Data Manipulation Language (DML)

lecture2.pdf
Lecture 3.pdf

Used by applications and users to retrieve, insert, modify, and delete records

`INSERT, DELETE, UPDATE, SELECT`
Simple set of commands for complicated selections

Insert into table

```
INSERT INTO person (
        first_name,
        last_name,
        gender,
        date_of_birth
)
VALUES ('Anne', 'Smith', 'FEMALE', DATE 'YYYY-MM-DD');
```

Select from table

Get everything from table

```
SELECT * FROM <table_name>;
```

Get column(s) from table

```
SELECT <column_name1>, <column_name2> FROM <table_name>;
```

Expression are available in SELECT with renaming

```
SELECT price * 0.965 AS priceInYEN FROM store;
```

Relation-based Aggregation

SUM(), AVG(), COUNT(), MIN() and MAX() can be used in a SELECT.

Note: `NULL` never contributes to a aggregation statement. - But if the only values in a column is `NULL`, then the result of the aggregation is `NULL`

Get avarage price for the store table

```
SELECT AVG(price) from store;
```

Group-based Aggregations

Use the GROUP BY Keyword to clause groups rows that have the same values into summary rows, like "find the number of customers in each country"
`GROUP BY` is often used with Relation-based Aggregation

The keyword `HAVING` may follow a `GROUP BY`. The condition of `HAVING` applies to each group, and groups not satisfying the condition are eliminated.

Group the amount of customers by country

```
SELECT COUNT(customer) FROM customers GROUP BY country;
```

Find the avarage price of those coffees that are served in at lease two coffehouses

```
SELECT coffee, AVG(price) FROM Sells GROUP BY coffee
HAVING COUNT(coffeehouse) > 1;
```

Sort table data

Use the ORDER BY Keyword

Get everything from table in ascending/descending order from a column
ascending by default - or use `ASC` keyword

```
SELECT * FROM <table_name> ORDER BY <column_name>;
```

descending order - use `DESC` keyword

```
SELECT * FROM <table_name> ORDER BY <column_name> DESC;
```

Remove duplicates (Distinct)

Get distinct values from column in a table

```
SELECT DISTINCT <column_name> FROM <table_name>;
```

Time complexity: $O(N \log N)$

Filter data by condition (WHERE, AND, OR)

```
SELECT * FROM <table_name> WHERE <condition> AND <condition> OR <condition>;
```

Get every "person" where gender = female

```
SELECT * FROM person WHERE gender = 'Female';
```

Get every "person" WHERE gender = female AND country = Denmark

```
SELECT * FROM person WHERE gender = 'Female' AND contry = 'Denmark';
```

Attribute renaming

Use the `AS` keyword to rename the Attribute name

```
SELECT * FROM person AS female WHERE gender = 'Female';
```

Patterns

Docs

A condition can compare a string to a pattern

```
<String> LIKE <pattern>
```

The `LIKE` expression returns true if the *`string`* matches the supplied *`pattern`*.

```
<String> NOT LIKE <pattern>
```

The `NOT LIKE` expression returns false if `LIKE` returns true, and vice versa. An equivalent expression is `NOT (_`string`_ LIKE _`pattern`_)`.

If *`pattern`* does not contain percent signs or underscores, then the pattern only represents the string itself; in that case `LIKE` acts like the equals operator. An underscore (`_`) in *`pattern`* stands for (matches) any single character; a percent sign (`%`) matches any sequence of zero or more character

Example

How many customers have no "Sales" in their contact title?

```
SELECT COUNT(*) FROM customers c
WHERE c.contact_title NOT LIKE '%Sales%'
```

How many customers have "Sales" as the first string in their contact title?

```
SELECT COUNT(*) FROM customers c
WHERE c.contact_title NOT LIKE 'Sales%'
```

```
<String> SIMILAR TO <pattern>
```

The `SIMILAR TO` operator returns true or false depending on whether its pattern matches the given string. It is similar to `LIKE`, except that it interprets the pattern using the SQL standard's definition of a regular expression. SQL regular expressions are a curious cross between `LIKE` notation and common (POSIX) regular expression notation.

