

# Databases

Jörg Endrullis

VU University Amsterdam

2015

## Overview

1. **SELECT-FROM-WHERE Blocks, Tuple Variables**
2. Subqueries, Non-Monotonic Constructs
3. Aggregations I: Aggregation Functions
4. Aggregations II: GROUP BY, HAVING
5. UNION, Conditional Expressions
6. ORDER BY
7. SQL-92 Joins, Outer Join

# Basic SQL Query Syntax

## Basic SQL query (extensions follow)

```
SELECT   $A_1, \dots, A_n$   
FROM     $R_1, \dots, R_m$   
WHERE    $C$ 
```

# Basic SQL Query Syntax

## Basic SQL query (extensions follow)

```
SELECT   $A_1, \dots, A_n$   
FROM     $R_1, \dots, R_m$   
WHERE    $C$ 
```

- The **FROM clause** declares which **table(s)** are accessed.

# Basic SQL Query Syntax

## Basic SQL query (extensions follow)

```
SELECT   $A_1, \dots, A_n$   
FROM     $R_1, \dots, R_m$   
WHERE    $C$ 
```

- The **FROM clause** declares which **table(s)** are accessed.
- The **WHERE clause** specifies a **condition** for rows in these tables that are considered in this query.

*The absence of  $C$  is equivalent to  $C \equiv \text{TRUE}$ .*

# Basic SQL Query Syntax

## Basic SQL query (extensions follow)

```
SELECT   $A_1, \dots, A_n$   
FROM     $R_1, \dots, R_m$   
WHERE    $C$ 
```

- The **FROM clause** declares which **table(s)** are accessed.
- The **WHERE clause** specifies a **condition** for rows in these tables that are considered in this query.  
*The absence of  $C$  is equivalent to  $C \equiv \text{TRUE}$ .*
- The **SELECT clause** specifies the attributes of the **result**.  
*Here  $*$   $\equiv$  output all attributes occurring in  $R_1, \dots, R_m$ .*

# Example Database

STUDENTS

<u>SID</u>	FIRST	LAST	EMAIL
101	Ann	Smith	...
102	Michael	Jones	(null)
103	Richard	Turner	...
104	Maria	Brown	...

EXERCISES

<u>CAT</u>	<u>ENO</u>	TOPIC	MAXPT
H	1	Rel.Alg.	10
H	2	SQL	10
M	1	SQL	14

RESULTS

<u>SID</u>	<u>CAT</u>	<u>ENO</u>	POINTS
101	H	1	10
101	H	2	8
101	M	1	12
102	H	1	9
102	H	2	9
102	M	1	10
103	H	1	5
103	M	1	7

# Tuple Variables

The FROM clause can be understood as **declaring variables** that **range over tuples** of a relation.



# Tuple Variables

The FROM clause can be understood as **declaring variables** that **range over tuples** of a relation.

EXERCISES			
<u>CAT</u>	<u>ENO</u>	<u>TOPIC</u>	<u>MAXPT</u>
H	1	Rel.Alg.	10
H	2	SQL	10
M	1	SQL	14

```
SELECT  E.ENO, E.TOPIC
FROM    EXERCISES E
WHERE   E.CAT = 'H'
```

# Tuple Variables

The FROM clause can be understood as **declaring variables** that **range over tuples** of a relation.

EXERCISES			
CAT	ENO	TOPIC	MAXPT
H	1	Rel.Alg.	10
H	2	SQL	10
M	1	SQL	14

```
SELECT E.ENO, E.TOPIC
FROM EXERCISES E
WHERE E.CAT = 'H'
```

- The query may be thought of as

```
for all E ∈ EXERCISES do
  if E.CAT = 'H' then
    print E.ENO, E.TOPIC
  end if
end for
```

# Tuple Variables

The FROM clause can be understood as **declaring variables** that **range over tuples** of a relation.

EXERCISES			
CAT	ENO	TOPIC	MAXPT
H	1	Rel.Alg.	10
H	2	SQL	10
M	1	SQL	14

```
SELECT  E.ENO, E.TOPIC
FROM    EXERCISES E
WHERE   E.CAT = 'H'
```

- The query may be thought of as

```
for all E ∈ EXERCISES do
  if E.CAT = 'H' then
    print E.ENO, E.TOPIC
  end if
end for
```

- **Tuple variable** E represents a **single row** of EXERCISES.  
*The loop assigns each row in succession.*

# Tuple Variables

The FROM clause can be understood as **declaring variables** that **range over tuples** of a relation.

