CMPT 321 Fall 2017

# Structured Query Language SQL

Lecture 03.01

By Marina Barsky

#### **SELECT FROM WHERE**

## Structured Query Language (SQL)

- SQL is a high-level special-purpose language for manipulating relations
- SQL is mostly a declarative language:

you declare what you want without specifying how you want to get answer

• SQL provides a limited set of operations:

mostly implementations of Relational Algebra operators

 SQL programmer needs to focus on readability and on getting the right results – do not need to worry about efficiency:

because the DMBS optimizes every query and chooses the most efficient implementation for each operation

#### Sub-sets of SQL

- Data Definition Language (DDL): CREATE, ALTER, DROP, RENAME
- Data Manipulation Language (DML): INSERT, UPDATE, DELETE, SELECT, Transaction control: COMMIT, ROLLBACK
- Data Control Language (DCL): GRANT, REVOKE

#### Language elements

- Clauses
- Expressions produce either scalar values, or tables
- Predicates specify conditions that can be evaluated according to SQL three-valued logic (3VL) to true/false/unknown
- Queries
- Statements

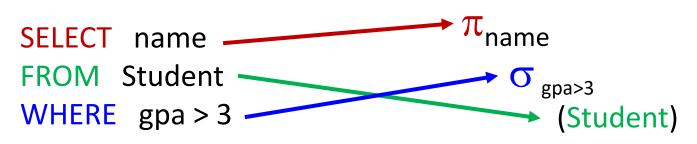
#### SELECT clause corresponds to projection $\pi$ in RA Query: list student names with GPA >3

Student			
Name GPA Country			
Bob	3	Canada	
John	3	Britain	
Tom	3.5	Canada	
Maria	4	Mexico	



S		
Name	GPA	Country
Tom	3.5	Canada
Maria	4	Mexico

S =



#### How the query is evaluated

- Each tuple of *Student* is inspected
- Each attribute of WHERE clause is substituted with the actual tuple value
- The condition is then evaluated, and if true – this tuple is added to the output relation

Student			
Name	GPA	Country	
Bob	3	Canada	
John	3	Britain	
Tom	3.5	Canada	
Maria	4	Mexico	

```
WHERE 3 > 3
FALSE
```

WHERE 4 > 3 TRUE

#### How to parse SQL query

SELECT a,b FROM X,Y,Z WHERE X.c=Y.c AND Z.d > 12

- 1. What relations are involved: FROM clause
- 2. Selection condition on rows: WHERE clause
- 3. Projection on columns: SELECT clause

## FROM clause

#### FROM clause

FROM is always followed by name(s) of input relation(s):

SELECT \* FROM Student

#### FROM clause: sub-queries

- You can construct a new relation using a sub-query, give it a name (optional in most DBMSs), and use it in FROM clause
- Thus, the result of one query (*sub-query*) becomes an input to another.

Student		
Name GPA Country		Country
Bob	3	Canada
John	3	Britain
Tom	3.5	Canada
Maria	4	Mexico

SELECT name FROM (SELECT \* FROM Student WHERE gpa > 3) AS goodStudent

### FROM clause: table alias example I

- We can rename input relations and their attributes to use in SELECT and WHERE clauses
- In that way we can perform queries on self-relationships

	Faculty		
ID	Name	SupID	
1	Dr. Monk	2	
2	Dr. Pooh	3	
3	Dr. Patel		

SELECT e.name [AS] employee, s.name [AS] supervisor FROM Faculty AS e, Faculty AS s WHERE e.SupID = s.ID

### FROM clause: table alias example II

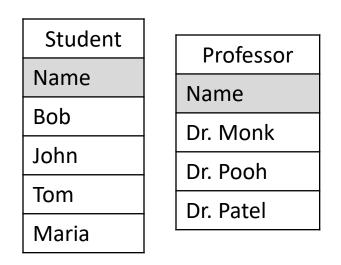
- We can rename input relations and their attributes to use in SELECT and WHERE clauses
- Or perform join of table with itself

