
Heisenberg	33	Engineering
Rafferty	31	Sales

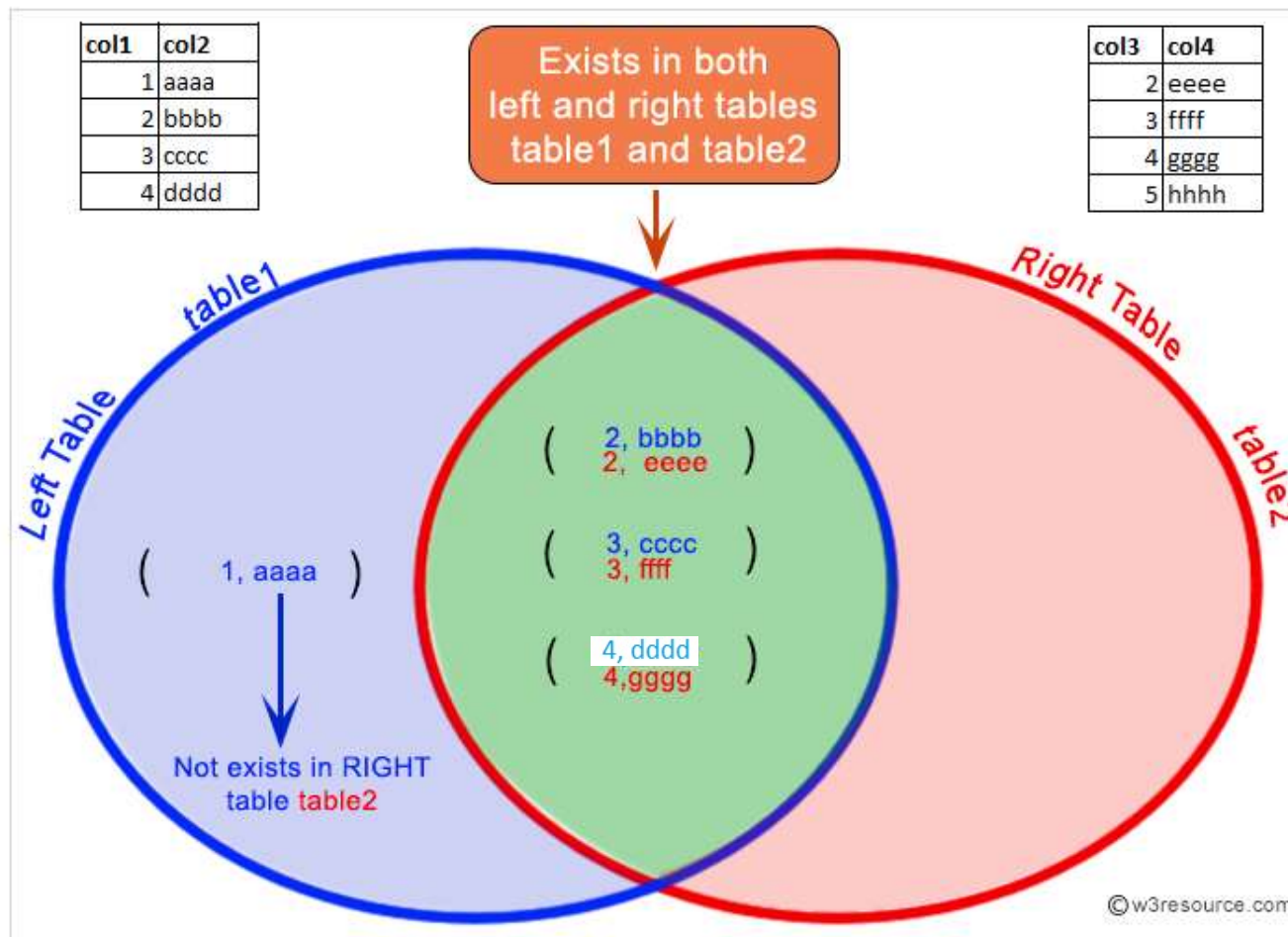


Outer Join

- An extension of the join operation that avoids loss of information.
- Computes the join and then adds tuples from one relation that does not match tuples in the other relation to the result of the join.
- Uses *null* values.
- Three forms of outer join:
 - left outer join
 - right outer join
 - full outer join