EXERCISES			
CAT	ENO	TOPIC	MAXPT
H	1	Rel.Alg.	10
H	2	SQL	10
M	1	SQL	14

```
SELECT  E.ENO, E.TOPIC
FROM    EXERCISES E
WHERE   E.CAT = 'H'
```

1	Rel.Alg.
2	SQL

- The query may be thought of as

```
for all E ∈ EXERCISES do
  if E.CAT = 'H' then
    print E.ENO, E.TOPIC
  end if
end for
```

- **Tuple variable** E represents a **single row** of EXERCISES.  
*The loop assigns each row in succession.*

# Tuple Variables

```
EXERCISES(CAT,ENO,TOPIC,MAXPT)
```

- For each table in the FROM clause there is a tuple variable.

# Tuple Variables

EXERCISES(CAT, ENO, TOPIC, MAXPT)

- For each table in the FROM clause there is a tuple variable.
- If the name of the tuple variable is not given explicitly, the variable will have the name of the relation:

```
SELECT EXERCISES.ENO, EXERCISES.TOPIC
FROM EXERCISES
WHERE EXERCISES.CAT = 'H'
```

# Tuple Variables

```
EXERCISES(CAT, ENO, TOPIC, MAXPT)
```

- For each table in the FROM clause there is a tuple variable.
- If the name of the tuple variable is not given explicitly, the variable will have the name of the relation:

```
SELECT  EXERCISES.ENO, EXERCISES.TOPIC  
FROM    EXERCISES  
WHERE   EXERCISES.CAT = 'H'
```

In other words, FROM EXERCISES is understood as:

```
FROM EXERCISES EXERCISES
```

# Tuple Variables

```
EXERCISES(CAT, ENO, TOPIC, MAXPT)
```

- For each table in the FROM clause there is a tuple variable.
- If the the name of the tuple variable not given explicitly, the variable will have the name of the relation:

```
SELECT  EXERCISES.ENO, EXERCISES.TOPIC  
FROM    EXERCISES  
WHERE   EXERCISES.CAT = 'H'
```

In other words, FROM EXERCISES is understood as:

```
FROM EXERCISES EXERCISES
```

- If a tuple variable is explicitly declared, e.g.:

```
FROM EXERCISES E
```

then the implicit tuple variable EXERCISES is **not** declared and EXERCISES.ENO will yield an error.



# Attribute References

STUDENTS(SID, FIRST, LAST, EMAIL)

RESULTS(SID, CAT, ENO, POINTS)

- **Attributes are referenced** in the form

*R.A*

# Attribute References

STUDENTS(SID, FIRST, LAST, EMAIL)

RESULTS(SID, CAT, ENO, POINTS)

- **Attributes are referenced** in the form

$R.A$

- A reference to attribute  $A$  of variable  $R$  may be written as

$A$

if  $R$  is **the only tuple variable** with an attribute named  $A$ .

# Attribute References

STUDENTS(SID, FIRST, LAST, EMAIL)

RESULTS(SID, CAT, ENO, POINTS)

- **Attributes are referenced** in the form

$R.A$

- A reference to attribute  $A$  of variable  $R$  may be written as

$A$

if  $R$  is **the only tuple variable** with an attribute named  $A$ .

For example,

```
SELECT  CAT, ENO, POINTS
FROM    STUDENTS S, RESULTS R
WHERE   S.SID = R.SID
AND     FIRST = 'Ann' AND LAST = 'Smith'
```

- FIRST, LAST can only refer to S
- CAT, ENO, POINTS can only refer to R
- SID on its own is **ambiguous** (may refer to S or R)

# Attribute References

```
STUDENTS(SID, FIRST, LAST, EMAIL)
EXERCISES(CAT, ENO, TOPIC, MAXPT)
RESULTS(SID→STUDENTS, (CAT,ENO)→EXERCISES, POINTS)
```

- Consider this query:

```
SELECT ENO, SID, POINTS, MAXPT
FROM RESULTS R, EXERCISES E
WHERE R.ENO = E.ENO
AND R.CAT = 'H' AND E.CAT = 'H'
```

# Attribute References

```
STUDENTS(SID, FIRST, LAST, EMAIL)
EXERCISES(CAT, ENO, TOPIC, MAXPT)
RESULTS(SID→STUDENTS, (CAT,ENO)→EXERCISES, POINTS)
```

- Consider this query:

```
SELECT  ENO, SID, POINTS, MAXPT
FROM    RESULTS R, EXERCISES E
WHERE   R.ENO = E.ENO
AND     R.CAT = 'H' AND E.CAT = 'H'
```

Here ENO in the SELECT clause is ambiguous.