Student	
Name Address	
Bob	Canada 1
John Britain 2	
Tom	Canada 1
Maria	Britain 2

SELECT S1.name, S2.name FROM Student S1, Student S2 WHERE S1.address = S2.address AND S1.name < S2.name; Producing a new table from multiple tables

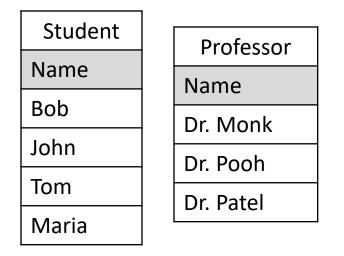
#### FROM clause: list of tables

• List of tables without any condition in the WHERE clause produces ...



SELECT \* FROM Student, Professor

#### Unexpected result?



#### SELECT \* FROM Student, Professor T=Student x Professor

Т	
S.Name	P.Name
Bob	Dr. Monk
Bob	Dr. Pooh
Bob	Dr. Patel
John	Dr. Monk
John	Dr. Pooh
John	Dr. Patel
Tom	Dr. Monk
Tom	Dr. Pooh
Tom	Dr. Patel
Maria	Dr. Monk
Maria	Dr. Pooh
Maria	Dr. Patel

# FROM clause: list of tables - warning

 List of tables without any condition in the WHERE clause produces Cartesian product

The implicit writing of Cartesian product - a dangerous illusion that you are asking the list of Professors to be appended to the end of the list of students, while in fact you are asking to pair each tuple in Student with each tuple in Professor

## Combination of 2 tables: Cartesian product in SQL

- Results from multi-table query that does not have a WHERE clause
- The product results in a huge output which normally is not very useful
- To avoid a Cartesian product, we use one or more valid join conditions

#### Joins: NATURAL JOIN

Student			
Name	Country	GPA	Ν
Bob	Canada	3	B
John	Britain	3	Jc
Tom	Canada	3.5	Тс
Maria	Mexico	4	B

#### SELECT \* FROM Student NATURAL JOIN RegisteredFor;

More explicit:

SELECT \* FROM Student JOIN RegisteredFor USING (name);

RegisteredFor	
Name	Торіс
Bob	Algorithms
John	Algorithms
Tom	Algorithms
Bob	Python
Tom	Python
Bob	Databases
John	Databases
Maria	Databases
John	GUI
Maria	GUI

#### Joins: NATURAL JOIN - USING

Teacher	
Name	Score
Bob	2
John	3
Tom	4

Student		
Name	Country	Score
Bob	Canada	3
John	Britain	3
Tom	Canada	3.5
Maria	Mexico	4

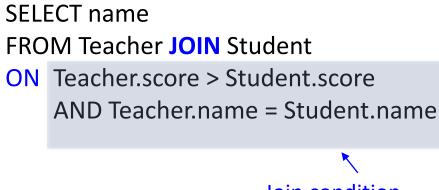
If you want to join only on a single common attribute – specify it with USING:

SELECT name, Teacher.score, Student.score FROM Teacher JOIN Student USING (name);

#### Joins: theta join

Teacher	
Name	Score
Bob	2
John	3
Tom	4

Student		
Name Country Score		Score
Bob	Canada	3
John	Britain 3	
Tom	Canada 3.5	
Maria Mexico 4		4



Join condition

# Multiple joins are required to collect information from multiple tables

Student		
Name	Country GPA	
Bob	Canada 3	
John	Britain 3	
Tom	Canada 3.5	
Maria	Mexico 4	

Teaches		
Name Topic		
Dr. Monk	Algorithms	
Dr. Pooh	Python	
Dr. Patel	Databases	
Dr. Patel	GUI	

RegisteredFor		
Name Topic		
Bob	Algorithms	
John	Algorithms	
Tom Algorithms		
Bob	Python	
Tom	Python	