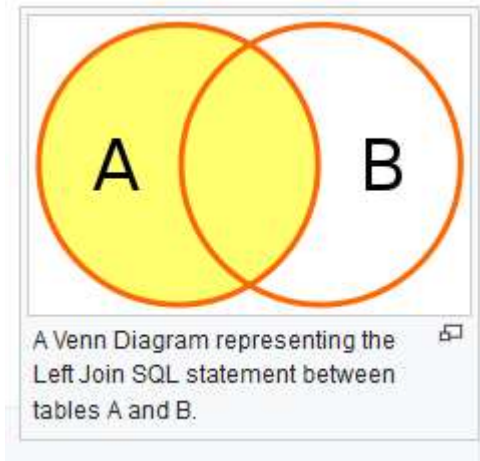


Left (outer) Join





Left Outer Join



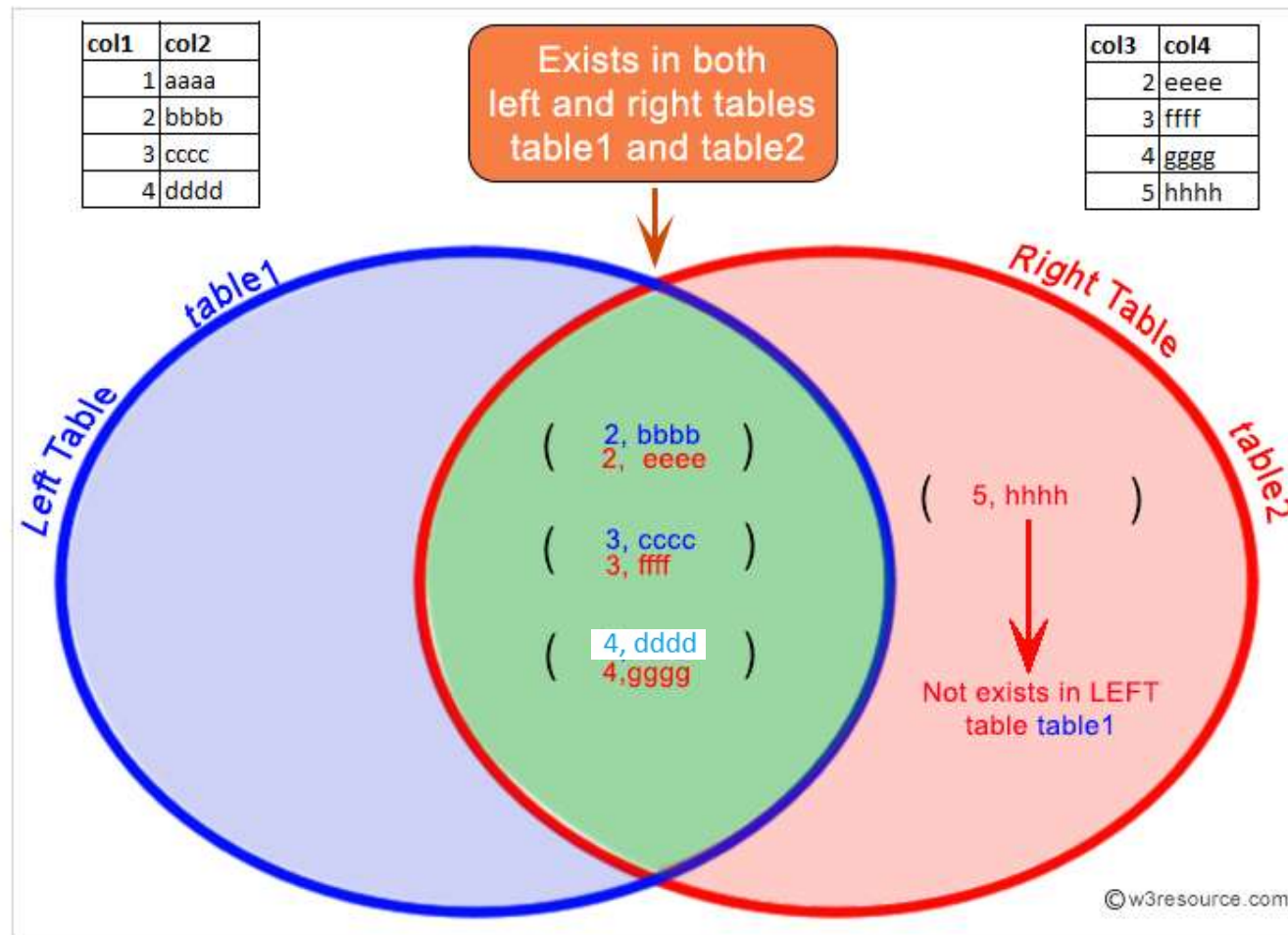
Employee	
<u>LastName</u>	<u>DepartmentID</u>
Rafferty	31
Jones	33
Heisenberg	33
Robinson	34
Smith	34
Williams	NULL

Department	
<u>DepartmentID</u>	<u>DepartmentName</u>
31	Sales
33	Engineering
34	Clerical
35	Marketing

Employee.LastName	Employee.DepartmentID	Department.DepartmentName	Department.DepartmentID
Jones	33	Engineering	33
Rafferty	31	Sales	31
Robinson	34	Clerical	34
Smith	34	Clerical	34
Williams	NULL	NULL	NULL
Heisenberg	33	Engineering	33

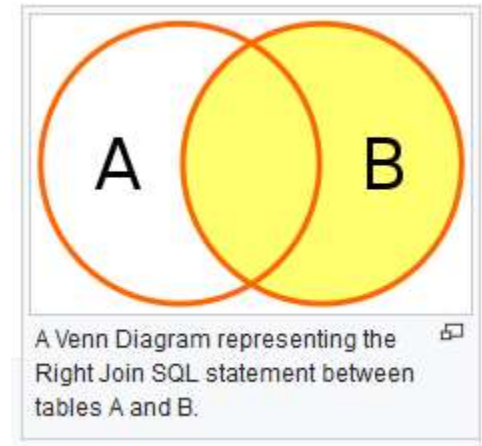


Right(Outer) Join





Right Outer Join



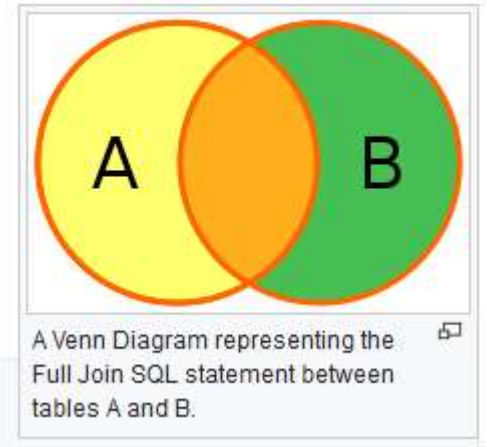
Employee	
<u>LastName</u>	<u>DepartmentID</u>
Rafferty	31
Jones	33
Heisenberg	33
Robinson	34
Smith	34
Williams	NULL

Department	
<u>DepartmentID</u>	<u>DepartmentName</u>
31	Sales
33	Engineering
34	Clerical
35	Marketing

Employee.LastName	Employee.DepartmentID	Department.DepartmentName	Department.DepartmentID
Smith	34	Clerical	34
Jones	33	Engineering	33
Robinson	34	Clerical	34
Heisenberg	33	Engineering	33
Rafferty	31	Sales	31
NULL	NULL	Marketing	35



Full Outer Join



Employee	
<u>LastName</u>	<u>DepartmentID</u>
Rafferty	31
Jones	33
Heisenberg	33
Robinson	34
Smith	34
Williams	NULL

Department	
<u>DepartmentID</u>	<u>DepartmentName</u>
31	Sales
33	Engineering
34	Clerical
35	Marketing