# Attribute References

```
STUDENTS(SID, FIRST, LAST, EMAIL)
EXERCISES(CAT, ENO, TOPIC, MAXPT)
RESULTS(SID→STUDENTS, (CAT,ENO)→EXERCISES, POINTS)
```

- Consider this query:

```
SELECT ENO, SID, POINTS, MAXPT
FROM   RESULTS R, EXERCISES E
WHERE  R.ENO = E.ENO
AND    R.CAT = 'H' AND E.CAT = 'H'
```

Here ENO in the SELECT clause is ambiguous.

Although forced to be equal by the join condition, SQL requires the user to specify unambiguously which of the ENO attributes (bound to R or E) is meant.

# Attribute References

```
STUDENTS(SID, FIRST, LAST, EMAIL)
EXERCISES(CAT, ENO, TOPIC, MAXPT)
RESULTS(SID→STUDENTS, (CAT,ENO)→EXERCISES, POINTS)
```

- Consider this query:

```
SELECT  ENO, SID, POINTS, MAXPT
FROM    RESULTS R, EXERCISES E
WHERE   R.ENO = E.ENO
AND     R.CAT = 'H' AND E.CAT = 'H'
```

Here ENO in the SELECT clause is ambiguous.

Although forced to be equal by the join condition, SQL requires the user to specify unambiguously which of the ENO attributes (bound to R or E) is meant.

The unambiguity rule thus is purely **syntactic** and does not depend on the query semantics.

# Joins

STUDENTS(SID, FIRST, LAST, EMAIL)

RESULTS(SID, CAT, ENO, POINTS)

Consider a query with two tuple variables:

```
SELECT   $A_1, \dots, A_n$ 
FROM    STUDENTS S, RESULTS R
WHERE   C
```



# Joins

STUDENTS(SID, FIRST, LAST, EMAIL)

RESULTS(SID, CAT, ENO, POINTS)

Consider a query with two tuple variables:

```
SELECT   $A_1, \dots, A_n$ 
FROM    STUDENTS S, RESULTS R
WHERE    $C$ 
```

- S ranges over 4 tuples in STUDENTS

# Joins

STUDENTS(SID, FIRST, LAST, EMAIL)

RESULTS(SID, CAT, ENO, POINTS)

Consider a query with two tuple variables:

```
SELECT   $A_1, \dots, A_n$ 
FROM    STUDENTS S, RESULTS R
WHERE   C
```

- S ranges over 4 tuples in STUDENTS
- R ranges over 8 tuples in RESULTS

# Joins

STUDENTS(SID, FIRST, LAST, EMAIL)

RESULTS(SID, CAT, ENO, POINTS)

Consider a query with two tuple variables:

```
SELECT   $A_1, \dots, A_n$ 
FROM    STUDENTS S, RESULTS R
WHERE   C
```

- S ranges over 4 tuples in STUDENTS
- R ranges over 8 tuples in RESULTS

In principle, all  $4 \times 8 = 32$  combinations will be considered:

```
for all S ∈ STUDENTS do
  for all R ∈ RESULTS do
    if C then
      print  $A_1, \dots, A_n$ 
    end if
  end for
end for
```

# Joins

A good DBMS will use a **better evaluation algorithm** (depending on the condition  $C$ ).

# Joins

A good DBMS will use a **better evaluation algorithm** (depending on the condition  $C$ ).

- This is the task of the **query optimiser**.

# Joins

A good DBMS will use a **better evaluation algorithm** (depending on the condition  $C$ ).

- This is the task of the **query optimiser**.
- For understanding the **semantics** of a query, the simple nested foreach algorithm suffices.

# Joins

A good DBMS will use a **better evaluation algorithm** (depending on the condition  $C$ ).

- This is the task of the **query optimiser**.
- For understanding the **semantics** of a query, the simple nested *foreach* algorithm suffices.
- The query optimizer may use any algorithm that produces the **exact same output** (except possibly the tuple order).

# Joins

A good DBMS will use a **better evaluation algorithm** (depending on the condition  $C$ ).