It is preferably to write joining attributes explicitly, using WHERE clause - to avoid mistakes:

SELECT **s**.name AS student, **r**.topic AS course, **t**.name AS professor FROM Student **s**, RegisteredFor **r**, Teaches **t** WHERE **s**.name = **r**.name AND **r**.topic = **t**.topic

#### NULL values in joined columns

We use NULL to indicate:

- Value unknown
- Value inapplicable
- Value withheld

#### NULL is a special value

- When joining on condition involving attributes A and B:
- If both A and B are NULL:
  - A=B returns false
  - A<>B returns false
- If one of A or B is NULL
  - A=B returns false
  - A<>B returns false
- The reason is that DBMS uses a **3-valued logic**
- The NULLs do not generally appear in the results of joins

#### OUTER JOIN

- Preserves dangling tuples (that did not match any tuple in another table) by padding them with NULL
- Has 3 types:
- Full: Pad dangling tuples in both tables.
  - L FULL OUTER JOIN R
- Left outerjoin: Only pad dangling tuples of L.
  - L LEFT OUTER JOIN R

- **Right outerjoin**: Only pad dangling tuples of R.
  - L RIGHT OUTER JOIN R

## Keywords INNER and OUTER

- There are keywords INNER and OUTER, but you never need to use them.
- Your intentions are clear anyway:
  - You get an OUTER join iff you use the keywords LEFT, RIGHT, or FULL.
  - If you don't use these keywords you get an inner join normal join.



#### OUTER JOIN example: LEFT JOIN

Teacher			Student			Result
Name	Score	Name	Country	Score	Name	Country
Bob	2	Bob	Canada	3	Bob	Canada
John	3	John	Britain	3	John	Britain
Tom	4	Tom	Canada	3.5	Tom	Canada
Kim	3	Maria	Mexico	4	Kim	NULL

SELECT t.name, country

FROM Teacher t LEFT JOIN Student s

ON t.name = s.name

#### OUTER JOIN example: FULL JOIN

Теас	her	Student		
Name	Score	Name	Country	Score
Bob	2	Bob	Canada	3
John	3	John	Britain	3
Tom	4	Tom	Canada	3.5
Kim	3	Maria	Mexico	4

Result			
Name	Country	t.score	s.score
Bob	Canada	2	3
John	Britain	3	3
Tom	Canada	4	3.5
Kim	NULL	3	NULL
Maria	Mexico	NULL	4

#### SELECT \*

FROM Teacher t FULL JOIN Student s

ON t.name = s.name

#### Subquery or Join?

- We can achieve the same result by using both subqueries and joins
- Which one is better?
- The one which is more readable both queries will be parsed and optimized into the same code by DBMS

#### Example 2

What does this do?
 SELECT studentID, courseID, grade
 FROM Took,
 (SELECT \*
 FROM Offering
 WHERE instructor='David') Doffering
 WHERE Took.courseID = Doffering. courseID;

• Can you suggest another version?

## WHERE clause

#### WHERE clause

The predicates (conditions) can be written using:

- Column names
- Logical and comparison operators
- Mathematical expressions
- Constants
- Built-in DBMS functions
- Sub-queries

#### **Building Boolean expressions**

- We can build Boolean expressions with operators that produce Boolean results.
  - comparison operators: =, <>, <, >, <=, >=
  - and many other operators:
     see section 6.1.2 6.1.3 in the textbook
- Compound conditions are constructed using logical operators: AND, OR, NOT.



#### Checking for NULLs

- Can't meaningfully use = or <>
- Should use: IS NULL
  - IS NOT NULL

SELECT \* FROM Students WHERE age IS NOT NULL;

#### **Operations involving NULL**

- A tuple is in a query result iff the WHERE clause evaluates to TRUE.
- When we compare using any comparison operators: (for example a < b), and a or b or both are NULL, the result is UNKNOWN – the third truth value, SQL special
- But a query only produces a tuple in the answer if its truth value for the WHERE clause is TRUE (not FALSE or UNKNOWN).

