BLG 317E
DATABASE SYSTEMS

WEEK 5

Slides credit:
S. Shivakumar, CS 145, Stanford University
Database System Concepts, Silberschatz, Korth, Sudarshan

ITU

Subquery in SELECT and FROM clauses

- A subquery in SELECT clause must return a single value.
- A subquery in FROM clause may return a single value, or multiple values (table).
- In general, subqueries execute very slowly, compared to other methods.
- Subqueries should be avoided for performance reasons, if there is another faster solution such as joins.

Subquery in SELECT clause

Query : Find the sum of quantities for each group of products in the SALES table. Sort the result set by sum of quantities, in descending order.

Solution) Correlated Subquery

```
SELECT DISTINCT P1.product_name,
       (SELECT SUM(P2.quantity)
        FROM SALES AS P2
        WHERE P1.product_name = P2.product_name
        ) AS total
FROM SALES  AS P1
ORDER BY total  DESC
```

product_name	total
Orange	121
Cherry	104
Apple	86
Apricot	78
Kiwi	34

Subquery in FROM clause

Query : Find all records whose price field are equal to the minimum price in table.

Solution1) Subquery in **FROM** clause

```
select P1.*
from SALES AS P1,
     (select MIN(price) AS cheapest
      from SALES) AS P2
where P1.price = P2.cheapest ;
```

```
+---------+--------------+-------+----------+-------+
| sale_id | product_name | price | quantity | month |
+---------+--------------+-------+----------+-------+
|     102 | Kiwi         |   0.8 |       34 | Jan   |
+---------+--------------+-------+----------+-------+
```

Solution2) Subquery in **WHERE** clause.

```
select *
from SALES
where price  =
    (select MIN(price)
     from SALES);
```

Same result-set

```
+---------+--------------+-------+----------+-------+
| sale_id | product_name | price | quantity | month |
+---------+--------------+-------+----------+-------+
|     102 | Kiwi         |   0.8 |       34 | Jan   |
+---------+--------------+-------+----------+-------+
```

COMMON TABLE EXPRESSION

Common Table Expression (CTE) (The WITH clause)

- Common Table Expression (CTE) is an alternative way of writing a subquery, that helps simplify a main query.
- A CTE generates a temporary virtual table (with records and columns), during the execution of a query.
- It is a temporary named result-set created from a simple SELECT statement, that can be used in a subsequent SQL statement.
- It is like a named subquery, whose result is stored in a virtual table to be referenced later in the main que

Common Table Expression (CTE)

The WITH clause should be written before the main SQL statement.

Syntax

```
WITH cte_name AS
(
  SELECT ...
)

SELECT columns
FROM   tables
WHERE  condition
```

CTE (subquery)
(It is part of the main query)

Main query

Query : Get a list of all records from SALES table.
Each row in result-set should also contain the **Average Price** information.

```
WITH RESULT AS (
  SELECT AVG(Price) AS avg_price
  FROM SALES
)
SELECT *
FROM SALES, RESULT;
```

sale_id	product_name	price	quantity	month	avg_price
101	Orange	3.0	22	Jan	2.21111
102	Kiwi	0.8	34	Jan	2.21111
103	Orange	1.5	23	Feb	2.21111
104	Apple	2.0	38	Feb	2.21111
105	Cherry	2.7	44	Feb	2.21111
106	Apricot	1.2	78	Feb	2.21111
107	Orange	3.0	76	March	2.21111
108	Cherry	4.1	60	March	2.21111
109	Apple	1.6	48	March	2.21111

CASE Statement in SELECT clause

- The CASE statement can be used as part of the SELECT clause.
- It goes through conditions and returns a value, when the first condition is met.
- It is similar to an if-then-else , or switch statement in C language.
- Once a condition is true, it will stop checking other branches and return the result.
- If no conditions are true, it returns the value in the ELSE clause.
- If there is no ELSE part and no conditions are true, it returns NULL.

Syntax

```
CASE
    WHEN condition1 THEN result1
    WHEN condition2 THEN result2
    WHEN conditionN THEN resultN
    ELSE result
END
```

CASE Statement in SELECT clause

Query : Find all student names, course titles that were taken by students, the grades for each course taken, and also a text explanation of grades.

```
select student_name, course_title, grade,
       CASE
          WHEN grade = 'A'  THEN 'Excellent'
          WHEN grade = 'A-' THEN 'Very good'
          WHEN grade = 'B+' THEN 'Good high'
          WHEN grade = 'B'  THEN 'Good medium'
          WHEN grade = 'B-' THEN 'Good low'
          WHEN grade = 'C+' THEN 'Passed high'
          WHEN grade = 'C'  THEN 'Passed medium'
          WHEN grade = 'C-' THEN 'Passed low'
          ELSE 'Failed'
       END
       AS evaluation

from STUDENT, TAKES, TEACHES, COURSE

where student.student_id  =  takes.student_id  AND
      takes.CRN           =  teaches.CRN       AND
      teaches.course_id   =  course.course_id
order by student_name;
```

Result of query

student_name	course_title	grade	evaluation
Blake	Accounting Methods	C-	Passed low
Blake	Optics Principles	B	Good medium
Bradley	Intro. to Algorithms	A	Excellent
Bradley	Robotics	B	Good medium
Brown	Accounting Methods	A	Excellent
Brown	World History	A	Excellent
Brown	Web Design	B	Good medium
Chavez	Molecular Biology	C+	Passed high
Clark	Database System Concepts	A	Excellent
Clark	Image Processing	A-	Very good
Hudson	Intro. to Biology	C	Passed medium
Patel	Database System Concepts	C	Passed medium
Patel	Music Production	A	Excellent
Patel	Optics Principles	A	Excellent
Patel	Image Processing	A	Excellent
Sanchez	Image Processing	A-	Very good
Walker	Database System Concepts	C	Passed medium
Walker	World History	B	Good medium
Williams	Accounting Methods	A-	Very good
Williams	Music Production	B+	Good high
Young	Intro. to Digital Systems	B-	Good low

Using INSERT command and SELECT command

- In the following example, the SELECT query replaces the VALUES keyword of the INSERT INTO command.
- The data from the result-set of SELECT command are added to the new table called student_course.
- In general, this method is used to build **temporary tables** only.

```
create table student_course
      (student_name  varchar(20),
       course_title  varchar(30),
       grade         varchar(2));
```

Using INSERT command and SELECT command

```
insert into student_course
select student_name, course_title, grade
from STUDENT, TAKES, TEACHES, COURSE
where student.student_id  = takes.student_id AND
      takes.CRN           = teaches.CRN AND
      teaches.course_id   = course.course_id
order by student_name;
```

The statement below drops (deletes) the temporary table from database.

```
DROP TABLE  student_course;
```

Table student_course

student_name	course_title	grade
Blake	Accounting Methods	C-
Blake	Optics Principles	B
Bradley	Intro. to Algorithms	A
Bradley	Robotics	B
Brown	Accounting Methods	A
Brown	World History	A
Brown	Web Design	B
Chavez	Molecular Biology	C+
Clark	Database System Concepts	A
Clark	Image Processing	A-
Hudson	Intro. to Biology	C
Patel	Database System Concepts	C
Patel	Music Production	A
Patel	Optics Principles	A
Patel	Image Processing	A
Sanchez	Image Processing	A-
Walker	Database System Concepts	C
Walker	World History	B
Williams	Accounting Methods	A-
Williams	Music Production	B+
Young	Intro. to Digital Systems	B-

NULL VALUES IN SQL

NULLs in SQL

- Whenever we don't have a value, we can put a NULL
- Can mean many things:
  - Value does not exists
  - Value exists but is unknown
  - Value not applicable
  - Etc.
- The schema specifies for each attribute if it can be null (*nullable* attribute) or not
- How does SQL cope with tables that have NULLs?

NULLs in SQL

- *For numerical operations,* NULL -> NULL:
  - If x = NULL then **4*(3-x)/7** is still NULL

- *For boolean operations,* in SQL there are three values:

FALSE	**=**	**0**
UNKNOWN	**=**	**0.5**
TRUE	**=**	**1**

  - If x= NULL then x="Joe" is UNKNOWN

NULLs in SQL

- C1 AND C2 = min(C1, C2)
- C1 OR C2 = max(C1, C2)
- NOT C1 = 1 – C1

```
SELECT *
FROM   Person
WHERE (age < 25)
  AND (height > 6 AND weight > 190);
```

Won't return e.g. (age=20 height=NULL weight=200)!

Rule in SQL: include only tuples that yield TRUE (1.0)

NULLs in SQL

Unexpected behavior:

```
SELECT *
FROM   Person
WHERE  age < 25 OR age >= 25
```

Some Persons are not included !

NULLs in SQL

Can test for NULL explicitly:

- x IS NULL
- x IS NOT NULL

```
SELECT *
FROM   Person
WHERE  age < 25 OR age >= 25
   OR age IS NULL;
```

Now it includes all Persons!

NULLs in SQL

Query : Find departments which have no building information.

```
select * from DEPARTMENT
where building IS NULL;
```

```
+---------+-----------+----------+
| dept_id | dept_name | building |
+---------+-----------+----------+
| SPR     | Sports    | NULL     |
+---------+-----------+----------+
```

Structured Query Language (SQL)

Outer Joins, Aggregations, Built-in Functions, Views, Set Operations

Topics

- Subqueries
- Common Table Expression
- Alternative Join Operations
- Set Operations

Alternative JOIN Operators

The INNER and OUTER keywords are optional.

Explicit JOIN operators	Description
[INNER] JOIN	• Returns rows when the join condition is satisfied in both tables. • In other words, it returns only those records that match the join condition in both tables.
LEFT [OUTER] JOIN	• Returns all rows from the left table, even if no matching rows have been found in the right table. • If there are no matches in the right table, the query will return NULL values for those columns.
RIGHT [OUTER] JOIN	• Returns all rows from the right table. • If there are no matches in the left table, NULL values are returned for those columns.