- This is the task of the **query optimiser**.
- For understanding the **semantics** of a query, the simple nested *foreach* algorithm suffices.
- The query optimizer may use any algorithm that produces the **exact same output** (except possibly the tuple order).

For example, if  $C$  contains the join condition

$$S.SID = R.SID$$

then the DBMS might execute the query efficiently by:

- looping over the tuples in RESULTS,
- finding the matching STUDENTS tuple via an **index** on STUDENT.SID



# Joins

A good DBMS will use a **better evaluation algorithm** (depending on the condition  $C$ ).

- This is the task of the **query optimiser**.
- For understanding the **semantics** of a query, the simple nested *foreach* algorithm suffices.
- The query optimizer may use any algorithm that produces the **exact same output** (except possibly the tuple order).

For example, if  $C$  contains the join condition

$$S.SID = R.SID$$

then the DBMS might execute the query efficiently by:

- looping over the tuples in RESULTS,
- finding the matching STUDENTS tuple via an **index** on STUDENT.SID

*DBMS typically create an index over the key attributes.*

# Joins

A **join** needs to be explicitly specified in the WHERE clause:

```
SELECT  R.CAT, R.ENO, R.POINTS
FROM    STUDENTS S, RESULTS R
WHERE   S.SID = R.SID      -- Join Condition
AND     S.FIRST = 'Ann' AND S.LAST = 'Smith'
```

# Joins

A **join** needs to be explicitly specified in the WHERE clause:

```
SELECT  R.CAT, R.ENO, R.POINTS
FROM    STUDENTS S, RESULTS R
WHERE   S.SID = R.SID      -- Join Condition
AND     S.FIRST = 'Ann' AND S.LAST = 'Smith'
```

STUDENTS			
<u>SID</u>	FIRST	LAST	EMAIL
101	Ann	Smith	...
102	Michael	Jones	(null)
103	Richard	Turner	...
104	Maria	Brown	...

RESULTS			
<u>SID</u>	<u>CAT</u>	<u>ENO</u>	<u>POINTS</u>
101	H	1	10
101	H	2	8
101	M	1	12
102	H	1	9
102	H	2	9
102	M	1	10
103	H	1	5
103	M	1	7

**Output of this query?**

```
SELECT  S.FIRST, S.LAST
FROM    STUDENTS S, RESULTS R
WHERE   R.CAT = 'H' AND R.ENO = 1
```

# Joins

Guideline: it is almost always an **error** if there are two tuples variables which are **not linked** (directly or indirectly) via join conditions.

# Joins

Guideline: it is almost always an **error** if there are two tuple variables which are **not linked** (directly or indirectly) via join conditions.

In this query, all three tuple variables are connected:

```
SELECT E.CAT, E.ENO, R.POINTS, E.MAXPT
FROM   STUDENTS S, RESULTS R, EXERCISES E
WHERE  S.SID = R.SID
AND    R.CAT = E.CAT AND R.ENO = E.ENO
AND    S.FIRST = 'Ann' AND S.LAST = 'Smith'
```

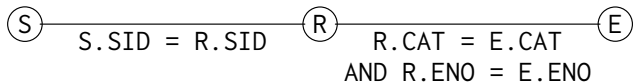
# Joins

Guideline: it is almost always an **error** if there are two tuple variables which are **not linked** (directly or indirectly) via join conditions.

In this query, all three tuple variables are connected:

```
SELECT E.CAT, E.ENO, R.POINTS, E.MAXPT
FROM   STUDENTS S, RESULTS R, EXERCISES E
WHERE  S.SID = R.SID
AND    R.CAT = E.CAT AND R.ENO = E.ENO
AND    S.FIRST = 'Ann' AND S.LAST = 'Smith'
```

- The tuple variable connection works as follows:



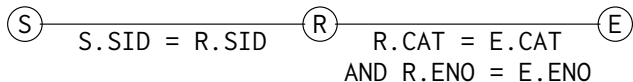
# Joins

Guideline: it is almost always an **error** if there are two tuple variables which are **not linked** (directly or indirectly) via join conditions.

In this query, all three tuple variables are connected:

```
SELECT E.CAT, E.ENO, R.POINTS, E.MAXPT
FROM   STUDENTS S, RESULTS R, EXERCISES E
WHERE  S.SID = R.SID
AND    R.CAT = E.CAT AND R.ENO = E.ENO
AND    S.FIRST = 'Ann' AND S.LAST = 'Smith'
```

- The tuple variable connection works as follows:



- Often (like in this example), the conditions correspond to the **foreign key relationships** between the tables.