#### 3-valued truth of databases

Rule to remember:

```
TRUE = 1, FALSE = 0, UNKNOWN (NULL) = \frac{1}{2}
```

AND: min, OR: max, NOT: 1-x

x	У	x AND y (min)	x OR y (max)
FALSE ( <b>0</b> )	FALSE ( <mark>0</mark> )		
FALSE ( <b>0</b> )	NULL( <mark>½</mark> )		
FALSE ( <b>0</b> )	TRUE (1)		
NULL( <mark>½</mark> )	NULL( <mark>½</mark> )		
NULL( <mark>½</mark> )	TRUE (1)		
TRUE ( <mark>1</mark> )	TRUE (1)		

x	NOT x (1-x)
FALSE ( <mark>0</mark> )	
NULL( <mark>½</mark> )	
TRUE ( <mark>1</mark> )	

#### 3-valued truth of databases

Rule to remember:

```
TRUE = 1, FALSE = 0, UNKNOWN (NULL) = \frac{1}{2}
```

AND: min, OR: max, NOT: 1-x

x	У	x AND y (min)	x OR y (max)
FALSE (0)	FALSE (0)	FALSE (0)	FALSE (0)
FALSE (0)	NULL(½ )	FALSE (0)	NULL(½)
FALSE (0)	TRUE (1)	FALSE (0)	TRUE (1)
NULL(½ )	NULL(½)	NULL(½ )	NULL(½)
NULL(½ )	TRUE (1)	NULL(½ )	TRUE (1)
TRUE (1)	TRUE (1)	TRUE (1)	TRUE (1)

x	NOT x (1-x)
FALSE (0)	TRUE (1)
NULL(½ )	NULL(½)
TRUE (1)	FALSE (0)

#### Example

SELECT \* FROM course WHERE year <=3 OR year >3

Course		
Торіс	Year	
Databases	3	
HTML		
GUI	2	

Meaning:

SELECT \* FROM course WHERE year is NOT NULL

### Comparison of strings

Strings can be compared (lexicographically) using the same operators:

	Student		
=	Name	Country	GPA
<>	Bob	Canada	3
<	John	Britain	3
>	Joan	Canada	3.5
<=	Maria	Mexico	4

>=

#### BETWEEN A and B – is equivalent to >=A and <=B

SELECT \* FROM student WHERE name <= 'John' SELECT \* FROM student WHERE name > 'Job'

#### Patterns

• General form:

<Attribute> LIKE <pattern>

or

<Attribute> NOT LIKE <pattern>

Student		
Name	Birthdate	Comment
Bob	'1990-12-04'	Mike's brother
John	'1987-11-30'	
Joan	'1993-12-09'	John's sister
Maria	'1989-02-28'	

<pattern> is a quoted string which may contain

% = meaning "any string"

\_ = meaning "any character."

SELECT \* FROM student WHERE name LIKE 'Jo%';

#### Patterns: apostrophe

• Two consecutive apostrophes represent one apostrophe and not the end of the string.

Student		
Name	Birthdate	Comment
Bob	'1990-12-04'	Mike's brother
John	'1987-11-30'	
Joan	'1993-12-09'	John's sister
Maria	'1989-02-28'	

SELECT name FROM student WHERE comment LIKE '%"s%';

#### Patterns: % and

- What if the pattern contains the characters % or \_? We should "escape" their special meaning preceding them by some escape character. **SQL** allows us to use a custom escape character.
- Syntax: s LIKE 'x%%x%%' ESCAPE 'x'

**x** will be the escape character.

Example of matched string: '%aaaa%bb'

	Student		
Name	Birthdate	Comment	SELECT name
Bob	'1990-12-04'	Mike's brother	FROM student
John	'1987-11-30'		WHERE comment LIKE 'my_
Joan	'1993-12-09'	John's sister	ESCAPE 'y';
Maria	'1989-02-28'	m_1	