Alternative JOIN Operators

Explicit JOIN operators	Description
`FULL JOIN`	• It is a combination of LEFT JOIN and RIGHT JOIN. • It returns all rows from both tables. • Where no match is found in either the right or left table, it returns NULLs for those columns. • In other words, it is the union of columns of the two tables.
`CROSS JOIN`	• Referred to as a Cartesian JOIN, returns every possible combination of rows from the tables that have been joined. • Since it returns all combinations, it is the only JOIN that does not need a join condition and therefore does not have an ON clause.

Explicit Inner Join Syntax

- The join is an SQL operation that links **two tables** in a SELECT statement.
- The join **condition** of the **ON** clause is the equality between the primary key columns in one table and columns referring to them in the other table (foreign key columns).
- The JOIN condition doesn't have to be an equality.
- It may also contain more than one condition with AND/OR operators.
- The following is the general syntax of explicit JOIN commands.

Syntax 1 : ON clause

```
SELECT   columns
FROM     LeftTable
JOIN     RightTable
   ON    LeftTable.foreign_key = RightTable.primary_key
```

Syntax 2 : USING clause

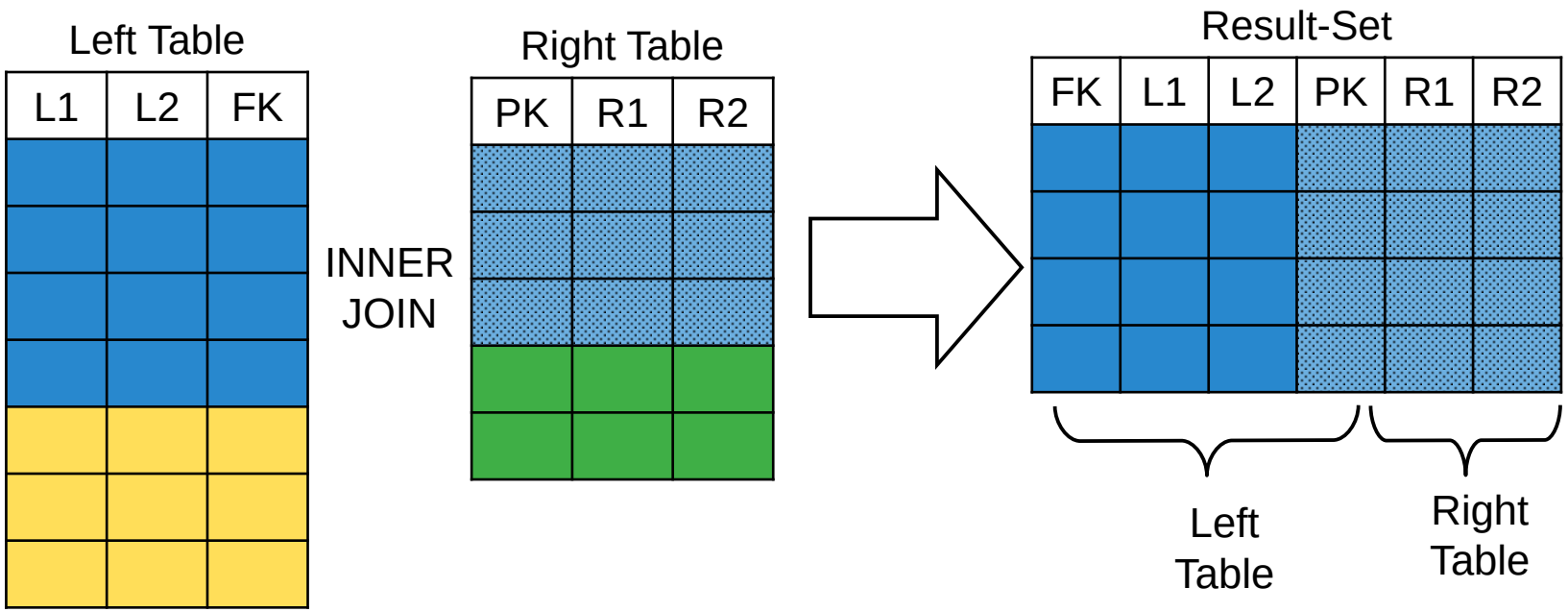
```
SELECT   columns
FROM     LeftTable
JOIN     RightTable
  USING (name_of_common_key)
```

Applies only if the names of foreign key and primary key is same.

INNER JOIN

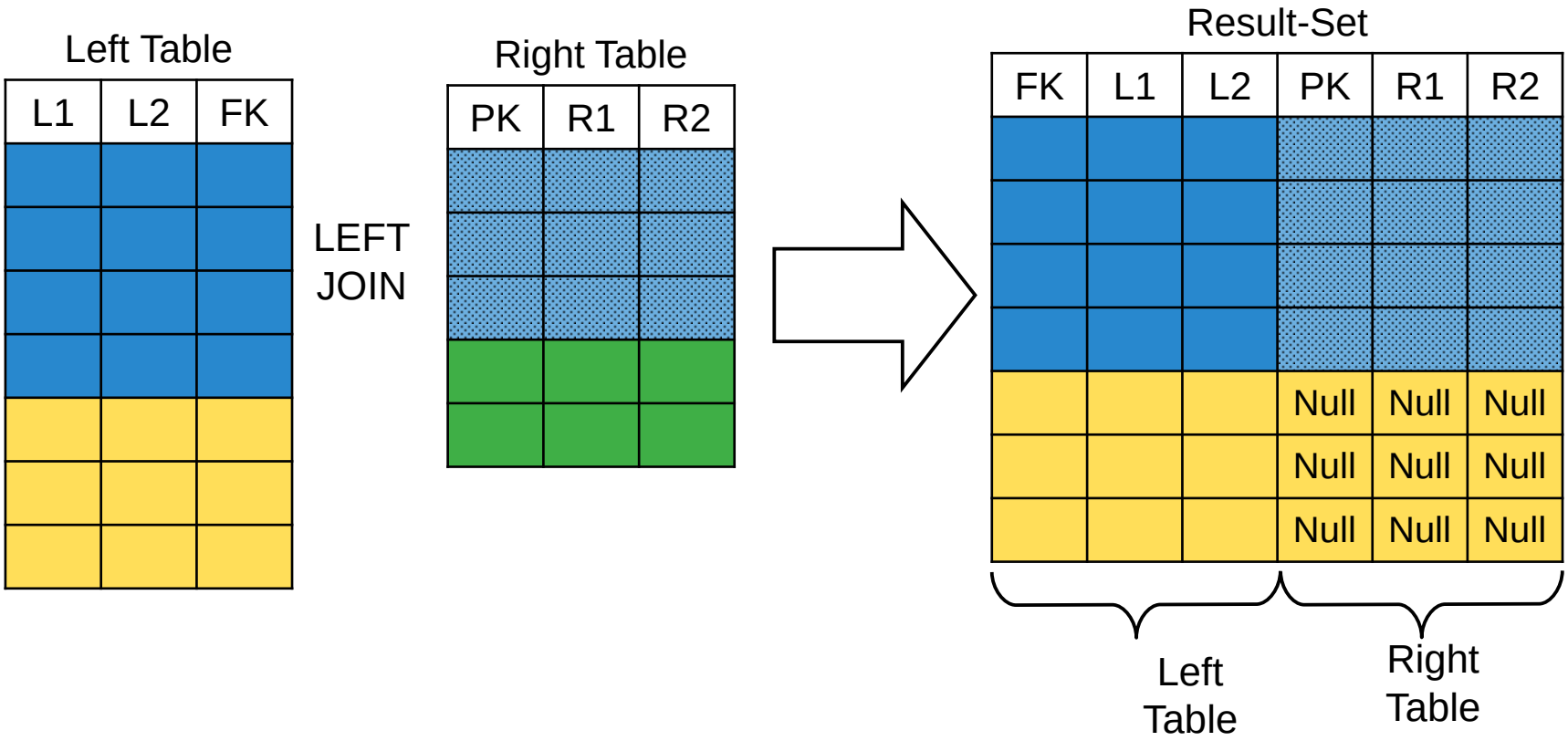
Color explanation:

- The records that have the **same color** on both tables means their key fields are equal (Primary Key and Foreign Key).
- The join operation dependends on the equal key fields.
- INNER JOIN takes only records from both tables, whose keys are equal (matched).
- Records that share the same Key value are matched.
- **Inner join:** Unmatched rows are excluded.
- L1, L2 : Left table columns
- R1, R2 : Right table columns
- PK, FK : Columns of common keys (Primary key and Foreign key)