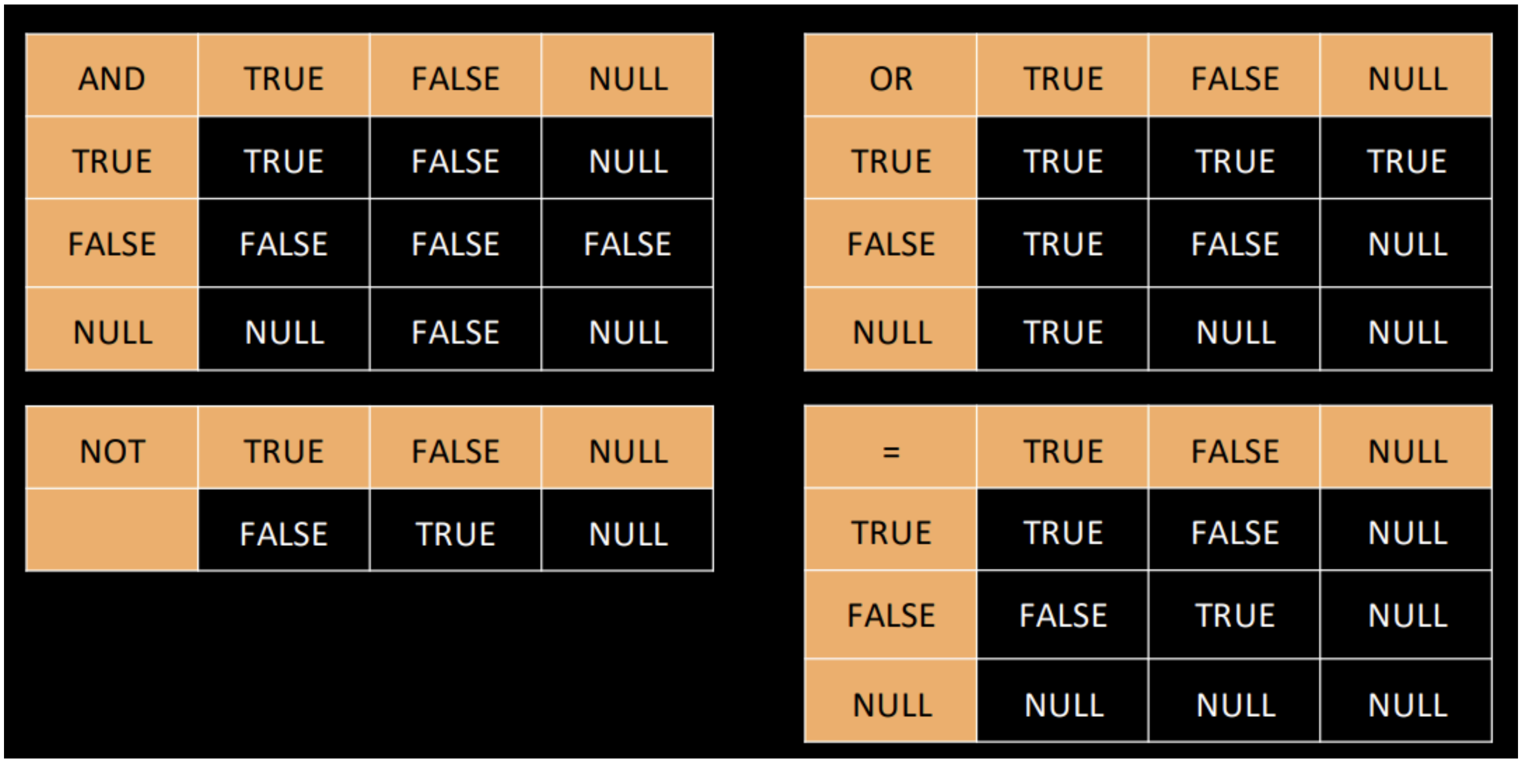
Like `LIKE`, the `SIMILAR TO` operator succeeds only if its pattern matches the entire string; this is unlike common regular expression behavior where the pattern can match any part of the string. Also like `LIKE`, `SIMILAR TO` uses `_` and `%` as wildcard characters denoting any single character and any string, respectively (these are comparable to `.` and `.*` in POSIX regular expressions).

In addition to these facilities borrowed from `LIKE`, `SIMILAR TO` supports these pattern-matching metacharacters borrowed from POSIX regular expressions:

- `|` denotes alternation (either of two alternatives).
- `*` denotes repetition of the previous item zero or more times.
- `+` denotes repetition of the previous item one or more times.
- `?` denotes repetition of the previous item zero or one time.
- `{ m }` denotes repetition of the previous item exactly `m` times.
- `{ m ,}` denotes repetition of the previous item `m` or more times.
- `{ m , n }` denotes repetition of the previous item at least `m` and not more than `n` times.
- Parentheses `()` can be used to group items into a single logical item.

- A bracket expression `[...]` specifies a character class, just as in POSIX regular expressions.

Three-Valued Truth Tables



AND	TRUE	FALSE	NULL
TRUE	TRUE	FALSE	NULL
FALSE	FALSE	FALSE	FALSE
NULL	NULL	FALSE	NULL

OR	TRUE	FALSE	NULL
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	NULL
NULL	TRUE	NULL	NULL

NOT	TRUE	FALSE	NULL
	FALSE	TRUE	NULL

=	TRUE	FALSE	NULL
TRUE	TRUE	FALSE	NULL
FALSE	FALSE	TRUE	NULL
NULL	NULL	NULL	NULL

Division

R_1/R_2 = tuples of R_1 associated with all tuples of R_2.