#### Comparison of dates

- SQL syntax is often DBMS-specific, and does not necessarily completely follows standards
- In particular date and time syntax, string concatenation, NULLs, and comparison case sensitivity vary from vendor to vendor
- In SQLite dates are stored as strings
- Standard date-time data type format is

**'YYYY-MM-DD HH:MM:SS'**: for example '1990-04-12 12:42:30'

#### Comparison of dates in SQLite

- Default date-time data type format is 'YYYY-MM-DD': for example '1990-04-12'
- Dates can be compared against string literal using regular comparison operators

Student		
Name	Birthdate	
Bob	'1990-12-04'	
John	'1987-11-30'	
Joan	'1993-12-09'	
Maria	'1989-02-28'	

SELECT name

**FROM** student

WHERE birthdate < '1989-03-28'

# Pattern example with dates in standard format

- Born in 1980s:
- SELECT name
- **FROM** student
- WHERE Birthdate > '1979-12-31'
- AND Birthdate < '1990-01-01'
- We can use LIKE:
- SELECT name
- **FROM** student
- WHERE Birthdate LIKE '\_\_\_8%'

Student		
Name	Birthdate	
Bob	'1990-12-04'	
John	'1987-11-30'	
Joan	'1993-12-09'	
Maria	'1989-02-28'	

### Conditions involving lists

SELECT name FROM student WHERE country = 'Canada' OR country = 'Britain' OR country='Australia'

Student		
Name	Country	GPA
Bob	Canada	3
John	Britain	3
Tom	Canada	3.5
Maria	Mexico	4

• Instead:

**SELECT** name

FROM

Student

WHERE country IN ('Canada', 'Australia', 'Britain')

#### WHERE clause

The conditions can be written using

- Column names
- Logical and comparison operators
- Mathematical expressions
- Constants
- Built-in DBMS functions
- Sub-queries

#### Sub-queries in WHERE clause

- We can compare the value in the column in the current tuple to a value in another column (of the same tuple)
- We can also compare it to the result of a subquery
- Syntax:
  - The subquery must be parenthesized.
  - May name the result, so you can refer to it in the outer query.

# Subquery as a value in a WHERE clause

- If a subquery is guaranteed to produce exactly one tuple, then the subquery can be used as a value.
- Simplest situation: that one tuple has only one component.



#### Example

• Find all students with a gpa greater than that of John.

SELECT name FROM Student WHERE gpa > (SELECT gpa FROM Student WHERE name = 'John');

This is analogous to something we can't do in RA:

πname σ gpa > («subquery») Student

Student		
Name	Country	GPA
Bob	Canada	3
John	Britain	3
Tom	Canada	3.5
Maria	Mexico	4

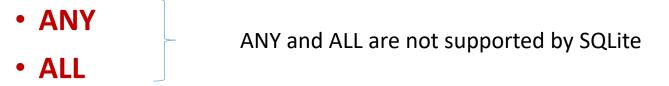
## What if the subquery returns more than one value?

We can make comparisons using a special quantifier.
 SELECT name
 FROM Student
 WHERE gpa >

 (SELECT gpa
 FROM Student
 WHERE name = 'John');

We can require that
 gpa >= all of them, or
 gpa > at least one of them.

SQL operators on subquery that returns multiple tuples - to produce a Boolean result



- IN
- EXISTS
- These operators can be negated by putting NOT in front of the entire expression.

#### ANY

- Suppose subquery returns relation R. If R is a unary relation (on a single column) then
- Condition s > ANY R is true if s is greater than at least one value in unary relation R.
  - Similarly we can use any other comparison operators in place of >. For instance, s = ANY R is the same as s IN R.

• If R is not unary we could match the entire tuple, but this feature is not supported by most DBMSs.

#### ALL

- Suppose subquery returns relation R.
- s > ALL R is true if s is greater than every value in the unary (one column) relation R.
  - Similarly, the > operator could be replaced by any other comparison operator with the analogous meaning. For instance, s <> ALL R is the same as s NOT IN R.