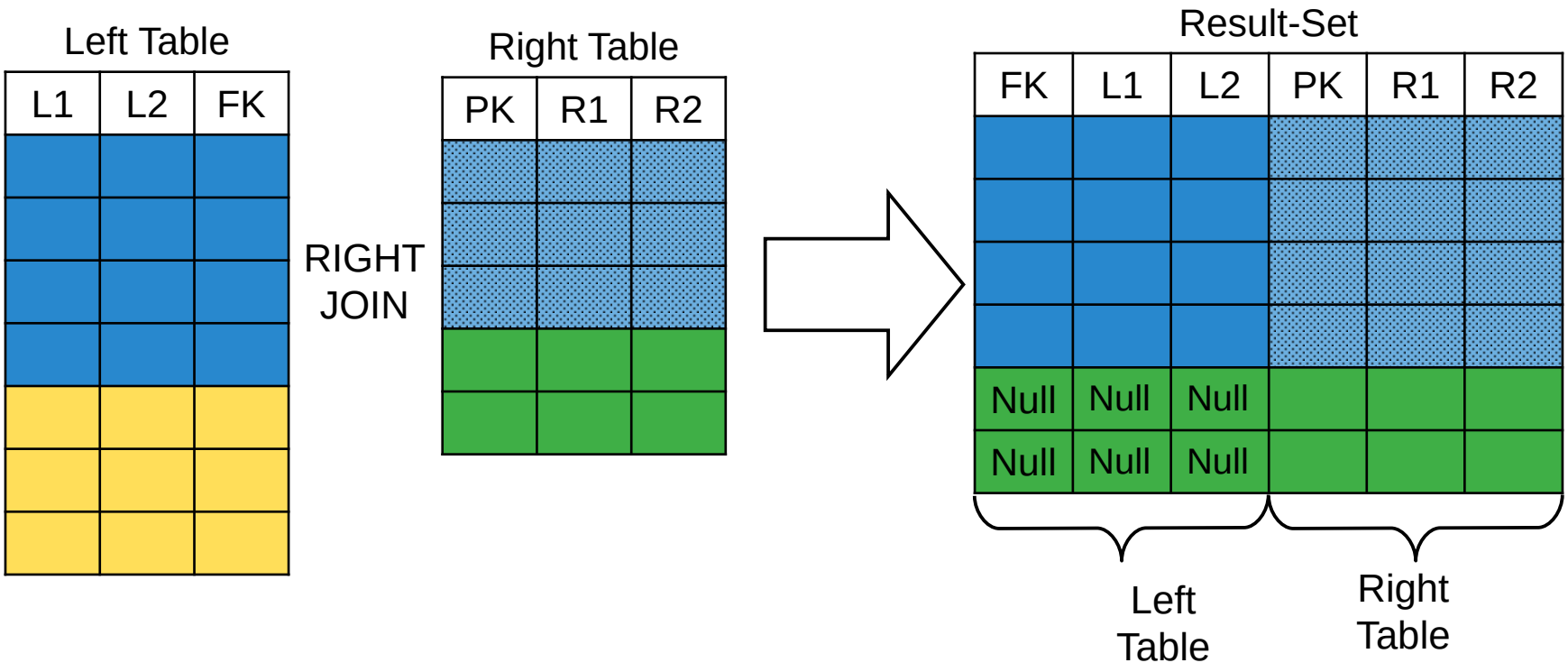
LEFT OUTER JOIN

- LEFT JOIN takes all records from left table (either matched or not matched).
- For unmatched records, NULL is filled in place of right table columns.
- **Outer join:** Unmatched rows are included, columns from the other table are null.



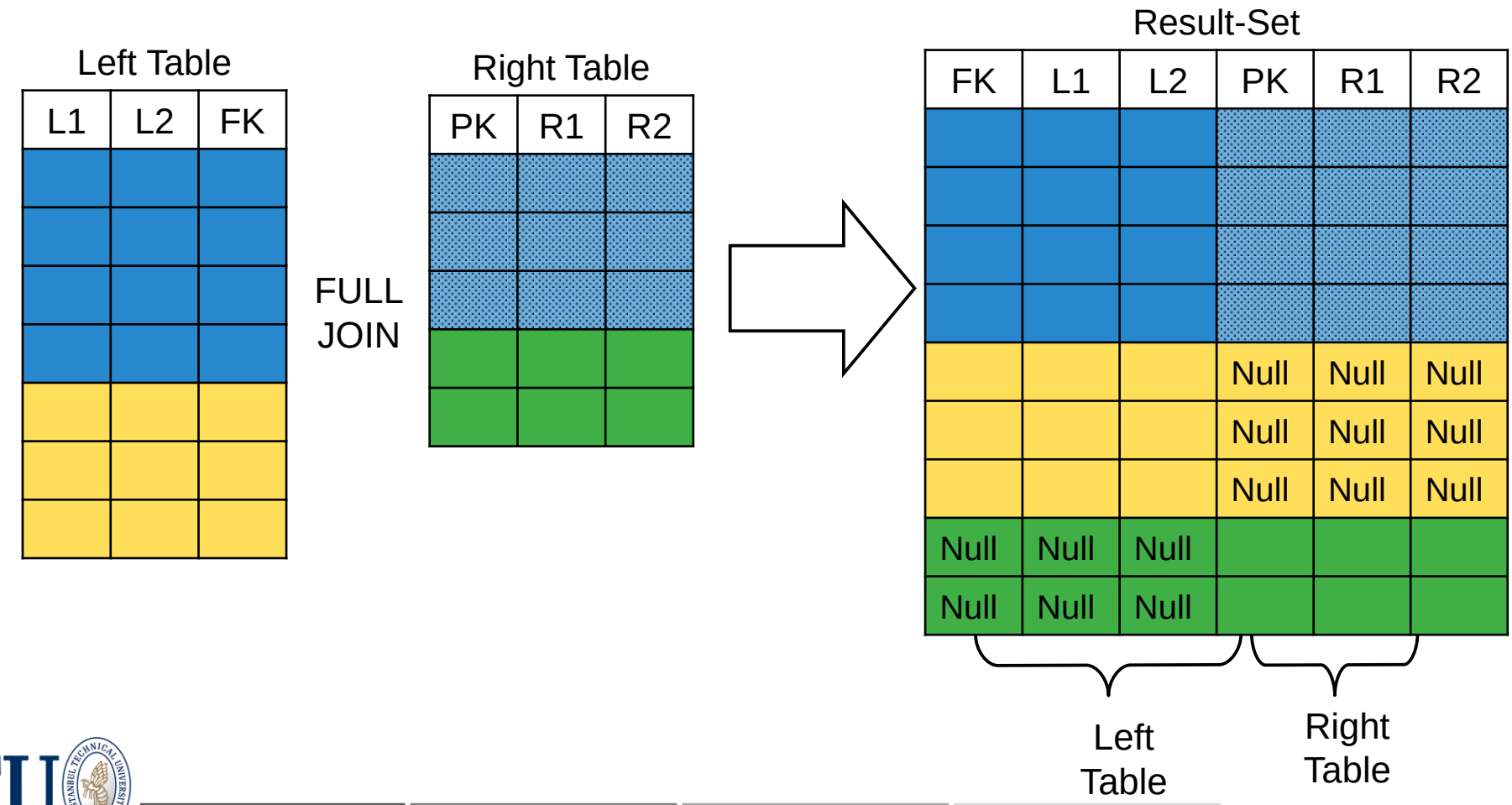
RIGHT OUTER JOIN

- RIGHT JOIN takes all records from right table (either matched or not matched).
- For unmatched records, NULL is filled in place of left table columns.



FULL OUTER JOIN

- FULL JOIN takes all records from both tables (either matched or not matched).
- For unmatched records, NULL is filled in place of right table columns, and NULL is filled in place of left table columns.
- Not available in MySql



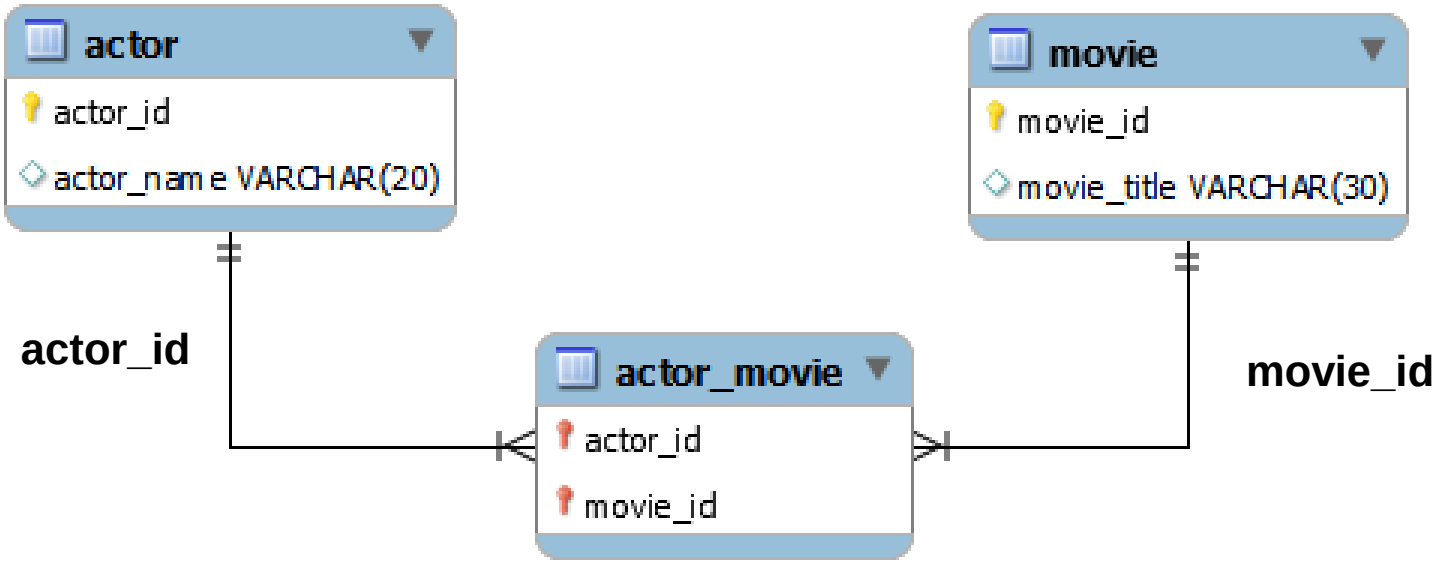
Implicit Inner Join Operation with WHERE clause

- **The Implicit Inner Join** is the most common join operation method.
- It is a basic join with a WHERE clause that contains a join condition.
- The **WHERE** clause of a SELECT command **implicitly** performs an Inner Join.