Employee.LastName	Employee.DepartmentID	Department.DepartmentName	Department.DepartmentID
Smith	34	Clerical	34
Jones	33	Engineering	33
Robinson	34	Clerical	34
Williams	NULL	NULL	NULL
Heisenberg	33	Engineering	33
Rafferty	31	Sales	31
NULL	NULL	Marketing	35



Other Join

Equi-join:

An equi-join is a specific type of comparator-based join, that uses only **equality (=)** comparisons in the join-predicate. Using other comparison operators (such as <) disqualifies a join as an equi-join. The query shown above has already provided an example of an equi-join:

```
SELECT *  
FROM employee JOIN department  
ON employee.DepartmentID = department.DepartmentID;
```

We can write equi-join as below:

```
SELECT *  
FROM employee, department  
WHERE employee.DepartmentID = department.DepartmentID;
```



Other Join

Natural join:

The natural join is a special case of equi-join. Natural join (\bowtie) is a binary operator that is written as $(R \bowtie S)$ where R and S are relations.

The result of the natural join is the set of all combinations of tuples in R and S that are **equal on their common attribute names**. For an example consider the tables *Employee* and *Dept* and their natural join:

<i>Employee</i>		
Name	EmpId	DeptName
Harry	3415	Finance
Sally	2241	Sales
George	3401	Finance
Harriet	2202	Sales

<i>Dept</i>	
DeptName	Manager
Finance	George
Sales	Harriet
Production	Charles

<i>Employee</i> \bowtie <i>Dept</i>			
Name	EmpId	DeptName	Manager
Harry	3415	Finance	George
Sally	2241	Sales	Harriet
George	3401	Finance	George
Harriet	2202	Sales	Harriet

```
SELECT *  
FROM employee NATURAL JOIN department;
```



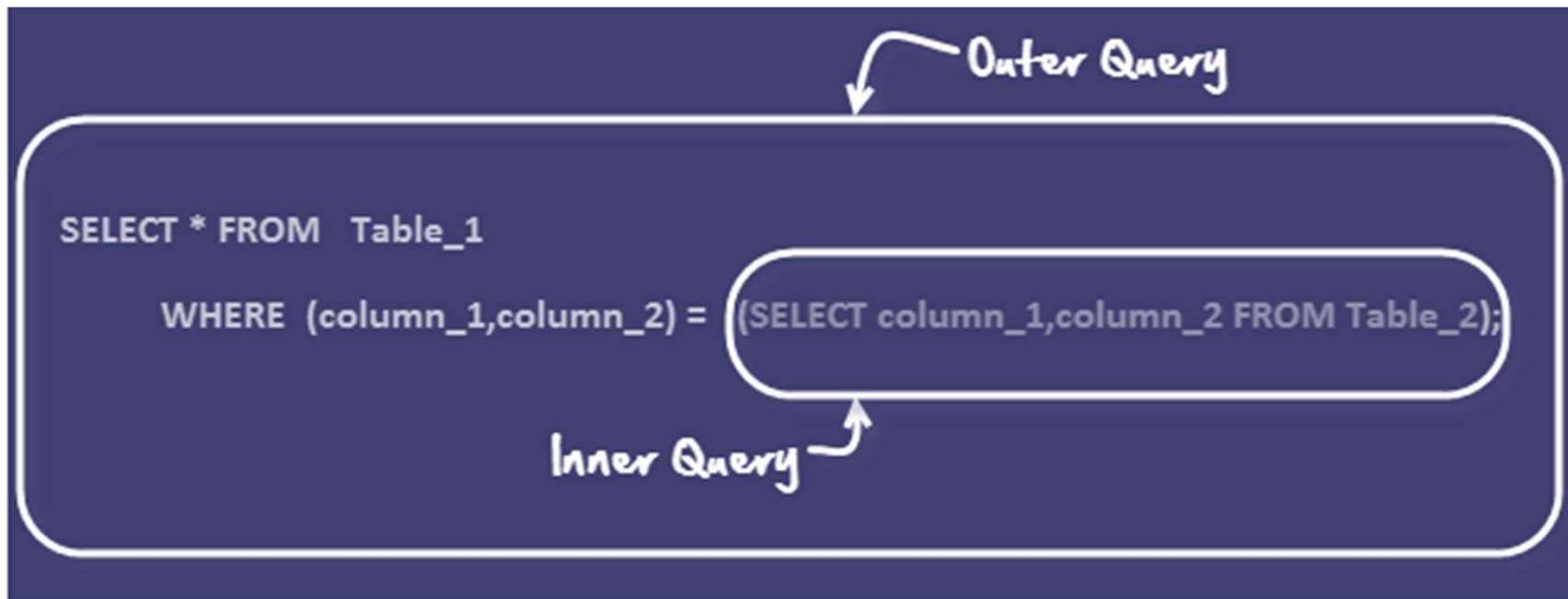
Implementation of Division

- Using Subquery



Subquery

- A sub query is a select query that is contained inside another query. The inner select query is usually used to determine the results of the outer select query.



Subqueries are embedded queries inside another query. The embedded query is known as the **inner query** and the container query is known as the **outer query**.



Example

movies = (movie_id, title, director, year_released, category_id)

```
SELECT category_name
FROM categories
WHERE category_id = ( SELECT MIN(category_id)
                     FROM movies );
```

First the INNER Query is executed

```
SELECT MIN(category_id) from movies
```

INNER Query gives following result

MIN(category_id)
1

Output of INNER Query is substituted in OUTER Query

```
SELECT category_name FROM categories WHERE category_id =1
```

On Execution OUTER Query gives following Result

category_name
Comedy

The above is a form of **Row Sub-Query**. In such sub-queries the, inner query can give only **ONE result**. The permissible operators when work with row subqueries are [=, >, =, <=, !=,]



Example

```
SELECT full_names,contact_number
FROM members
WHERE membership_number IN ( SELECT membership_number
                             FROM movierentals
                             WHERE return_date IS NULL );
```

First the INNER Query is executed

SELECT membership_number FROM movierentals WHERE return_date IS NULL

INNER Query gives following result

membership_number
1
3

Output of INNER Query is substituted in OUTER Query

SELECT full_names,contact_number FROM members WHERE membership_number IN (1,3)

On Execution OUTER Query gives following Result

full_names	contact_number
Janet Jones	0759 253 542
Robert Phil	12345

In this case, the inner query returns **more than one results**. The above is type of **Table sub-query**.