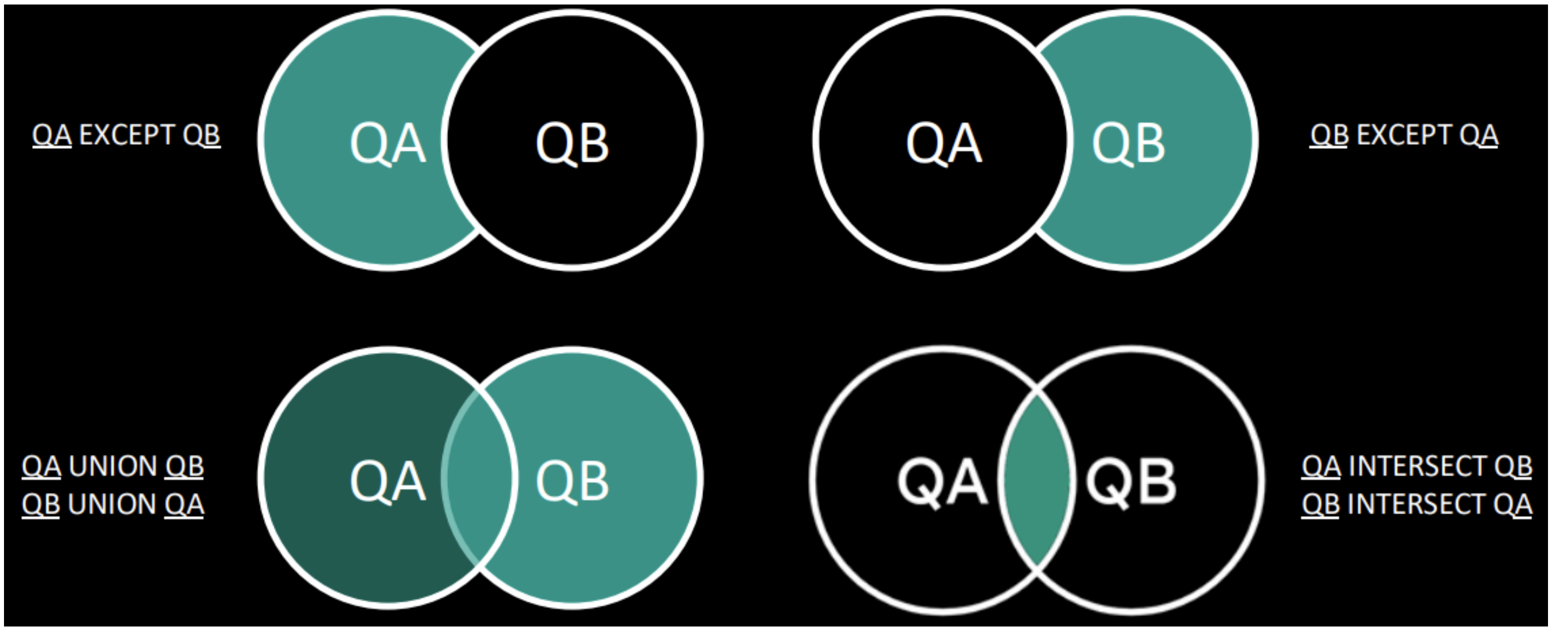
We can write division using GROUP BY, HAVING, and a COUNT subquery.

Examples:

- Find the students who have taken **all** courses in a program
- Find the airlines who land at **all** airports in a country/continent/the world
- Find the coffeehouses that sell **all** existing coffees

Set Operator

Set operations is a type of operations which allow the results of multiple queries to be combined into a single result set.



Set operator removes duplicates, but can be kept with the keyword `ALL` after the set operator, like:

```
R UNION ALL S -- Keep duplicates after UNION of the sets R & S
R EXCEPT ALL S -- Keep duplicates after EXCEPT of the sets R & S
R INTERSECT ALL S -- Keep duplicates after INTERSECT of the sets R & S
```

Union

The `UNION` operator is used to combine the result-set of two or more `SELECT` statements.

```
SELECT _column_name(s)_ FROM _table1_
UNION
SELECT _column_name(s)_ FROM _table2_;
```

Intersect

the `INTERSECT` statement will return only those rows which will be common to both of the `SELECT` statements.

```
SELECT _column_name(s)_ FROM _table1_
INTERSECT
SELECT _column_name(s)_ FROM _table2_
```