Syntax

```
SELECT  columns
FROM    Table1, Table2
WHERE   join_condition
```

Example: MOVIES Database



Create Commands

```
create database MOVIES;
use MOVIES;

create table actor(
  actor_id int,
  actor_name varchar(20),
  primary key (actor_id)
 );


create table movie(
  movie_id int,
  movie_title varchar(30),
  primary key (movie_id)
 );


create table actor_movie(
  actor_id int,
  movie_id int,
  primary key (actor_id, movie_id),
  FOREIGN KEY (actor_id) REFERENCES actor (actor_id),
  FOREIGN KEY (movie_id) REFERENCES movie (movie_id)
 );
```

Insert Commands

```
insert into actor values
  (10, 'Olivier'),
  (20, 'Tracy'),
  (30, 'Gable'),
  (40, 'Nicholson'),
  (50, 'Fawcett'),
  (60, 'Tracy'),
  (70, 'Wahlberg'),
  (80, 'Guiness');
```

```
insert into movie values
  (1, 'Garden Island'),
  (2, 'Charade Duffel'),
  (3, 'Ridgemont Submarine'),
  (4, 'Atlantis Cause'),
  (5, 'Fargo Sandhi'),
  (6, 'Music Boondock'),
  (7, 'League Virtual'),
  (8, 'Rugrats Shakespeare'),
  (9, 'Flying Hook');
```

```
insert into actor_movie values
  (10, 1),
  (10, 6),
  (20, 2),
  (30, 3),
  (40, 4),
  (50, 1),
  (60, 4);
```

Table Assumptions

The followings are assumptions for the example database.

- Some actors do not have any movies.
- Some movies do not have any actors.
- Some movies have multiple actors.
- Some actors have multiple movies.

Implicit Inner Join Operation with WHERE clause

Query1 : Get a list of actor names and their movies information (movie ID and movie title). Sort the result-set.

```
select actor_name, movie.movie_id, movie_title
from actor, actor_movie, movie
where actor.actor_id = actor_movie.actor_id and
      actor_movie.movie_id = movie.movie_id
order BY actor_name, movie.movie_id;
```

```
+------------+----------+---------------------+
| actor_name | movie_id | movie_title         |
+------------+----------+---------------------+
| Fawcett    |        1 | Garden Island       |
| Gable      |        3 | Ridgemont Submarine |
| Nicholson  |        4 | Atlantis Cause      |
| Olivier    |        1 | Garden Island       |
| Olivier    |        6 | Music Boondock      |
| Tracy      |        2 | Charade Duffel      |
| Tracy      |        4 | Atlantis Cause      |
+------------+----------+---------------------+
```

The = operators in WHERE clause are used as implicit inner join operators (Equi-Join).

Inner JOIN Operation (2 tables)

Query2 : Get a list of actor names and their movie ID information only. Sort the result-set.

```
select actor_name, actor_movie.movie_id
FROM actor
JOIN actor_movie
     ON actor.actor_id = actor_movie.actor_id
order BY actor_name, actor_movie.movie_id;
```

```
+------------+----------+
| actor_name | movie_id |
+------------+----------+
| Fawcett    |        1 |
| Gable      |        3 |
| Nicholson  |        4 |
| Olivier    |        1 |
| Olivier    |        6 |
| Tracy      |        2 |
| Tracy      |        4 |
+------------+----------+
```

Explicit JOIN operator is used.

Alternative method:

- The USING clause is written, instead of the ON clause.
- It is valid because the actor_id field is common key name in two tables.

```
select actor_name, actor_movie.movie_id
FROM actor
JOIN actor_movie
     USING (actor_id)
order BY actor_name, actor_movie.movie_id;
```

Same result

actor_name	movie_id
Fawcett	1
Gable	3
Nicholson	4
Olivier	1
Olivier	6
Tracy	2
Tracy	4

Inner JOIN Operation (3 tables)

Query3 : Get a list of actor names and their movies information (movie ID and movie title). Sort the result-set.

```
SELECT actor_name, movie.movie_id, movie_title
FROM actor
JOIN actor_movie
     ON actor.actor_id = actor_movie.actor_id
JOIN movie
     ON movie.movie_id = actor_movie.movie_id
ORDER BY actor_name, movie.movie_id;
```

Joining more than two tables:
First, two tables are joined.
Then, the third table is joined to the result of the previous join.

actor_name	movie_id	movie_title
Fawcett	1	Garden Island
Gable	3	Ridgemont Submarine
Nicholson	4	Atlantis Cause
Olivier	1	Garden Island
Olivier	6	Music Boondock
Tracy	2	Charade Duffel
Tracy	4	Atlantis Cause

Alternative method:

- The USING clause is written, instead of the ON clause.
- It is valid because the actor_id field is common key name in two tables.

```
SELECT actor_name, movie.movie_id, movie_title
FROM actor
JOIN actor_movie
     USING (actor_id)
JOIN movie
     USING (movie_id)
ORDER BY actor_name, movie.movie_id;
```

Same result

```
+------------+----------+---------------------+
| actor_name | movie_id | movie_title         |
+------------+----------+---------------------+
| Fawcett    |        1 | Garden Island       |
| Gable      |        3 | Ridgemont Submarine |
| Nicholson  |        4 | Atlantis Cause      |
| Olivier    |        1 | Garden Island       |
| Olivier    |        6 | Music Boondock      |
| Tracy      |        2 | Charade Duffel      |
| Tracy      |        4 | Atlantis Cause      |
+------------+----------+---------------------+
```

LEFT JOIN Operation (2 tables)

Query4 : Get a list of all actor names and their movie ID numbers.
Result-set should contain also the actors that have no movies.

```
select actor_name, actor_movie.movie_id
FROM actor
LEFT JOIN actor_movie
     ON actor.actor_id = actor_movie.actor_id
order by actor_name, actor_movie.movie_id;
```

Left table : actor

Right table : actor_movie

actor_name	movie_id
Fawcett	1
Gable	3
Guiness	NULL
Nicholson	4
Olivier	1
Olivier	6
Tracy	2
Tracy	4
Wahlberg	NULL

RIGHT JOIN Operation (2 tables)

Query5a : Instead of LEFT JOIN used in Query4, the same result can be obtained with RIGHT JOIN.

```
select actor_name, actor_movie.movie_id
FROM actor_movie
RIGHT JOIN actor
     ON actor.actor_id = actor_movie.actor_id
order by actor_name, actor_movie.movie_id;
```

Left table : actor_movie
Right table : actor

actor_name	movie_id
Fawcett	1
Gable	3
Guiness	NULL
Nicholson	4
Olivier	1
Olivier	6
Tracy	2
Tracy	4
Wahlberg	NULL

Same result with Query4

Query5b : Same as **Query5a**.
But it filters the result-set with the WHERE clause.
Only the actors whose names starts with 'G'
are listed in the result-set.

```
SELECT actor_name, actor_movie.movie_id
FROM actor_movie
RIGHT JOIN actor
      ON actor.actor_id = actor_movie.actor_id
WHERE actor_name LIKE 'G%'
ORDER BY actor_name, actor_movie.movie_id;
```

```
+------------+----------+
| actor_name | movie_id |
+------------+----------+
| Gable      |        3 |
| Guiness    |     NULL |
+------------+----------+
```

LEFT JOIN Operation (2 tables) and COUNT Aggregation Function

The **Query4** can be extended, so that aggregation functions can be used.

Query6a : Get a list of all actor names and their movie **counts**.
Result-set should contain also the actors whose movies count is zero.

```
select actor_name,
       COUNT(actor_movie.movie_id) AS MovieSayisi
FROM actor
LEFT JOIN actor_movie
     ON actor.actor_id = actor_movie.actor_id
GROUP BY actor_name
order by actor_name;
```

```
+------------+-------------+
| actor_name | MovieSayisi |
+------------+-------------+
| Fawcett    |           1 |
| Gable      |           1 |
| Guiness    |           0 |
| Nicholson  |           1 |
| Olivier    |           2 |
| Tracy      |           2 |
| Wahlberg   |           0 |
+------------+-------------+
```

Query6b : Similar to **Query6a**. This query filters the result-set with the HAVING clause after the GROUP BY clause. Only the actors who have at least 2 movies are listed in the result-set.

```
SELECT actor_name,
       COUNT(actor_movie.movie_id) AS movie_count
FROM actor
LEFT JOIN actor_movie
     ON actor.actor_id = actor_movie.actor_id
GROUP BY actor_name
HAVING  movie_count >= 2
ORDER BY actor_name;
```

```
+------------+-------------+
| actor_name | movie_count |
+------------+-------------+
| Olivier    |           2 |
| Tracy      |           2 |
+------------+-------------+
```

LEFT JOIN Operation (3 tables)

Query7 : Get a list of actor names and their movies information (movie ID and movie title). Result-set should contain also the actors that have no movies.

```
select actor_name, movie.movie_id, movie_title
FROM actor
LEFT JOIN actor_movie
     ON actor.actor_id = actor_movie.actor_id
LEFT JOIN movie
     ON movie.movie_id = actor_movie.movie_id
order BY actor_name, movie.movie_id;
```

actor_name	movie_id	movie_title
Fawcett	1	Garden Island
Gable	3	Ridgemont Submarine
Guiness	NULL	NULL
Nicholson	4	Atlantis Cause
Olivier	1	Garden Island
Olivier	6	Music Boondock
Tracy	2	Charade Duffel
Tracy	4	Atlantis Cause
Wahlberg	NULL	NULL

RIGHT JOIN Operation (3 tables)

Query8 : Get a list of movie information, with the actor names.
Result-set should contain also the movies that have no actors info.

```
SELECT movie.movie_id, movie_title, actor_name
FROM actor_movie
RIGHT JOIN actor
     ON actor_movie.actor_id = actor.actor_id
RIGHT JOIN movie
     ON movie.movie_id = actor_movie.movie_id
ORDER BY movie.movie_id, actor_name;
```

movie_id	movie_title	actor_name
1	Garden Island	Fawcett
1	Garden Island	Olivier
2	Charade Duffel	Tracy
3	Ridgemont Submarine	Gable
4	Atlantis Cause	Nicholson
4	Atlantis Cause	Tracy
5	Fargo Sandhi	NULL
6	Music Boondock	Olivier
7	League Virtual	NULL
8	Rugrats Shakespeare	NULL
9	Flying Hook	NULL

CROSS JOIN Operation

- **CROSS JOIN** operation is also known as **Cartesian Join.**
- The Cartesian product of two tables is another table consisting of all possible pairs of rows from the two tables.
- The columns of the product table (result-set of SELECT query) are all the columns of the first table followed by all the columns of the second table.
- If WHERE clause is not specified, SQL produces the Cartesian product of the two tables as the query result.
- Assume left table has N rows, right table has M rows.
  Then the result-set will be NxM rows.
- The ON clause (condition) is not used with CROSS JOIN command.
- The result-set actually does not contain meaningful information.