Subqueries

- A subquery may occur in:
 - A SELECT clause
 - A FROM clause
 - A WHERE clause
- **Rule of thumb:** avoid writing nested queries when possible; keep in mind that sometimes it's impossible



Correlated Nested Queries

Output a row $\langle prof, dept \rangle$ if $prof$ has taught a course in $dept$.

```
SELECT P.Name, D.Name           --outer query
FROM Professor P, Department D
WHERE P.Id IN                   -- set of all ProfId's who have taught a course in D.DeptId
    (SELECT T.ProfId           --subquery
     FROM Teaching T, Course C
     WHERE T.CrsCode = C.CrsCode AND
           C.DeptId = D.DeptId --correlation
    )
```



Correlated Nested Queries (con't)

- Tuple variables T and C are local to subquery
- Tuple variables P and D are global to subquery
- Correlation: subquery uses a global variable, D
- Correlated queries can be expensive to evaluate



1. Subqueries in SELECT

Company(cid, cname, city)

Product (pname, price, cid)

For each product return the city where it is manufactured



1. Subqueries in SELECT

Product (pname, price, cid)

Company(cid, cname, city)

For each product return the city where it is manufactured

```
SELECT X.pname, ( SELECT Y.city  
                  FROM Company Y  
                  WHERE Y.cid=X.cid) as City  
FROM Product X
```



1. Subqueries in SELECT

Product (pname, price, cid)

Company(cid, cname, city)

For each product return the city where it is manufactured

```
SELECT X.pname, (SELECT Y.city  
                  FROM Company Y  
                  WHERE Y.cid=X.cid) as City  
FROM Product X
```

What happens if the subquery returns more than one city?

We get a runtime error

(Some DBMS simply ignore the extra values)



1. Subqueries in SELECT

Product (pname, price, cid)

Company(cid, cname, city)

For each product return the city where it is manufactured

```
SELECT X.pname, (SELECT Y.city  
                  FROM Company Y  
                  WHERE Y.cid=X.cid) as City  
FROM Product X
```

“correlated
subquery”

What happens if the subquery returns more than one city?

We get a runtime error

(Some DBMS simply ignore the extra values)



1. Subqueries in SELECT

Product (pname, price, cid)
Company(cid, cname, city)

For each product return the city where it is manufactured

```
SELECT X.pname, (SELECT Y.city  
FROM Company Y  
WHERE Y.cid=X.cid) as City  
FROM Product X
```

“correlated
subquery”

What happens if the subquery returns more than one city?

We get a runtime error

(Some DBMS simply ignore the extra values)



1. Subqueries in SELECT

Product (pname, price, cid)

Company(cid, cname, city)

Whenever possible, don't use a nested queries:

```
SELECT X.pname, (SELECT Y.city  
                  FROM Company Y  
                  WHERE Y.cid=X.cid) as City  
FROM Product X
```

||

```
SELECT X.pname, Y.city  
FROM Product X, Company Y  
WHERE X.cid=Y.cid
```



1. Subqueries in SELECT

Product (pname, price, cid)

Company(cid, cname, city)

Whenever possible, don't use a nested queries:

```
SELECT X.pname, (SELECT Y.city
                  FROM Company Y
                  WHERE Y.cid=X.cid) as City
FROM Product X
```

||

```
SELECT X.pname, Y.city
FROM Product X, Company Y
WHERE X.cid=Y.cid
```

We have
“unnested”
the query



1. Subqueries in SELECT

Product (pname, price, cid)

Company(cid, cname, city)

Compute the number of products made by each company

```
SELECT DISTINCT C.cname, (SELECT count(*)  
                           FROM Product P  
                           WHERE P.cid=C.cid)  
FROM Company C
```



1. Subqueries in SELECT

Product (pname, price, cid)

Company(cid, cname, city)

Compute the number of products made by each company

```
SELECT DISTINCT C.cname, (SELECT count(*)  
                           FROM Product P  
                           WHERE P.cid=C.cid)  
FROM Company C
```

Better: we can
unnest by using
a GROUP BY

```
SELECT C.cname, count(*)  
FROM Company C, Product P  
WHERE C.cid=P.cid  
GROUP BY C.cname
```



1. Subqueries in SELECT

Product (pname, price, cid)

Company(cid, cname, city)

But are these really equivalent?

```
SELECT DISTINCT C.cname, (SELECT count(*)  
                           FROM Product P  
                           WHERE P.cid=C.cid)  
FROM Company C
```

```
SELECT C.cname, count(*)  
FROM Company C, Product P  
WHERE C.cid=P.cid  
GROUP BY C.cname
```



1. Subqueries in SELECT

Product (pname, price, cid)

Company(cid, cname, city)

But are these really equivalent?

```
SELECT DISTINCT C.cname, (SELECT count(*)  
                           FROM Product P  
                           WHERE P.cid=C.cid)  
FROM Company C
```

```
SELECT C.cname, count(*)  
FROM Company C, Product P  
WHERE C.cid=P.cid  
GROUP BY C.cname
```

No! Different results if a company has no products

```
SELECT C.cname, count(pname)  
FROM Company C LEFT OUTER JOIN Product P  
ON C.cid=P.cid  
GROUP BY C.cname
```



2. Subqueries in FROM

Product (pname, price, cid)

Company(cid, cname, city)

Find all products whose prices is > 20 and < 500

```
SELECT X.pname
FROM (SELECT * FROM Product AS Y WHERE price > 20) as X
WHERE X.price < 500
```

Unnest this query !

```
SELECT pname FROM Product
WHERE price > 20 and price < 500
```




3. Subqueries in WHERE

Product (pname, price, cid)

Company(cid, cname, city)

Find all companies that make some products with price < 200



3. Subqueries in WHERE

Find all companies that make some products with price < 200

Product (pname, price, cid)