Except

Retrieve all the unique records from the left operand (query), except the records that are present in the result set of the right operand (query)

```
SELECT _column_name(s)_ FROM _table1_
EXCEPT
SELECT _column_name(s)_ FROM _table2_
```

Join

Docs

Queries can access multiple tables at once, or access the same table in such a way that multiple rows of the table are being processed at the same time. Queries that access multiple tables (or multiple instances of the same table) at one time are called *join* queries. **They combine rows from one table with rows from a second table, with an expression specifying which rows are to be paired.**

```
SELECT * FROM weather JOIN cities ON city = name;
```

To return all the weather records together with the location of the associated city, the database needs to compare the `city` column of each row of the `weather` table with the `name` column of all rows in the `cities` table, and select the pairs of rows where these values match. This would be accomplished by this query.

Natural and Cross Joins (Product)

- **Natural join** on the base of the common columns in the tables.
  - Can be dangerous, not recommended!
- **CROSS JOIN*** returns all the records from the left and right tables. CROSS JOIN returns a combination of each row in the left table paired with each row in the right table
  - Can simulate with an always true condition (ON 1 = 1)
  - Rarely (but not never) the right thing to do!

```
SELECT * FROM table1 NATURAL JOIN table2;
```

```
SELECT * FROM table1 CROSS JOIN table2;
```

Self-Join

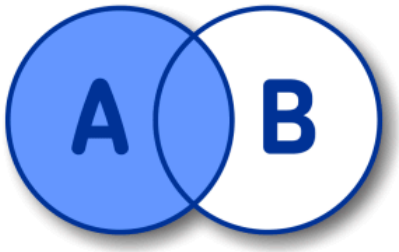
Join the same table with itself.

```
SELECT _column(s) FROM table1 t1, table1 t2 WHERE _condition;
```

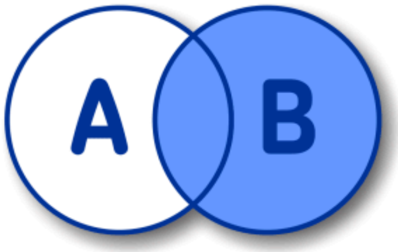
Self-Join with regular Join

```
SELECT _column(s) FROM table1 t1 JOIN table1 t2 ON _condition WHERE _condition;
```

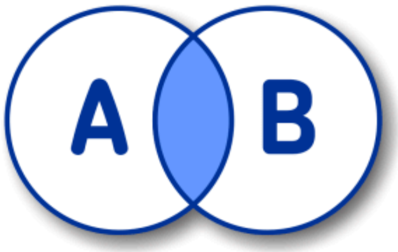
SQL JOINS



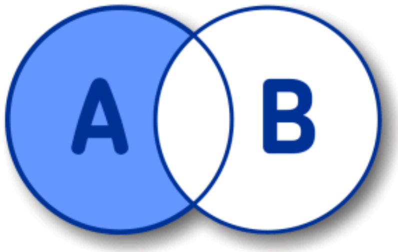
SELECT * FROM
A LEFT JOIN B
ON A.KEY = B.KEY



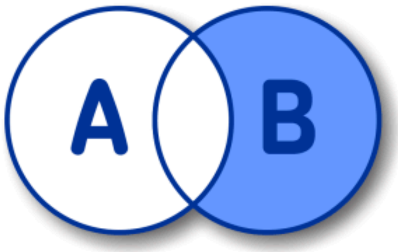
SELECT * FROM
A RIGHT JOIN B
ON A.KEY = B.KEY



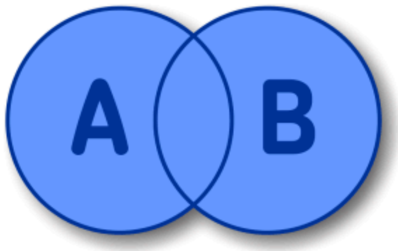
SELECT * FROM
A INNER JOIN B
ON A.KEY = B.KEY



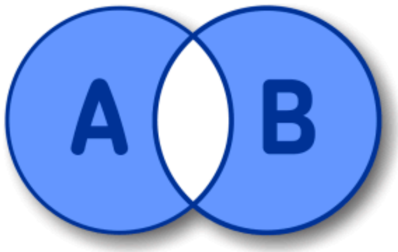
SELECT * FROM A
LEFT JOIN B
ON A.KEY = B.KEY
WHERE B.KEY IS NULL



SELECT * FROM A
RIGHT JOIN B
ON A.KEY = B.KEY
WHERE A.KEY IS NULL



SELECT * FROM A
FULL OUTER JOIN B
ON A.KEY = B.KEY



SELECT * FROM A FULL
OUTER JOIN B ON A.KEY =
B.KEY WHERE A.KEY IS
NULL OR B.KEY IS NULL