# Query Formulation

**Formulate the following query in SQL**

Which are the topics of all exercises solved by Ann Smith?



# Query Formulation

**Formulate the following query in SQL**

Which are the topics of all exercises solved by Ann Smith?

To formulate this query:

# Query Formulation

**Formulate the following query in SQL**

Which are the topics of all exercises solved by Ann Smith?

To formulate this query:

- consider that Ann Smith is a student, we need

# Query Formulation

## Formulate the following query in SQL

Which are the topics of all exercises solved by Ann Smith?

To formulate this query:

- consider that Ann Smith is a student, we need
  - tuple variable  $s$  over STUDENTS

# Query Formulation

## Formulate the following query in SQL

Which are the topics of all exercises solved by Ann Smith?

To formulate this query:

- consider that Ann Smith is a student, we need
  - tuple variable S over STUDENTS
  - identifying condition in the WHERE clause  
S.FIRST = 'Ann' AND S.LAST = 'Smith'

# Query Formulation

## Formulate the following query in SQL

Which are the topics of all exercises solved by Ann Smith?

To formulate this query:

- consider that Ann Smith is a student, we need
  - tuple variable S over STUDENTS
  - identifying condition in the WHERE clause  
S.FIRST = 'Ann' AND S.LAST = 'Smith'
- exercise topics are of interest, so we need

# Query Formulation

## Formulate the following query in SQL

Which are the topics of all exercises solved by Ann Smith?

To formulate this query:

- consider that Ann Smith is a student, we need
  - tuple variable S over STUDENTS
  - identifying condition in the WHERE clause  
S.FIRST = 'Ann' AND S.LAST = 'Smith'
- exercise topics are of interest, so we need
  - tuple variable E over EXERCISES

# Query Formulation

## Formulate the following query in SQL

Which are the topics of all exercises solved by Ann Smith?

To formulate this query:

- consider that Ann Smith is a student, we need
  - tuple variable S over STUDENTS
  - identifying condition in the WHERE clause  
S.FIRST = 'Ann' AND S.LAST = 'Smith'
- exercise topics are of interest, so we need
  - tuple variable E over EXERCISES

Thus we start from:

```
SELECT DISTINCT E.TOPIC
FROM STUDENTS S, EXERCISES E
WHERE S.FIRST = 'Ann' AND S.LAST = 'Smith'
```

(DISTINCT since several exercises may have the same topic.)

# Query Formulation

```
SELECT DISTINCT E.TOPIC  
FROM STUDENTS S, EXERCISES E  
WHERE S.FIRST = 'Ann' AND S.LAST = 'Smith'
```



# Query Formulation

```
SELECT DISTINCT E.TOPIC  
FROM STUDENTS S, EXERCISES E  
WHERE S.FIRST = 'Ann' AND S.LAST = 'Smith'
```

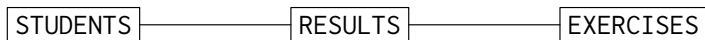
Note: S and E are still **unconnected**:

# Query Formulation

```
SELECT DISTINCT E.TOPIC
FROM STUDENTS S, EXERCISES E
WHERE S.FIRST = 'Ann' AND S.LAST = 'Smith'
```

Note: S and E are still **unconnected**:

- The **connection graph** of the tables in a database schema (connections are foreign key relations) helps in understanding the connection requirements:



STUDENTS(SID, FIRST, LAST, EMAIL)

EXERCISES(CAT, ENO, TOPIC, MAXPT)

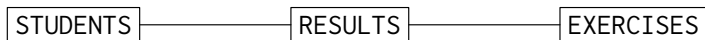
RESULTS(SID→STUDENTS, (CAT,ENO)→EXERCISES, POINTS)

# Query Formulation

```
SELECT DISTINCT E.TOPIC
FROM STUDENTS S, EXERCISES E
WHERE S.FIRST = 'Ann' AND S.LAST = 'Smith'
```

Note: S and E are still **unconnected**:

- The **connection graph** of the tables in a database schema (connections are foreign key relations) helps in understanding the connection requirements:



```
STUDENTS(SID, FIRST, LAST, EMAIL)
EXERCISES(CAT, ENO, TOPIC, MAXPT)
RESULTS(SID→STUDENTS, (CAT,ENO)→EXERCISES, POINTS)
```