Example Tables

```
DEPARTMENT (dept_id   varchar(5),
            dept_name varchar(20) )

STUDENT (student_id    int,
         student_name  varchar(20),
         dept_id       varchar(5) )
```

DEPARTMENT table

dept_id	dept_name
BIO	Biology
CS	Computer
EE	Electronics

3 rows

STUDENT table

student_id	student_name	dept_id
601	Brown	CS
602	Clark	CS
603	Hudson	CS
604	Murphy	EE
605	Williams	EE

5 rows

CROSS JOIN Operation (2 tables)

Query9 : Get a list of all possible combinations of records from the DEPARTMENT table and the STUDENT table.

Explicit CROSS JOIN command
(ON clause not used)

```
select *
FROM department
CROSS JOIN student
order by department.dept_id, student_name;
```

Alternative method : Implicit cross join
(WHERE clause not used)

```
select *
from  department, student
order by department.dept_id,
      student_name;
```

Result of CROSS JOIN query

(All STUDENT records are repeated in all DEPARTMENT records)

dept_id	dept_name	student_id	student_name	dept_id
BIO	Biology	601	Brown	CS
BIO	Biology	602	Clark	CS
BIO	Biology	603	Hudson	CS
BIO	Biology	604	Murphy	EE
BIO	Biology	605	Williams	EE
CS	Computer	601	Brown	CS
CS	Computer	602	Clark	CS
CS	Computer	603	Hudson	CS
CS	Computer	604	Murphy	EE
CS	Computer	605	Williams	EE
EE	Electronics	601	Brown	CS
EE	Electronics	602	Clark	CS
EE	Electronics	603	Hudson	CS
EE	Electronics	604	Murphy	EE
EE	Electronics	605	Williams	EE

3x5 = 15 rows

Self Join

- The self join operation joins data from the same table.
- In other words, it joins a table with itself.
- Records taken from the table are matched to other records from the same table.
- There is no dedicated JOIN operator for self join.
- Instead, the SQL self join uses one of the JOIN operators.
- The difference is that a single table is listed as both the left and right table in the join.

Self Join Example1

- Suppose the WORKERS table stores worker ID numbers, worker names, and their manager's ID number.
- Everyone has a manager, with the exception of the boss.
- The boss can have a NULL value in the manager_id column.

```
WORKERS (worker_id   int primary key,
         worker_name  varchar(20),
         manager_id   int)
```

WORKERS table

worker_id	worker_name	manager_id
1	Watson	null
2	Brown	1
3	James	1
4	Miller	3
5	Davis	3
6	Smith	3

Query10 : Get a list of worker id numbers, worker names, and the name of his/her manager. Result-set should contain the boss who does not have a manager. Sort the results by worker names.

```
SELECT W1.worker_id,
       W1.worker_name,
       W2.worker_name  AS  manager_name
FROM Workers W1
LEFT JOIN Workers W2
     ON  W1.manager_id = W2.worker_id
order by W1.worker_name;
```

Result of self join query

worker_id	worker_name	manager_name
2	Brown	Watson
5	Davis	James
3	James	Watson
4	Miller	James
6	Smith	James
1	Watson	NULL

- The WORKERS table above is used as both the left and right table in the **LEFT JOIN**.
- To join data from the same table, it is necessary to assign two aliases (**W1** and **W2**) to the table name.
- When performing self joins, using table aliases is compulsory.
- Because two columns from the same table is joined, they will have the same names.
- The table alias renames the columns so the DBMS can execute the query.

Self Join Example2

- Suppose the PERSONS table contains id numbers and names of persons.
- The id field is unique, but the name field is not.

```
PERSONS (id   int   unique,
         name  varchar(20) )
```

PERSONS table

id	name
101	Laurence
102	Thompson
103	Smith
104	Laurence
105	Smith
106	Smith

Query11 : Find the persons who have the same names in table.
The result-set should contain all pairs of records that have duplicated (repeated) names.

```
SELECT P1.id AS id1, P1.name AS name1,
       P2.id AS id2, P2.name AS name2
FROM PERSONS P1
JOIN PERSONS P2
     ON P1.name = P2.name AND
        P1.id < P2.id ;
```

The condition (P1.id < P2.id) is used to skip identical records from both tables, as well as the same pairs of records in reverse order (symmetrical).

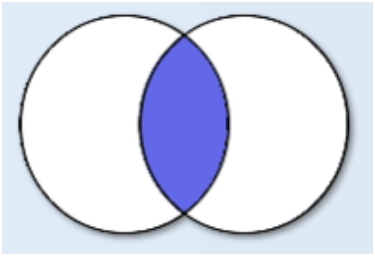
Result of self join query

id1	name1	id2	name2
101	Laurence	104	Laurence
103	Smith	105	Smith
103	Smith	106	Smith
105	Smith	106	Smith

Join Operations (1)

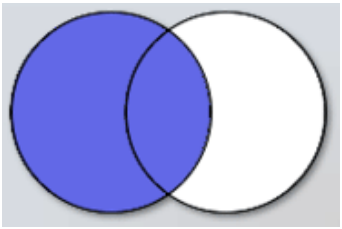
Assumptions:
T1: Left table
T2: Right table
T1.KEY : Foreign key
T2.KEY : Primary key

INNER JOIN



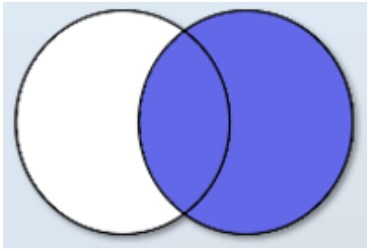
```
SELECT *
FROM T1
INNER JOIN T2
ON T1.KEY = T2.KEY
```

LEFT JOIN



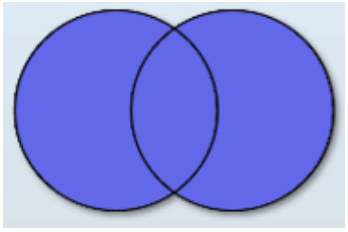
```
SELECT *
FROM T1
LEFT JOIN T2
ON T1.KEY = T2.KEY
```

RIGHT JOIN



```
SELECT *
FROM T1
RIGHT JOIN T2
ON T1.KEY = T2.KEY
```

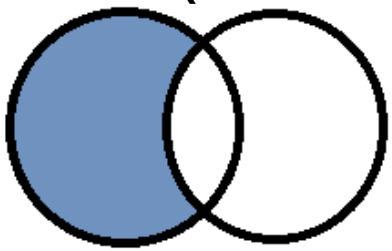
Full Join



```
SELECT * FROM T1 LEFT JOIN T2
UNION
SELECT * FROM T1 RIGHT JOIN T2
```

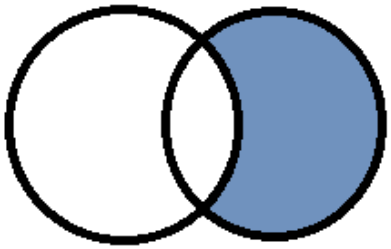
Join Operations (2)

LEFT JOIN (with exclusion)



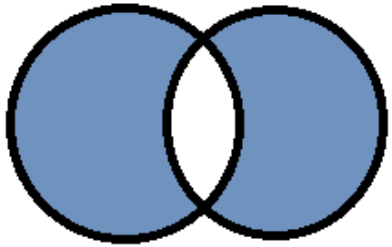
```
SELECT * FROM T1 LEFT JOIN T2
ON T1.KEY = T2.KEY
WHERE T2.KEY IS NULL
```

RIGHT JOIN (with exclusion)



```
SELECT * FROM T1 RIGHT JOIN T2
ON T1.KEY = T2.KEY
WHERE T1.KEY IS NULL
```

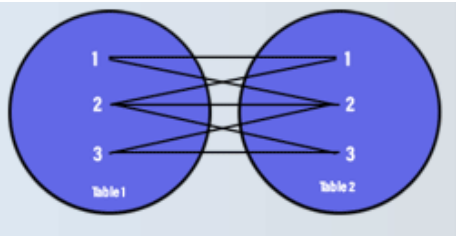
Full Join (with exclusion)



```
SELECT * FROM T1 LEFT JOIN T2
WHERE T2.KEY IS NULL
UNION
SELECT * FROM T1 RIGHT JOIN T2
WHERE T1.KEY IS NULL
```

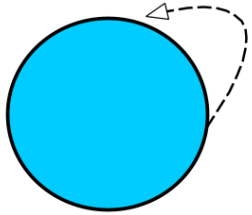
Join Operations (3)

Cross Join




```
SELECT *
FROM T1, T2;
```

Self Join



```
SELECT *
FROM T1  A
LEFT JOIN T1  B
ON A.KEY = B.KEY
```

Topics

- Table Joins (Equi-Join)
- Views
- Aggregation Functions
- Set Operations

Aggregation Functions

- Aggrate functions are used to compute summaries (aggregates) of data.
- They compute a summary value over non-NULL values.
- COUNT(*) computes the number of rows including the NULL values.
- They can be used in SELECT statement.