#### Example with ANY

SELECT name FROM student WHERE GPA > **ANY** (SELECT GPA FROM Student)

What is the result?

Student		
Name	Country	GPA
Bob	Canada	3
John	Britain	3
Tom	Canada	3.5
Maria	Mexico	4

#### Example with ANY

SELECT name
FROM student
WHERE GPA > ANY
(SELECT GPA
FROM
Student)

Student		
Name	Country	GPA
Bob	Canada	3
John	Britain	3
Tom	Canada	3.5
Maria	Mexico	4

- "Any" sounds a lot like "every" in this query. But it means "any one or more".
- Remember that ANY is existentially quantified.
- This query sounds much better when we express it instead with the keyword SOME, which is a synonym for ANY in SQL.

#### Example with ALL

SELECT name FROM student WHERE GPA > ALL (SELECT GPA FROM Student)

What is the result?

Student		
Name	Country	GPA
Bob	Canada	3
John	Britain	3
Tom	Canada	3.5
Maria	Mexico	4

#### Example with ALL

SELECT name FROM student WHERE GPA > = ALL (SELECT GPA FROM Student)

What is the result?

Student		
Name	Country	GPA
Bob	Canada	3
John	Britain	3
Tom	Canada	3.5
Maria	Mexico	4

#### IN

- For subquery R:
- s IN R is true if s is equal to one of the tuples in R. Likewise,
   s NOT IN R is true if and only if s is equal to no tuple in R.

• s can be a list of attributes and the entire tuple is compared

#### Example with IN

SELECT name FROM Student WHERE country IN (SELECT countryName FROM EnglishSpeakingCountries)

Student		
Name	Country	GPA
Bob	Canada	3
John	Britain	3
Tom	Canada	3.5
Maria	Mexico	4



Suppose we have tables R(a, b) and S(b, c).

1.What does this query do? SELECT a FROM R WHERE b IN (SELECT b FROM S);

2.Can we express this query without using IN?

#### EXISTS

- For subquery R:
- EXISTS *R* is a condition that is true if **R** is not empty.

• Read it as "exists at least one row in the subquery result"

#### Correlated subqueries

- EXISTS (NOT EXISTS) are used with *correlated subqueries*
- The EXISTS operator checks if the inner query returns at least one row, and it returns TRUE or FALSE
- If a subquery refers only to names defined inside it, it can be evaluated once and used repeatedly in the outer query.
- If it refers to any name defined outside of itself, it must be evaluated once for each tuple in the outer query. These are called *correlated subqueries*.

#### Example 1: EXISTS

SELECT Teacher.Name FROM Teacher outer WHERE EXISTS (SELECT '1' FROM Student WHERE name = outer. name);

Teacher		
Name Score		
Tom 4		
Kim 3		

Student		
Name	Country	Score
Bob	Canada	3
John	Britain	3
Tom	Canada	3.5
Maria	Mexico	4

#### Example 2: EXISTS

SELECT name, gpa FROM Student WHERE **EXISTS** ( SELECT \* FROM Took WHERE Student.name = Took.name AND grade > 85 );

Student		
Name	GPA	
Bob	3	
John	3	
Tom	3.5	
Maria	4	

Took		
Name	Course	Grade
Bob	Algo	55
John	Algo	90
Tom	DB	85
Maria	HCI	100

#### Example 3: EXISTS

SELECT DISTINCT Course FROM Took WHERE EXISTS ( SELECT \* FROM Took t, Offering o WHERE t.course = o.course AND t.course <> Took.course AND

o.dept = 'CSC' AND

took.name = t.name );