Company(cid, cname, city)

Existential quantifiers



3. Subqueries in WHERE

Product (pname, price, cid)

Company(cid, cname, city)

- Find all companies that make some products with price < 200

■ Existential quantifiers

-
- Using **IN**

```
SELECT DISTINCT C.cname
FROM   Company C
WHERE  C.cid IN (SELECT P.cid
                  FROM Product P
                  WHERE P.price < 200)
```



SQL EXISTS Operator

- The EXISTS operator is used to test for the existence of any record in a subquery.
- The EXISTS operator returns true if the subquery returns one or more records.
- **Exists** Syntax:

```
SELECT column_name(s)
FROM table_name
WHERE EXISTS (SELECT column_name
                  FROM table_name
                  WHERE condition);
```



Example 1

ProductID	ProductName	SupplierID	Price
1	Chais	1	18
2	Chang	1	21
3	Syrup	1	10
4	Seasoning	2	22
5	Gumbo	2	19

SupplierID	SupplierName	ContactName	City
1	Exotic Liquid	Charlotte Cooper	LA
2	Cajun Delights	Shelley Burke	NY
3	Homestead	Regina Murphy	SF

```
SELECT DISTINCT SupplierName
FROM Suppliers AS S
WHERE EXISTS (SELECT ProductName
              FROM Products AS P
              WHERE P.SupplierID = S.SupplierID
              AND Price < 20);
```

This SQL statement returns TRUE and lists the suppliers with a product price less than 20



Example 2

ProductID	ProductName	SupplierID	Price
1	Chais	1	18
2	Chang	1	21
3	Syrup	1	10
4	Seasoning	2	22
5	Gumbo	2	19

SupplierID	SupplierName	ContactName	City
1	Exotic Liquid	Charlotte Cooper	LA
2	Cajun Delights	Shelley Burke	NY
3	Homestead	Regina Murphy	SF

```
SELECT DISTINCT SupplierName
FROM Suppliers AS S
WHERE EXISTS (SELECT ProductName
              FROM Products AS P
              WHERE P.SupplierID = S.SupplierID
              AND Price = 22);
```

This SQL statement returns TRUE and lists the suppliers with a product price equal to 22



3. Subqueries in WHERE

Product (pname, price, cid)

Company(cid, cname, city)

- Find all companies that make some products with price < 200

■ Existential quantifiers

- Using **EXISTS**:

```
SELECT DISTINCT C.cname
FROM   Company C
WHERE  EXISTS (SELECT *
               FROM Product P
               WHERE C.cid = P.cid and P.price < 200)
```



3. Subqueries in WHERE

Product (pname, price, cid)

Company(cid, cname, city)

- Find all companies that make some products with price < 200

■ Existential quantifiers

-
- Using **ANY**:

```
SELECT DISTINCT C.cname
FROM   Company C
WHERE  200 > ANY (SELECT price
                   FROM Product P
                   WHERE P.cid = C.cid)
```




3. Subqueries in WHERE

Product (pname, price, cid)

Company(cid, cname, city)

- Find all companies that make some products with price < 200

■ Existential quantifiers

-
- Using **ANY**:

```
SELECT DISTINCT C.cname
FROM   Company C
WHERE  200 > ANY (SELECT price
                  FROM Product P
                  WHERE P.cid = C.cid)
```

Not supported
in MySQL



3. Subqueries in WHERE

Product (pname, price, cid)

Company(cid, cname, city)

- Find all companies that make some products with price < 200

■ Existential quantifiers

-
- Now let's unnest it:

```
SELECT DISTINCT C.cname
FROM   Company C, Product P
WHERE  C.cid= P.cid and P.price < 200
```



3. Subqueries in WHERE

Product (pname, price, cid)

Company(cid, cname, city)

- Find all companies that make some products with price < 200

- Existential quantifiers

-
- Now let's unnest it:

```
SELECT DISTINCT C.cname
FROM   Company C, Product P
WHERE  C.cid= P.cid and P.price < 200
```

Existential quantifiers are easy! 😊



3. Subqueries in WHERE

Find all companies s.t. all their products have price < 200

same as:

Find all companies that make only products with price < 200

Product (pname, price, cid)

Company(cid, cname, city)



3. Subqueries in WHERE

Find all companies s.t. all their products have price < 200

same as:

Find all companies that make only products with price < 200

Product (pname, price, cid)

Company(cid, cname, city)

Universal quantifiers



3. Subqueries in WHERE

Find all companies s.t. all their products have price < 200

same as:

Find all companies that make only products with price < 200

Product (pname, price, cid)

Company(cid, cname, city)

Universal quantifiers

Universal quantifiers are hard! ☹️



3. Subqueries in WHERE

Find all companies s.t. all their products have price < 200

1. Find *the other* companies: i.e. s.t. some product ≥ 200

```
SELECT DISTINCT C.cname
FROM   Company C
WHERE  C.cid IN (SELECT P.cid
                  FROM Product P
                  WHERE P.price  $\geq$  200)
```

Product (pname, price, cid)
Company(cid, cname, city)



3. Subqueries in WHERE

Find all companies s.t. all their products have price < 200

1. Find *the other* companies: i.e. s.t. some product ≥ 200

```
SELECT DISTINCT C.cname
FROM   Company C
WHERE  C.cid IN (SELECT P.cid
                  FROM Product P
                  WHERE P.price >= 200)
```

Product (pname, price, cid)
Company(cid, cname, city)

2. Find all companies s.t. all their products have price < 200

```
SELECT DISTINCT C.cname
FROM   Company C
WHERE  C.cid NOT IN (SELECT P.cid
                     FROM Product P
                     WHERE P.price >= 200)
```




3. Subqueries in WHERE

Find all companies s.t. all their products have price < 200

Product (pname, price, cid)

Company(cid, cname, city)

Universal quantifiers

Using EXISTS:

```
SELECT DISTINCT C.cname
FROM   Company C
WHERE NOT EXISTS (SELECT *
                  FROM Product P
                  WHERE P.cid = C.cid and P.price >= 200)
```