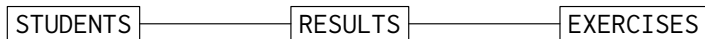
- We see that the S–E connection is **indirect** and needs to be established via a tuple variable R over RESULTS:

# Query Formulation

```
SELECT DISTINCT E.TOPIC
FROM STUDENTS S, EXERCISES E
WHERE S.FIRST = 'Ann' AND S.LAST = 'Smith'
```

Note: S and E are still **unconnected**:

- The **connection graph** of the tables in a database schema (connections are foreign key relations) helps in understanding the connection requirements:



```
STUDENTS(SID, FIRST, LAST, EMAIL)
EXERCISES(CAT, ENO, TOPIC, MAXPT)
RESULTS(SID→STUDENTS, (CAT,ENO)→EXERCISES, POINTS)
```

- We see that the S–E connection is **indirect** and needs to be established via a tuple variable R over RESULTS:

```
S.SID = R.SID AND R.CAT = E.CAT AND R.ENO = E.ENO
```

# Query Formulation

It is not always that trivial. . .

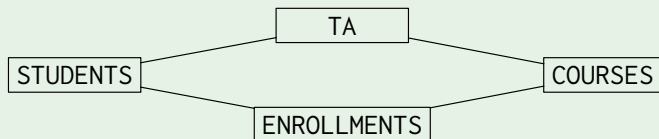
The connection graph may contain **cycles**, which makes the selection of the “right path” more difficult (and error-prone).

# Query Formulation

It is not always that trivial. . .

The connection graph may contain **cycles**, which makes the selection of the “right path” more difficult (and error-prone).

Consider a course registration database that also contains teaching assistants (TA) assignments:



# Unnecessary Joins

Do not join **more** tables than needed.

*Query will run slowly if the optimizer overlooks the redundancy.*

# Unnecessary Joins

Do not join **more** tables than needed.

*Query will run slowly if the optimizer overlooks the redundancy.*

```
STUDENTS(SID, FIRST, LAST, EMAIL)
EXERCISES(CAT, ENO, TOPIC, MAXPT)
RESULTS(SID→STUDENTS, (CAT,ENO)→EXERCISES, POINTS)
```

## Results for homework 1

```
SELECT  R.SID, R.POINTS
FROM    RESULTS R, EXERCISES E
WHERE   R.CAT = E.CAT AND R.ENO = E.ENO
AND     E.CAT = 'H' AND E.ENO = 1
```

## Will the following query produce the same results?

```
SELECT  SID, POINTS
FROM    RESULTS R
WHERE   R.CAT = 'H' AND R.ENO = 1
```



# Unnecessary Joins

EXERCISES			
CAT	ENO	TOPIC	MAXPT
H	1	Rel.Alg.	10
H	2	SQL	10
M	1	SQL	14

RESULTS			
SID	CAT	ENO	POINTS
101	H	1	10
101	H	2	8
101	M	1	12
102	H	1	9
102	H	2	9
102	M	1	10
103	H	1	5
103	M	1	7

**What will be the result of this query?**

```
SELECT R.SID, R.POINTS
FROM RESULTS R, EXERCISES E
WHERE R.CAT = 'H' AND R.ENO = 1
```

# Unnecessary Joins

STUDENTS			
<u>SID</u>	<u>FIRST</u>	<u>LAST</u>	<u>EMAIL</u>
101	Ann	Smith	...
102	Michael	Jones	(null)
103	Richard	Turner	...
104	Maria	Brown	...

RESULTS			
<u>SID</u>	<u>CAT</u>	<u>ENO</u>	<u>POINTS</u>
101	H	1	10
101	H	2	8
101	M	1	12
102	H	1	9
102	H	2	9
102	M	1	10
103	H	1	5
103	M	1	7

**Is there any difference between these two queries?**

```
SELECT S.FIRST, S.LAST  
FROM STUDENTS S
```

```
SELECT DISTINCT S.FIRST, S.LAST  
FROM STUDENTS S, RESULTS R  
WHERE S.SID = R.SID
```

# Self Joins

In some query scenarios, we might have to consider **more than one tuple of the same relation** to generate a result tuple.

# Self Joins

In some query scenarios, we might have to consider **more than one tuple of the same relation** to generate a result tuple.

**Is there a student with 9 points for both, homework 1 & 2?**

# Self Joins

In some query scenarios, we might have to consider **more than one tuple of the same relation** to generate a result tuple.