Offering		
Course Dept		
Algo	CSC	
DB	CSC	
Java	CSC	
НСІ	CSC	

Took		
Name	Course	Grade
Bob	Algo	55
John	Algo	90
Tom	DB	85
Maria	HCI	100

## SELECT clause

#### Expressions in SELECT clauses

- Instead of a simple projection, you can use an expression in a SELECT clause.
- Operands: attributes, constants Operators: arithmetic ops, string ops
- Examples: SELECT name, grade+10 AS adjusted FROM Took;

```
SELECT dept||course
FROM Offering;
```

### **Operations involving NULL**

If we operate with arithmetic operators on two values: a + b
 and a is NULL, the result is NULL

### Substituting NULL's in SELECT

• The *coalesce* function converts a NULL value to an actual value supplied as an argument

#### coalesce (column, value)

 Coalesce evaluates the arguments in order and returns the current value of the first expression that initially does not evaluate to NULL

Examples: coalesce (comission,0) coalesce (prerequisites, 'None')

## DISTINCT

# Relations can have duplicates in SQL

- A table can have duplicate tuples, unless this would violate an integrity constraint.
- And SELECT-FROM-WHERE statements leave duplicates in unless you say not to.
- Why?
  - Getting rid of duplicates is expensive!
  - We may want the duplicates because they tell us how many times something occurred.
- To eliminate duplicates need to explicitly use DISTINCT:
   SELECT DISTINCT \*

FROM R;



#### Bags - reminder

- SQL treats tables as "bags" (or "multisets") rather than sets.
- Bags are just like sets, but duplicates are allowed.
- {6, 2, 7, 1, 9} is a set (and a bag)
  {6, 2, 2, 7, 1, 9} is not a set, but is a bag.
- Like with sets, order doesn't matter.
   {6, 2, 7, 1, 9} = {1, 2, 6, 7, 9}

#### Impact of null values on DISTINCT

Does SELECT DISTINCT treat two NULLs as the same?

```
create table X(a int, b int);
insert into X values (1, 2), (null, 3), (null, 4);
select * from X
a | b
---+---
1 | 2
   3
    4
```

#### Impact of null values on DISTINCT

 If we ask for distinct values, the two NULLs are collapsed to one - SELECT DISTINCT has considered the two NULL values to be the same.

#### select distinct a from x;

a

1

(2 rows)

This behavior is DBMS-dependent

#### CASE in SELECT

• We can substitute values during selection using CASE clause:

**CASE** case\_expression

WHEN when\_expression\_1 THEN result\_1

WHEN when\_expression\_2 THEN result\_2

... [ ELSE result\_default]

END

#### CASE example

SELECT name, CASE WHEN country = 'USA' THEN 'domestic' ELSE 'intenational' END student\_group FROM Student

Student			
Name	Country	GPA	
Bob	Canada	3	
John	Britain	3	
Tom	USA	3.5	
Maria	Mexico	4	

#### CASE example: alternative syntax

SELECT name, CASE country WHEN 'USA' THEN 'domestic' ELSE 'intenational' END student\_group FROM Student

Student		
Name	Country	GPA
Bob	Canada	3
John	Britain	3
Tom	USA	3.5
Maria	Mexico	4

### ORDER BY clause

#### ORDER BY

- To put the tuples in order, add this as the final clause: ORDER BY *«attribute list»* [DESC]
- The default is ascending order; DESC overrides it to force descending order.
- The attribute list can include expressions: e.g., ORDER BY sales+rentals
- The ordering is the last thing done before the SELECT, so all attributes are still available.



#### **TOP-N** analysis

• Top-N queries are used to sort rows in a table and then to find the first-N largest (smallest) values

In PostgreSQL and in SQLite:
 SELECT gpa, name FROM Student
 ORDER BY gpa DESC
 LIMIT 5

#### Example 1: TOP-4 largest rooms

SELECT Building, RoomNo, Capacity FROM location ORDER BY Capacity DESC LIMIT 4;

### Example 2: TOP(BOTTOM)-3 lowest salaries

SELECT Lname, Fname, Salary FROM employee ORDER BY Salary LIMIT 3;