3. Subqueries in WHERE

Find all companies s.t. all their products have price < 200

Product (pname, price, cid)

Company(cid, cname, city)

Universal quantifiers

Using **ALL**:

```
SELECT DISTINCT C.cname
FROM   Company C
WHERE  200 >= ALL (SELECT price
                    FROM Product P
                    WHERE P.cid = C.cid)
```



3. Subqueries in WHERE

Find all companies s.t. all their products have price < 200

Product (pname, price, cid)

Company(cid, cname, city)

Universal quantifiers

Using **ALL**:

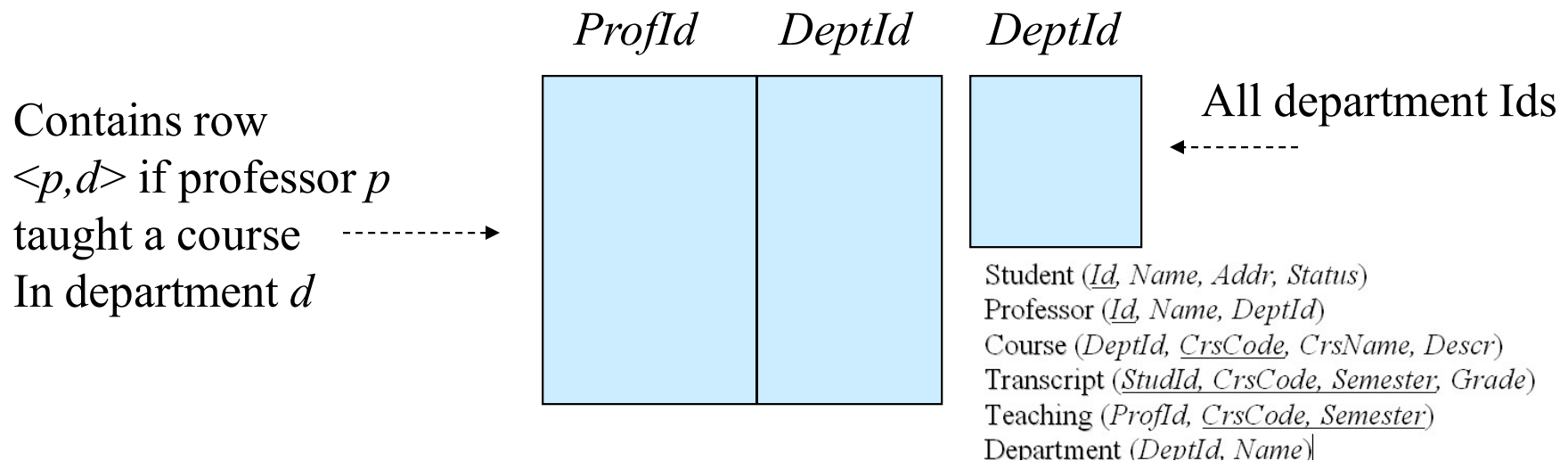
```
SELECT DISTINCT C.cname
FROM   Company C
WHERE  200 > ALL (SELECT price
                  FROM Product P
                  WHERE P.cid = C.cid)
```

Not supported
in MySQL



Division in SQL

- *Query type*: Find the subset of items in one set that are related to *all* items in another set
- *Example*: Find professors who taught courses in *all* departments
 - Why does this involve division?



$$\pi_{ProfId, DeptId}(Teaching \bowtie Course) / \pi_{DeptId}(Department)$$

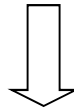


Division Solution Sketch (1)

```
SELECT P.id  
FROM Professor P  
WHERE P taught courses in all departments
```



```
SELECT P.id  
FROM Professor P  
WHERE there does not exist any department that P has  
never taught a course
```

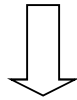


```
SELECT P.id  
FROM Professor P  
WHERE NOT EXISTS (the departments that P has never  
taught a course)
```



Division Solution Sketch (1)

```
SELECT P.Id  
FROM Professor P  
WHERE NOT EXISTS (the departments that P has never  
taught a course)
```



```
SELECT P.Id  
FROM Professor P  
WHERE NOT EXISTS (  
  B: All departments  
  EXCEPT  
  A: the departments that P has ever taught a course)
```

- But how do we formulate A and B?



Division – SQL Solution in details

```
SELECT P.Id
FROM Professor P
WHERE NOT EXISTS
    ( SELECT D.DeptId      -- set B of all dept Ids
      FROM Department D
    EXCEPT
      SELECT C.DeptId      -- set A of dept Ids of depts in which P taught a course
      FROM Teaching T, Course C
      WHERE T.ProfId = P.Id  -- global variable
      AND T.CrsCode = C.CrsCode)
```



Constraints on a Single Relation

- **not null**
- **Default value**
- **unique**
- **check (P)**, where P is a predicate



Not Null Constraints

- **not null**
 - Declare *name* and *budget* to be **not null**
name **varchar(20) not null**
budget **numeric(12,2) not null**



Default Value

-Value to be assigned if attribute value in a row is not specified

```
CREATE TABLE Student (  
    Id INTEGER,  
    Name CHAR(20) NOT NULL,  
    Address CHAR(50),  
    Status CHAR(10) DEFAULT 'freshman',  
    PRIMARY KEY (Id) )
```



Unique Constraints

- **unique** (A_1, A_2, \dots, A_m)
 - The unique specification states that the attributes A_1, A_2, \dots, A_m form a candidate key.
 - Candidate keys are permitted to be null (in contrast to primary keys).



The check clause

- The **check** (P) clause specifies a predicate P that must be satisfied by every tuple in a relation.
- Example: ensure that semester is one of fall, winter, spring or summer

```
create table section
  (course_id varchar (8),
   sec_id varchar (8),
   semester varchar (6),
   year numeric (4,0),
   building varchar (15),
   room_number varchar (7),
   time slot id varchar (4),
   primary key (course_id, sec_id, semester, year),
   check (semester in ('Fall', 'Winter', 'Spring', 'Summer')))
```



Referential Integrity

- Ensures that a value that appears in one relation for a given set of attributes also appears for a certain set of attributes in another relation.
 - Example: If “Biology” is a department name appearing in one of the tuples in the *instructor* relation, then there exists a tuple in the *department* relation for “Biology”.
- Let A be a set of attributes. Let R and S be two relations that contain attributes A and where A is the primary key of S. A is said to be a **foreign key** of R if for any values of A appearing in R these values also appear in S.