**Is there a student with 9 points for both, homework 1 & 2?**

```
SELECT  S.FIRST, S.LAST
FROM    STUDENTS S, RESULTS H1, RESULTS H2
WHERE   S.SID = H1.SID AND S.SID = H2.SID
AND     H1.CAT = 'H' AND H1.ENO = 1
AND     H2.CAT = 'H' AND H2.ENO = 2
AND     H1.POINTS = 9 AND H2.POINTS = 9
```

# Self Joins

**Find students who solved at least two exercises.**

*(This may also be solved using aggregations.)*

```
SELECT  S.FIRST, S.LAST
FROM    STUDENTS S, RESULTS E1, RESULTS E2
WHERE   S.SID = E1.SID AND S.SID = E2.SID
```

# Self Joins

**Find students who solved at least two exercises.**

*(This may also be solved using aggregations.)*

```
SELECT  S.FIRST, S.LAST  
FROM    STUDENTS S, RESULTS E1, RESULTS E2  
WHERE   S.SID = E1.SID AND S.SID = E2.SID
```

**“Unexpected” result**

What is going wrong here?

# Self Joins

**Find students who solved at least two exercises.**

*(This may also be solved using aggregations.)*

```
SELECT  S.FIRST, S.LAST
FROM    STUDENTS S, RESULTS E1, RESULTS E2
WHERE   S.SID = E1.SID AND S.SID = E2.SID
```

**“Unexpected” result**

What is going wrong here?

We need to ensure that E1 and E2 refer to distinct exercises:

```
⋮
AND (E1.CAT <> E2.CAT OR E1.ENO <> E2.ENO)
```



# Duplicate Elimination

A core difference between SQL and relational algebra is that **duplicates have to explicitly eliminated** in SQL.

# Duplicate Elimination

A core difference between SQL and relational algebra is that **duplicates have to explicitly eliminated** in SQL.

Which exercises have been solved by at least one student?

```
SELECT CAT, ENO
FROM RESULTS
```

CAT	ENO
H	1
H	2
M	1
H	1
⋮	⋮

# Duplicate Elimination

A core difference between SQL and relational algebra is that **duplicates have to explicitly eliminated** in SQL.

Which exercises have been solved by at least one student?

```
SELECT  CAT, ENO
FROM    RESULTS
```

CAT	ENO
H	1
H	2
M	1
H	1
⋮	⋮

The `DISTINCT` modifier may be applied to the `SELECT` clause to request explicit duplicate row elimination

# Duplicate Elimination

A core difference between SQL and relational algebra is that **duplicates have to explicitly eliminated** in SQL.

Which exercises have been solved by at least one student?

```
SELECT  CAT, ENO
FROM    RESULTS
```

CAT	ENO
H	1
H	2
M	1
H	1
⋮	⋮

The `DISTINCT` modifier may be applied to the `SELECT` clause to request explicit duplicate row elimination

```
SELECT  DISTINCT CAT, ENO
FROM    RESULTS
```

CAT	ENO
H	1
H	2
M	1

# Duplicate Elimination

## Sufficient condition for superfluous DISTINCT:

1. Let  $\mathcal{K}$  be the set of attributes in the SELECT clause.
2. Add to  $\mathcal{K}$  attributes  $A$  such that
  - $A = c$  (constant  $c$ ) appears in the WHERE clause
  - $A = B$  ( $B \in \mathcal{K}$ ) appears in the WHERE clause
  - if  $\mathcal{K}$  contains a key of a tuple variable, add all attributes of that variable.

Here we assume that the WHERE clause is a conjunction.

Repeat 2 until  $\mathcal{K}$  is stable.

If  $\mathcal{K}$  **contains a key of every tuple variable** listed under FROM, then DISTINCT is superfluous.

# Duplicate Elimination

## Sufficient condition for superfluous DISTINCT:

1. Let  $\mathcal{K}$  be the set of attributes in the SELECT clause.
2. Add to  $\mathcal{K}$  attributes  $A$  such that
  - $A = c$  (constant  $c$ ) appears in the WHERE clause
  - $A = B$  ( $B \in \mathcal{K}$ ) appears in the WHERE clause
  - if  $\mathcal{K}$  contains a key of a tuple variable, add all attributes of that variable.

Here we assume that the WHERE clause is a conjunction.

Repeat 2 until  $\mathcal{K}$  is stable.

If  $\mathcal{K}$  **contains a key of every tuple variable** listed under FROM, then DISTINCT is superfluous.