Syntaxes

```
COUNT (*)
COUNT ( [DISTINCT] Attribute)
SUM ( [DISTINCT] Attribute)
AVG ( [DISTINCT] Attribute)
MIN (Attribute)
MAX (Attribute)
```

Example table : SALES

```
create table SALES(sale_id       int,
                   product_name  varchar(15),
                   price         NUMERIC(5, 1),
                   quantity      int,
                   month         varchar(10) );

insert into SALES values
  (101, 'Orange',  3.0,   22, 'Jan'),
  (102, 'Kiwi',    0.8,   34, 'Jan'),
  (103, 'Orange',  1.5,   23, 'Feb'),
  (104, 'Apple',   2.0,   38, 'Feb'),
  (105, 'Cherry',  2.7,   44, 'Feb'),
  (106, 'Apricot', 1.2,   78, 'Feb'),
  (107, 'Orange',  3.0,   76, 'March'),
  (108, 'Cherry',  4.1,   60, 'March'),
  (109, 'Apple',   1.6,   48, 'March') ;
```

COUNT Function

Query1 : Find how many records are there in the SALES table.

```
select  COUNT(*)
from    SALES;
```

Result

```
+----------+
| COUNT(*) |
+----------+
|        9 |
+----------+
```

COUNT Function

Query2 : Find how many distinct products are there in the table.

Version1:

```
select COUNT(Product_Name)
from SALES;
```

Result

```
+--------------------+
|COUNT(product_name)|
+--------------------+
|                   9 |
+--------------------+
```

Version2: Using column alias.

```
select
   COUNT(distinct Product_name)
   AS "product_type_cnt"
from SALES;
```

Result

```
+-----------------+
|product_type_cnt |
+-----------------+
|                5 |
+-----------------+
```

SUM Function

Query3 : Find total spending amount of SALES for all records.

```
select  SUM(Price * Quantity)
from    SALES;
```

Result

```
+-------------------------+
| SUM(Price * Quantity) |
+-------------------------+
|                   966.9 |
+-------------------------+
```

SUM Function

Query4 : Find total amount of SALES for the 'Orange' product only.

```
select  SUM(Price * Quantity)
from    SALES
where   Product_name = 'Orange';
```

Result

```
+-----------------------+
| SUM(Price * Quantity) |
+-----------------------+
|                 328.5 |
+-----------------------+
```

AVG, MIN, MAX Functions

Query5 : Find count, sum, average, minimum, maximum of all prices.

```
select  COUNT(Price),
        SUM(Price),
        AVG(Price),
        MIN(Price),
        MAX(Price)
from    SALES;
```

Result

COUNT(Price)	SUM(Price)	AVG(Price)	MIN(Price)	MAX(Price)
9	19.9	2.21111	0.8	4.1

AVG, MIN, MAX Functions

Query6 : Find count, average, minimum, maximum prices for the month 'March' only.

```
select  COUNT(Price),
        AVG(Price),
        MIN(Price),
        MAX(Price)
from    SALES
where   Month = 'March';
```

Result

COUNT(Price)	AVG(Price)	MIN(Price)	MAX(Price)
3	2.90000	1.6	4.1

Aggregation and Grouping

- The GROUP BY statement groups together rows that have the same values in specified columns.
- It computes summaries (aggregates) for each unique combination of values.

Syntax

```
SELECT    Selections
FROM      Table1, Table2, ...
WHERE     Condition1
GROUP BY  Attr1, Attr2, ...
HAVING    Condition2
ORDER BY  Attr1, Attr2, ...
```

Selections : → Attributes specified in GROUP BY
→ Aggregate functions
→ No other non-aggregated attributes

Condition1 : Condition on the attributes in tables.

Condition2 : HAVING clause contains conditions on groups.

Attr1, Attr2 : GROUP BY clause specifies the grouping attributes

GROUP BY and HAVING statements in aggregations

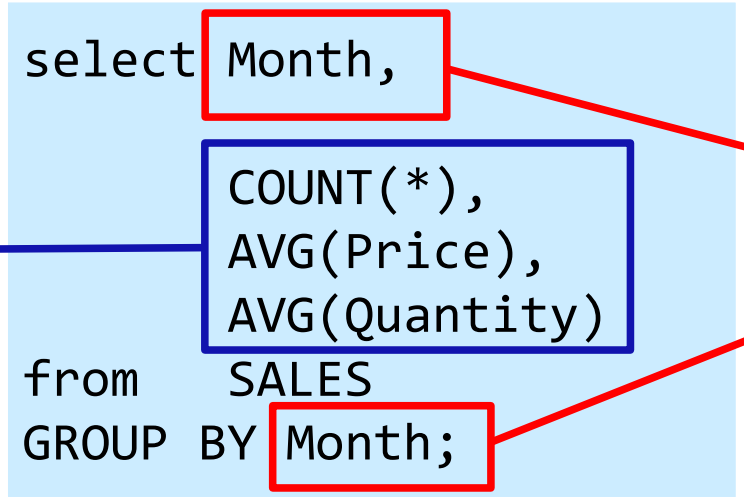
- **GROUP BY statement**
  - It groups rows that have the same values into summary rows.
  - It is often used with aggregate functions COUNT(), MAX(), MIN(), SUM(), AVG() to group the result-set by one or more columns.
  - Aggregation operations are applied on attributes (fields/columns) of a table.

- **HAVING statement**
  - It is used for specifying a selection condition on groups.
  - It can not be used without the GROUP BY clause.

- The SELECT clause includes only the grouping attribute(s) and the aggregation functions to be applied on each group of tuples.

GROUP BY clause

Query7 : For each month, retrieve the month name, the number of SALES in the month, the average price and the average quantity in each month.



Aggregated non-grouping columns Price and Quantity.

The non-aggregated column Month must appear in GROUP BY clause. Otherwise SQL gives error.

Result

Month	COUNT(*)	AVG(Price)	AVG(Quantity)
Jan	2	1.90000	28.0000
Feb	4	1.85000	45.7500
March	3	2.90000	61.3333

Execution Steps by DBMS

Step1) Group the attributes in the GROUP BY clause.
Step2) Select tuples for every group, and apply aggregation function of the SELECT clause. Obtain a single tuple per group.

```
SELECT COUNT(*), AVG(Price), AVG(Quantity), Month
FROM   SALES
GROUP  BY Month;
```

Step1) GROUP BY Months

Sale_ID	Product	Price	Quantity	Month
101	Orange	3.0	22	Jan
102	Kiwi	0.8	34	Jan
103	Orange	1.5	23	Feb
104	Apple	2.0	38	Feb
105	Cherry	2.7	44	Feb
106	Apricot	1.2	78	Feb
107	Orange	3.0	76	March
108	Cherry	4.1	60	March
109	Apple	1.6	48	March

Intermediate result

Step2) Compute and display aggregations

COUNT(*)	AVG(Price)	AVG(Quantity)	Month
2	1.90000	28.0000	Jan
4	1.85000	45.7500	Feb
3	2.90000	61.3333	March

Final result

GROUP BY and HAVING clauses

Query8 : Find how many records are there for each Product.

```
select product_name,
       COUNT(product_name)
from SALES
GROUP BY Product_name;
```

Result

product_name	COUNT(product_name)
Orange	3
Kiwi	1
Apple	2
Cherry	2
Apricot	1

GROUP BY and HAVING clauses

Query9 : Find how many records are there for the products, whose count is bigger than or equal to 2.

```
SELECT product_name, COUNT(product_name)
FROM SALES
GROUP BY Product_name
HAVING COUNT(product_name) >= 2;
```

Result

product_name	COUNT(product_name)
Orange	3
Apple	2
Cherry	2

WHERE and GROUP BY clauses

Query10 : Find how many records are there for the products, whose product name begins with 'A'.

```
SELECT product_name, COUNT(product_name)
FROM     SALES
WHERE    product_name LIKE 'A%'
GROUP BY Product_name;
```

Result

```
+--------------+----------------------+
| product_name | COUNT(product_name) |
+--------------+----------------------+
| Apple        |                    2 |
| Apricot      |                    1 |
+--------------+----------------------+
```

WHERE, GROUP BY, and HAVING clauses

Query11 : Find how many records are there for the products, whose product name begins with 'A', and whose record count is bigger than 1.

```
SELECT product_name, COUNT(product_name)
FROM     SALES
WHERE    product_name LIKE 'A%'
GROUP BY Product_name
HAVING   COUNT(product_name) > 1;
```

Result

```
+--------------+----------------------+
| product_name | COUNT(product_name) |
+--------------+----------------------+
| Apple        |                    2 |
+--------------+----------------------+
```

Execution Steps by DBMS

Step1) Execute the FROM and WHERE clauses.
Step2) Group the attributes in the GROUP BY clause.
Step3) Select one tuple for every group, and apply aggregation function of SELECT clause.
Step4) Apply the HAVING clause, eliminate groups based on HAVING.