Referential Integrity (Cont.)

- Foreign *keys can be* specified as part of the SQL **create table** statement
foreign key (*dept_name*) **references** *department*
- By default, a foreign key references the primary-key attributes of the referenced table.
- SQL allows a list of attributes of the referenced relation to be specified explicitly.

foreign key (*dept_name*) **references** *department*
(*dept_name*)



Cascading Actions in Referential Integrity

- When a referential-integrity constraint is violated, the normal procedure is to reject the action that caused the violation.
- An alternative, in case of delete or update is to cascade

```
create table course (  
    (...  
    dept_name varchar(20),  
    foreign key (dept_name) references department  
        on delete cascade  
        on update cascade,  
    ...)
```

- Instead of cascade we can use :
 - **set null**,
 - **set default**



Integrity Constraint Violation During Transactions

- Consider:

```
create table person (  
    ID char(10),  
    name char(40),  
    mother char(10),  
    father char(10),  
    primary key ID,  
    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 (not possible if father and mother attributes declared to be **not null**)
 - OR defer constraint checking



Assertions

- An **assertion** is a predicate expressing a condition that we wish the database always to satisfy.
- Element of schema (like table)
- Applies to entire database (not just the individual rows of a single table)
 - hence it works even if Employee is empty
- The following constraints, can be expressed using assertions:
- For each tuple in the *student* relation, the value of the attribute *tot_cred* must equal the sum of credits of courses that the student has completed successfully.
- An instructor cannot teach in two different classrooms in a semester in the same time slot
- An assertion in SQL takes the form:
create assertion <assertion-name> **check** (<predicate>);



Assertion Example

```
CREATE ASSERTION DontFireEveryone  
CHECK (0 < SELECT COUNT (*) FROM Employee)
```



Sample

Id	MgrId	EmpName	Salary	StartDate
1111	3333	Kathy	50K	2012
2222	3333	John	60K	2011
3333	0000	Cook	100K	2000
4444	0000	Mathew	75K	2012
5555	1111	Jun	40K	1015

Primary Key (ID),
FOREIGN KEY (MgrId) References Employee (Id)



Assertion

```
CREATE ASSERTION KeepEmployeeSalariesDown
CHECK (NOT EXISTS(
    SELECT * FROM Employee E
    WHERE E.Salary > E.MngrSalary))
```

EXISTS(R) is a boolean function (called predicate)

- Returns true when R is not empty
- Return false otherwise

NOT EXISTS(R) \equiv isEmpty(R) \equiv (R = Φ)



Assertions and Inclusion Dependency

```
CREATE ASSERTION NoEmptyCourses
CHECK (NOT EXISTS (
    SELECT * FROM Teaching T
    WHERE T.roster() =  $\Phi$ )
)
```

Idea: search those courses in Teaching such that they have no registered students.

But how to write $T.roster() = \Phi$ in SQL?



Assertions and Inclusion Dependency

```
CREATE ASSERTION NoEmptyCourses
CHECK (NOT EXISTS (
    SELECT * FROM Teaching T
    WHERE -- for each row T check the following condition
    NOT EXISTS (
        SELECT * FROM Transcript R
        WHERE R.CrsCode = T.CrsCode
        AND R.Semester = T.Semester )
    ))
```

Idea: search those courses in Teaching such that they have no registered students.



User-Defined Types

- **create type** construct in SQL creates user-defined type

create type *Dollars* as numeric (12,2) final

- Example:

```
create table department  
(dept_name varchar (20),  
building varchar (15),  
budget Dollars);
```



Domains

- **create domain** construct in SQL-92 creates user-defined domain types

create domain *person_name* **char(20) not null**

- Types and domains are similar. Domains can have constraints, such as **not null**, specified on them.
- Example:

```
create domain degree_level varchar(10)  
constraint degree_level_test  
check (value in ('Bachelors', 'Masters', 'Doctorate'));
```




Index Creation

- Many queries reference only a small proportion of the records in a table.
- It is inefficient for the system to read every record to find a record with particular value
- An **index** on an attribute of a relation is a data structure that allows the database system to find those tuples in the relation that have a specified value for that attribute efficiently, without scanning through all the tuples of the relation.
- We create an index with the **create index** command
create index <name> **on** <relation-name> (attribute);



Index Creation Example

- **create table** *student*
(*ID* **varchar** (5),
name **varchar** (20) **not null**,
dept_name **varchar** (20),
tot_cred **numeric** (3,0) **default** 0,
primary key (*ID*))
- **create index** *studentID_index* **on** *student*(*ID*)
- The query:
select *
from *student*
where *ID* = '12345'

can be executed by using the index to find the required record, without looking at all records of *student*



Authorization

- We may assign a user several forms of authorizations on parts of the database.
 - **Read** - allows reading, but not modification of data.
 - **Insert** - allows insertion of new data, but not modification of existing data.
 - **Update** - allows modification, but not deletion of data.
 - **Delete** - allows deletion of data.
- Each of these types of authorizations is called a **privilege**. We may authorize the user all, none, or a combination of these types of privileges on specified parts of a database, such as a relation or a view.



Authorization (Cont.)

- Forms of authorization to modify the database schema
 - **Index** - allows creation and deletion of indices.
 - **Resources** - allows creation of new relations.
 - **Alteration** - allows addition or deletion of attributes in a relation.
 - **Drop** - allows deletion of relations.



Authorization Specification in SQL

- The **grant** statement is used to confer authorization
grant <privilege list> **on** <relation or view > **to** <user list>
- <user list> is:
 - a user-id
 - **public**, which allows all valid users the privilege granted
 - A role (more on this later)
- Example:
 - **grant select on department to** Amit, Satoshi
- Granting a privilege on a view does not imply granting any privileges on the underlying relations.
- The grantor of the privilege must already hold the privilege on the specified item (or be the database administrator).



Privileges in SQL

- **select**: allows read access to relation, or the ability to query using the view
 - Example: grant users U_1 , U_2 , and U_3 **select** authorization on the *instructor* relation:

grant select on *instructor* to U_1 , U_2 , U_3

- **insert**: the ability to insert tuples
- **update**: the ability to update using the SQL update statement
- **delete**: the ability to delete tuples.
- **all privileges**: used as a short form for all the allowable privileges



Revoking Authorization in SQL

- The **revoke** statement is used to revoke authorization.
revoke <privilege list> **on** <relation or view> **from** <user list>
- Example:
revoke select on student from U_1, U_2, U_3
- <privilege-list> may be **all** to revoke all privileges the revokee may hold.
- If <revokee-list> includes **public**, all users lose the privilege except those granted it explicitly.
- If the same privilege was granted twice to the same user by different grantees, the user may retain the privilege after the revocation.
- All privileges that depend on the privilege being revoked are also revoked.



Roles

- A **role** is a way to distinguish among various users as far as what these users can access/update in the database.
- To create a role we use:
create a role <name>
- Example:
 - **create role** instructor
- Once a role is created we can assign “users” to the role using:
 - **grant** <role> **to** <users>



Roles Example

- **create role** instructor;
- **grant** *instructor* **to** Amit;
- Privileges can be granted to roles:
 - **grant select on** *takes* **to** *instructor*;
- Roles can be granted to users, as well as to other roles
 - **create role** *teaching_assistant*
 - **grant** *teaching_assistant* **to** *instructor*;
 - *Instructor* inherits all privileges of *teaching_assistant*
- Chain of roles
 - **create role** *dean*;
 - **grant** *instructor* **to** *dean*;
 - **grant** *dean* **to** Satoshi;



View Definition

- A view is defined using the **create view** statement which has the form

create view *v* **as** < query expression >

where <query expression> is any legal SQL expression.
The view name is represented by *v*.

- Once a view is defined, the view name can be used to refer to the virtual relation that the view generates.
- View definition is not the same as creating a new relation by evaluating the query expression
 - Rather, a view definition causes the saving of an expression; the expression is substituted into queries using the view.



View Definition and Use

- A view of instructors without their salary

```
create view faculty as  
    select ID, name, dept_name  
    from instructor
```

- Find all instructors in the Biology department

```
select name  
from faculty  
where dept_name = 'Biology'
```

- Create a view of department salary totals

```
create view departments_total_salary(dept_name, total_salary) as  
    select dept_name, sum (salary)  
    from instructor  
    group by dept_name;
```



View - Substitution

When used in an SQL statement, the view definition is substituted for the view name in the statement. As SELECT statement nested in FROM clause

```
SELECT S.Name, C.Cum
FROM (SELECT T.StudId, AVG (T.Grade)
      FROM Transcript T
      GROUP BY T.StudId) C, Student S
WHERE C.StudId = S.StudId AND C.Cum > 3.5
```



View Benefits

- *Access Control*: Users not granted access to base tables. Instead they are granted access to the view of the database appropriate to their needs.
 - *External schema* is composed of views.
 - View allows owner to provide SELECT access to a subset of columns (analogous to providing UPDATE and INSERT access to a subset of columns)



Views – Limiting Visibility

Grade projected out

```
CREATE VIEW PartOfTranscript (StudId, CrsCode, Semester) AS
  SELECT T. StudId, T. CrsCode, T. Semester    -- limit columns
  FROM Transcript T
  WHERE T. Semester = 'S2000'                -- limit rows
```

Give permissions to access data through view:

```
GRANT SELECT ON PartOfTranscript TO joe
```

This would have been analogous to:

```
GRANT SELECT (StudId, CrsCode, Semester)
              ON Transcript TO joe
```



View Benefits (cont'd)

- *Customization*: Users need not see full complexity of database.
- View creates the illusion of a simpler database customized to the needs of a particular category of users
- A view is *similar in many ways to a subroutine* in standard programming
 - Can be reused in multiple queries



Views Defined Using Other Views

- **create view** *physics_fall_2017* **as**
 select *course.course_id, sec_id, building, room_number*
 from *course, section*
 where *course.course_id = section.course_id*
 and *course.dept_name = 'Physics'*
 and *section.semester = 'Fall'*
 and *section.year = '2017';*
- **create view** *physics_fall_2017_watson* **as**
 select *course_id, room_number*
 from *physics_fall_2017*
 where *building= 'Watson';*



Materialized Views

- Certain database systems allow view relations to be physically stored.
 - Physical copy created when the view is defined.
 - Such views are called **Materialized view**:
- If relations used in the query are updated, the materialized view result becomes out of date
 - Need to **maintain** the view, by updating the view whenever the underlying relations are updated.



Update of a View

- Add a new tuple to *faculty* view which we defined earlier

insert into *faculty*

values ('30765', 'Green', 'Music');

- This insertion must be represented by the insertion into the *instructor* relation

- Must have a value for salary.

- Two approaches

- Reject the insert
- Inset the tuple

('30765', 'Green', 'Music', null)

into the *instructor* relation



Some Updates Cannot be Translated Uniquely

- **create view** *instructor_info* **as**
 select *ID, name, building*
 from *instructor, department*
 where *instructor.dept_name= department.dept_name;*
- **insert into** *instructor_info*
 values ('69987', 'White', 'Taylor');
- Issues
 - Which department, if multiple departments in Taylor?
 - What if no department is in Taylor?



And Some Not at All

- **create view** *history_instructors* **as**
 select *
 from *instructor*
 where *dept_name*= 'History';
- What happens if we insert
 ('25566', 'Brown', 'Biology', 100000)
 into *history_instructors*?



View Updates in SQL

- 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 query does not have a **group** by or **having** clause.



Authorization on Views

- **create view** *geo_instructor* **as**
(**select** *
from *instructor*
where *dept_name* = 'Geology');
- **grant select on** *geo_instructor* **to** *geo_staff*



End of Chapter 4