Step1) FROM SALES WHERE product LIKE 'A%'

Sale_ID	Product	Price	Quantity	Month
104	Apple	200	99	Feb
106	Apricot	100	88	Feb
109	Apple	400	55	March

Intermediate result

Step2) GROUP BY Product

Sale_ID	Product	Price	Quantity	Month
104	Apple	200	99	Feb
109	Apple	400	55	March
106	Apricot	100	88	Feb

Intermediate result

Step3) Apply group level filters in HAVING clause
COUNT(product) > 1

Product	COUNT(product)
Apple	2
~~Apricot~~	~~1~~

Intermediate result

Step4) Process SELECT clause

Product	COUNT(product)
Apple	2

Final result

GROUP BY, and ORDER BY clauses

Query12 : Find sum of quantities for each group of products. Sort the result set by sum of quantities, in descending order.

```
SELECT product_name, SUM(quantity)
FROM     SALES
GROUP BY product_name
ORDER BY SUM(quantity) DESC;
```

Result

```
+--------------+---------------+
| product_name | SUM(quantity) |
+--------------+---------------+
| Orange       |           121 |
| Cherry       |           104 |
| Apple        |            86 |
| Apricot      |            78 |
| Kiwi         |            34 |
+--------------+---------------+
```

GROUP BY, and ORDER BY clauses

Query13 : Find sum of quantities for each group of products. Sort the result set by product names.

```
SELECT product_name, SUM(quantity)
FROM     SALES
GROUP BY product_name
ORDER BY product_name;
```

Result

```
+--------------+---------------+
| product_name | SUM(quantity) |
+--------------+---------------+
| Apple        |            86 |
| Apricot      |            78 |
| Cherry       |           104 |
| Kiwi         |            34 |
| Orange       |           121 |
+--------------+---------------+
```

GROUP BY, HAVING, and ORDER BY clauses

Query14 : Find sum of quantities for each group of products, whose sum is at least 100.
Sort the result set by sum of quantities.

```
SELECT product_name, SUM(quantity)
FROM       SALES
GROUP BY   product_name
HAVING     SUM(quantity) >= 100
ORDER BY   SUM(quantity) DESC;
```

Result

```
+--------------+---------------+
| product_name | SUM(quantity) |
+--------------+---------------+
| Orange       |           121 |
| Cherry       |           104 |
+--------------+---------------+
```

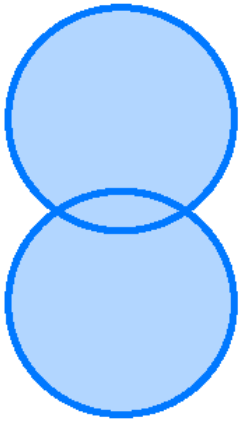
Topics

- Subqueries
- Join Operations
- Common Table Expression
- Set Operations

Set Operations

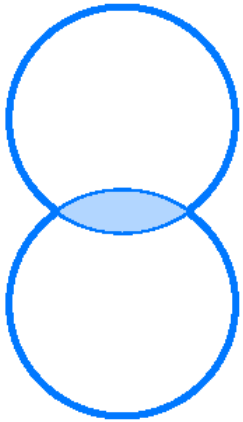
- The followings are SQL relational set operators.
- They are used to apply set operations on the result-sets of two **SELECT** statements.
- Two SELECT statements must return exactly the same number and types of columns from two tables. Otherwise SQL gives error.
- The column names would be the same as those in the first subquery.

UNION



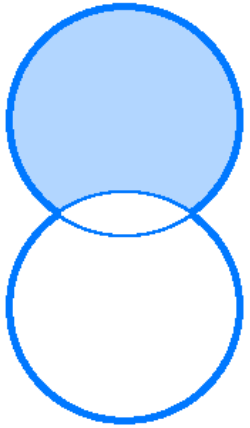
```
SELECT columns FROM T1
UNION
SELECT columns FROM T2
```

INTERSECT



```
SELECT columns FROM T1
INTERSECT
SELECT columns FROM T2
```

EXCEPT

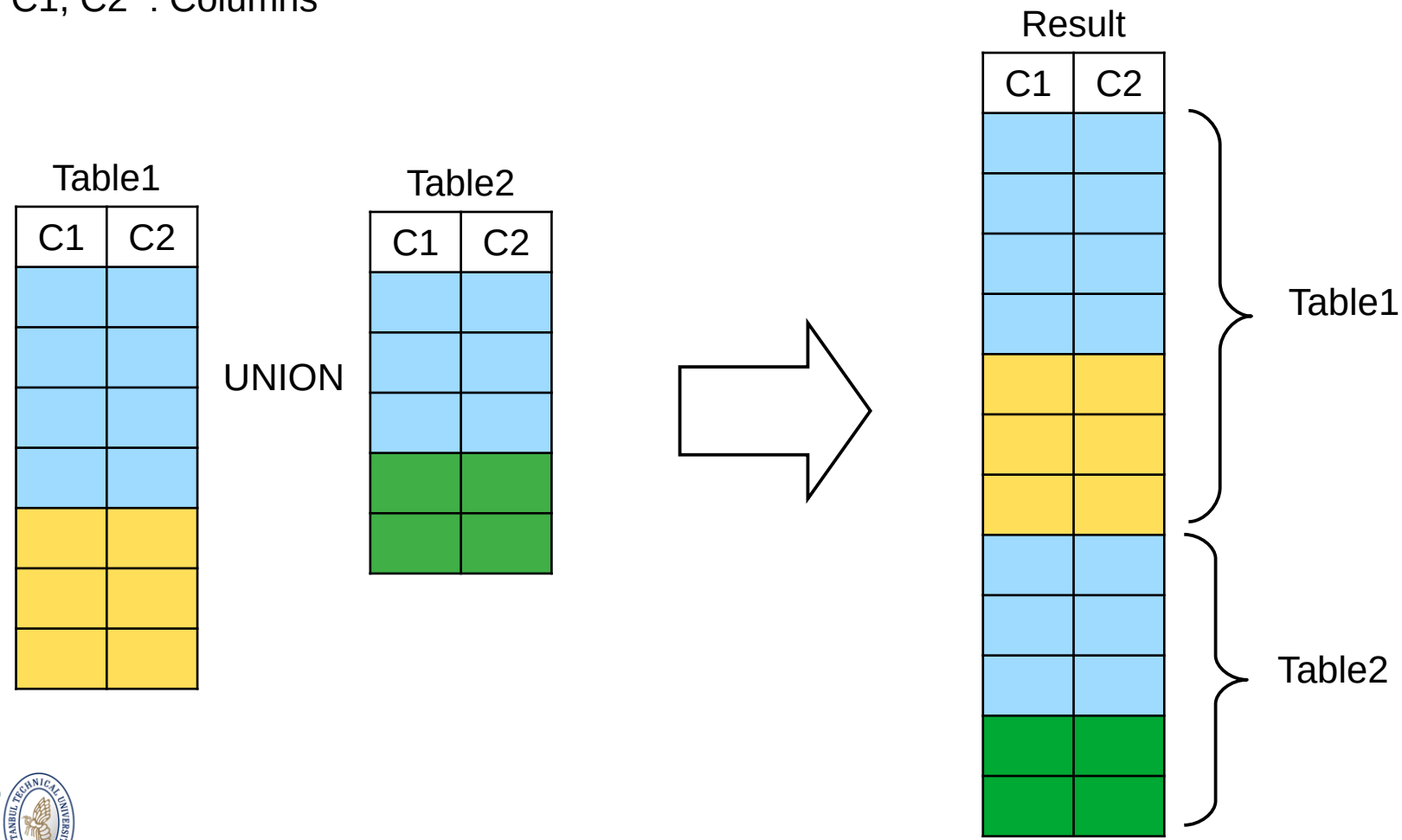


```
SELECT columns FROM T1
EXCEPT
SELECT columns FROM T2
```

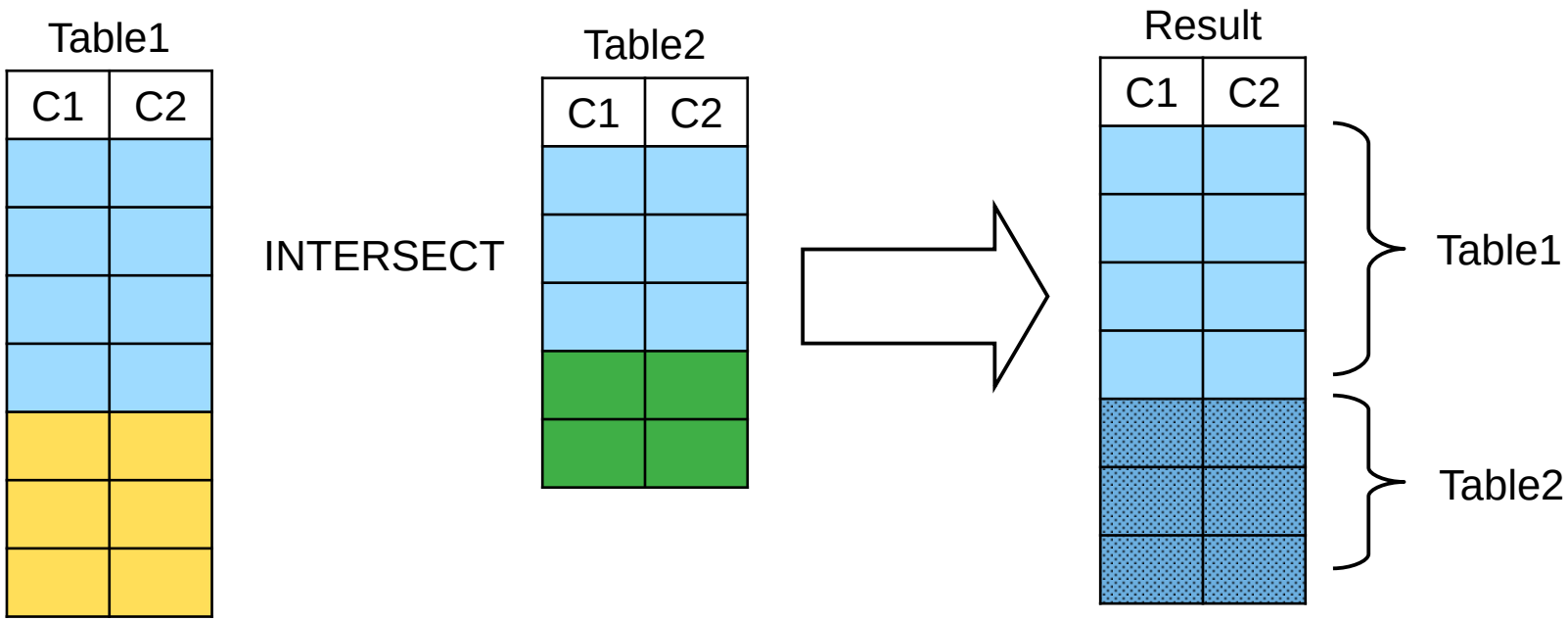
UNION command

Color explanation:

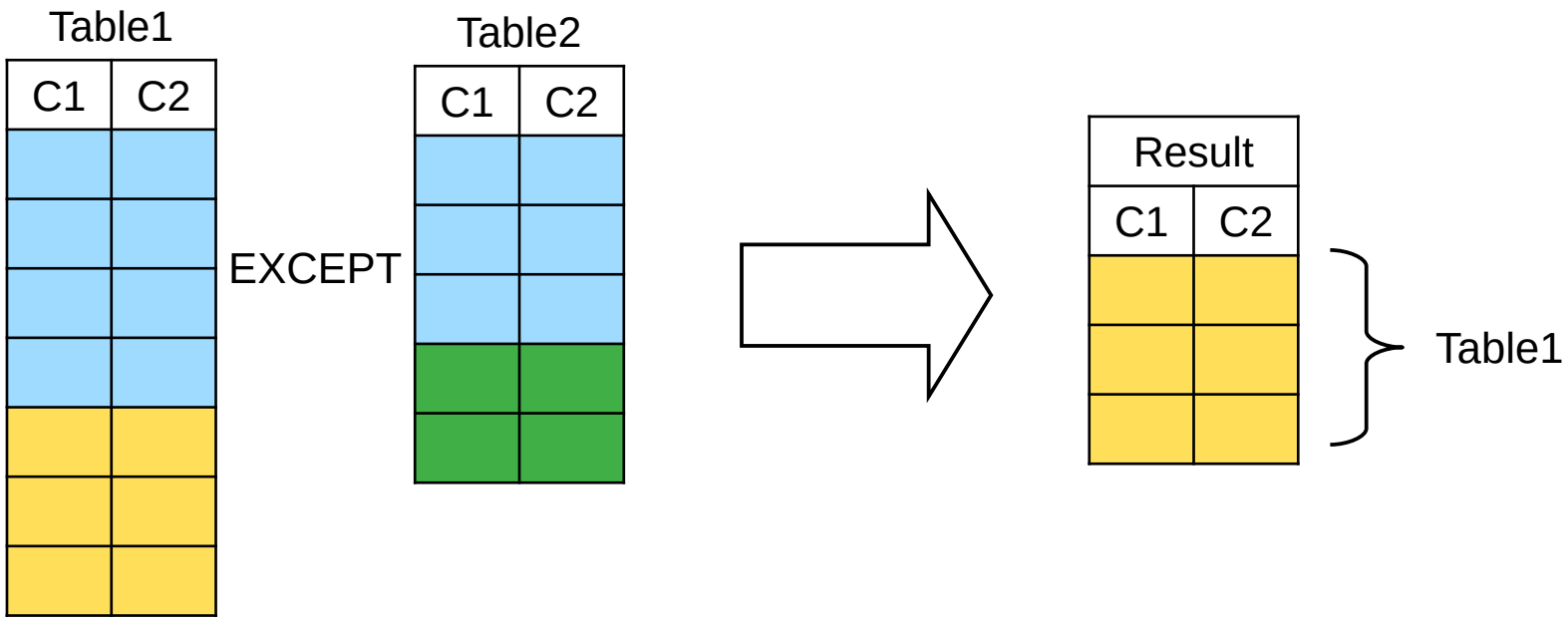
- The records that have the **same color** on both tables means their data contents are same.
- Set operations do not use any key fields.
- C1, C2 : Columns



INTERSECT command



EXCEPT command



Example: Customers and Employees Tables

- Suppose some persons can be both a Customer, and also an Employee at the same time.
- Schema definitions:

```
CUSTOMERS (CustomerID    int,
           CustomerName  varchar(30) );

EMPLOYEES (EmployeeID    int,
           EmployeeName  varchar(20) );
```

Sample Data in Tables

CUSTOMERS table

CustomerID	CustomerName
1001	Barbara Jones
1002	Christina Berglund
1003	Elizabeth Brown
1004	Elizabeth Lincoln
1005	Francisco Chang
1006	Frderique Citeaux
1007	Jennifer Davis
1008	Julia Trujillo
1009	Kathy Moos
1010	Laurence Clarks
1011	Maria Anders
1012	Patricia Johnson
1013	Patricio Simpson
1014	Pedro Afonso
1015	Victoria Ashworth
1016	Yang Wang

EMPLOYEES table

EmployeeID	EmployeeName
1	Antonio Moreno
2	Barbara Jones
3	Elizabeth Brown
4	Elizabeth Lincoln
5	Francisco Chang
6	Linda Brown
7	Linda Williams
8	Martin Sommer
9	Patricia Johnson
10	Thomas Hewlett
11	Victoria Ashworth

UNION Operator

- The UNION operator is used to combine the result-set of two or more SELECT statements.
- Every SELECT statement within UNION must have the same number of columns.
- The columns must also have similar data types.
- The columns in every SELECT statement must also be in the same order.
- UNION operator selects only distinct values by default.
  To allow duplicate values, **UNION ALL** is used.
- In MySql, the column names in the result-set are the same as those in the first SELECT statement.

Syntax

```
SELECT column_names FROM table1
[WHERE condition]
UNION
SELECT column_names FROM table2
[WHERE condition];
```

UNION Operator

Query1 : Build a complete list of all persons (either CUSTOMER or EMPLOYEE) containing the ID numbers and person names.

Use the UNION operation over CUSTOMERS and EMPLOYEES tables.

```
SELECT CustomerID as PersonID,
       CustomerName as PersonName,
      'Müşteri' AS PersonStatus
FROM CUSTOMERS

UNION

SELECT EmployeeID as PersonID,
       EmployeeName as PersonName,
       'Çalışan'
FROM EMPLOYEES
;
```

PersonID	PersonName	PersonStatus
1001	Barbara Jones	Müşteri
1002	Christina Berglund	Müşteri
1003	Elizabeth Brown	Müşteri
1004	Elizabeth Lincoln	Müşteri
1005	Francisco Chang	Müşteri
1006	Frderique Citeaux	Müşteri
1007	Jennifer Davis	Müşteri
1008	Julia Trujillo	Müşteri
1009	Kathy Moos	Müşteri
1010	Laurence Clarks	Müşteri
1011	Maria Anders	Müşteri
1012	Patricia Johnson	Müşteri
1013	Patricio Simpson	Müşteri
1014	Pedro Afonso	Müşteri
1015	Victoria Ashworth	Müşteri
1016	Yang Wang	Müşteri
1	Antonio Moreno	Çalışan
2	Barbara Jones	Çalışan
3	Elizabeth Brown	Çalışan
4	Elizabeth Lincoln	Çalışan
5	Francisco Chang	Çalışan
6	Linda Brown	Çalışan
7	Linda Williams	Çalışan
8	Martin Sommer	Çalışan
9	Patricia Johnson	Çalışan
10	Thomas Hewlett	Çalışan
11	Victoria Ashworth	Çalışan

INTERSECT and EXCEPT Operators

- INTERSECT operator is used to get the result-set records that are both in the first SELECT query result, and also in the second SELECT query result.
- Same rules with the UNION operator are required. Duplicate elimination behavior is the same as well. Use ALL keyword to retain duplicates.
- EXCEPT operator is used to get the result-set records that are only in the first SELECT query result, but not in the second SELECT query result.

Syntax

```
SELECT column_names FROM table1
[WHERE condition]
INTERSECT
SELECT column_names FROM table2
[WHERE condition]
;
```

```
SELECT column_names FROM table1
[WHERE condition]
EXCEPT
SELECT column_names FROM table2
[WHERE condition]
;
```

INTERSECT Operator

Query2 : Build a list of person names who are **both** CUSTOMERS and also EMPLOYEES at the same time.

```
SELECT CustomerName as PersonName
FROM CUSTOMERS

INTERSECT

SELECT EmployeeName
FROM EMPLOYEES
;
```

```
+--------------------+
| PersonName         |
+--------------------+
| Barbara Jones      |
| Elizabeth Brown    |
| Elizabeth Lincoln  |
| Francisco Chang    |
| Patricia Johnson   |
| Victoria Ashworth  |
+--------------------+
```

EXCEPT Operator

Query3 : Build a list of person names who are in the **only** CUSTOMERS table, but not in the EMPLOYEES table.

```
SELECT CustomerName
FROM CUSTOMERS

EXCEPT

SELECT EmployeeName
FROM EMPLOYEES
;
```

```
+--------------------+
| CustomerName       |
+--------------------+
| Christina Berglund |
| Frderique Citeaux  |
| Jennifer Davis     |
| Julia Trujillo     |
| Kathy Moos         |
| Laurence Clarks    |
| Maria Anders       |
| Patricio Simpson   |
| Pedro Afonso       |
| Yang Wang          |
+--------------------+
```

EXCEPT Operator

Query4 : Build a list of person names who are in the **only** EMPLOYEES table, but not in the CUSTOMERS table.

```
SELECT EmployeeName
FROM EMPLOYEES

EXCEPT

SELECT CustomerName
FROM CUSTOMERS
;
```

```
+----------------+
| EmployeeName   |
+----------------+
| Antonio Moreno |
| Linda Brown    |
| Linda Williams |
| Martin Sommer  |
| Thomas Hewlett |
+----------------+
```