Intuition behind the algorithm: think of  $\mathcal{K}$  as the set of attributes that are uniquely determined by the result.

# Duplicate Elimination

```
SELECT DISTINCT S.FIRST, S.LAST, R.ENO, R.POINTS  
FROM STUDENTS S, RESULTS R  
WHERE R.CAT = 'H' AND R.SID = S.SID
```

Let us assume that (FIRST, LAST) is a key for STUDENTS.

# Duplicate Elimination

```
SELECT DISTINCT S.FIRST, S.LAST, R.ENO, R.POINTS  
FROM STUDENTS S, RESULTS R  
WHERE R.CAT = 'H' AND R.SID = S.SID
```

Let us assume that (FIRST, LAST) is a key for STUDENTS.

1. Initialize  $\mathcal{K} =$



# Duplicate Elimination

```
SELECT DISTINCT S.FIRST, S.LAST, R.ENO, R.POINTS  
FROM STUDENTS S, RESULTS R  
WHERE R.CAT = 'H' AND R.SID = S.SID
```

Let us assume that (FIRST, LAST) is a key for STUDENTS.

1. Initialize  $\mathcal{K} = \{S.FIRST, S.LAST, R.ENO, R.POINTS\}$ .

# Duplicate Elimination

```
SELECT DISTINCT S.FIRST, S.LAST, R.ENO, R.POINTS
FROM   STUDENTS S, RESULTS R
WHERE  R.CAT = 'H' AND R.SID = S.SID
```

Let us assume that (FIRST, LAST) is a key for STUDENTS.

1. Initialize  $\mathcal{K} = \{S.FIRST, S.LAST, R.ENO, R.POINTS\}$ .
2.  $\mathcal{K} + \{R.CAT\}$  because of the conjunct  $R.CAT = 'H'$

# Duplicate Elimination

```
SELECT DISTINCT S.FIRST, S.LAST, R.ENO, R.POINTS  
FROM STUDENTS S, RESULTS R  
WHERE R.CAT = 'H' AND R.SID = S.SID
```

Let us assume that (FIRST, LAST) is a key for STUDENTS.

1. Initialize  $\mathcal{K} = \{S.FIRST, S.LAST, R.ENO, R.POINTS\}$ .
2.  $\mathcal{K} + \{R.CAT\}$  because of the conjunct  $R.CAT = 'H'$
2.  $\mathcal{K} + \{S.SID, S.EMAIL\}$  as  $\mathcal{K}$  contains a key of STUDENTS

# Duplicate Elimination

```
SELECT DISTINCT S.FIRST, S.LAST, R.ENO, R.POINTS
FROM   STUDENTS S, RESULTS R
WHERE  R.CAT = 'H' AND R.SID = S.SID
```

Let us assume that (FIRST, LAST) is a key for STUDENTS.

1. Initialize  $\mathcal{K} = \{S.FIRST, S.LAST, R.ENO, R.POINTS\}$ .
2.  $\mathcal{K} + \{R.CAT\}$  because of the conjunct  $R.CAT = 'H'$
2.  $\mathcal{K} + \{S.SID, S.EMAIL\}$  as  $\mathcal{K}$  contains a key of STUDENTS
2.  $\mathcal{K} + \{R.SID\}$  because of the conjunct  $S.SID = R.SID$

# Duplicate Elimination

```
SELECT DISTINCT S.FIRST, S.LAST, R.ENO, R.POINTS
FROM STUDENTS S, RESULTS R
WHERE R.CAT = 'H' AND R.SID = S.SID
```

Let us assume that (FIRST, LAST) is a key for STUDENTS.

1. Initialize  $\mathcal{K} = \{S.FIRST, S.LAST, R.ENO, R.POINTS\}$ .
2.  $\mathcal{K} + \{R.CAT\}$  because of the conjunct  $R.CAT = 'H'$
2.  $\mathcal{K} + \{S.SID, S.EMAIL\}$  as  $\mathcal{K}$  contains a key of STUDENTS
2.  $\mathcal{K} + \{R.SID\}$  because of the conjunct  $S.SID = R.SID$

$\mathcal{K}$  contains a key of

- STUDENTS S (S.FIRST, S.LAST) and
- RESULTS R (R.SID, R.CAT, R.ENO)

# Duplicate Elimination

```
SELECT DISTINCT S.FIRST, S.LAST, R.ENO, R.POINTS
FROM STUDENTS S, RESULTS R
WHERE R.CAT = 'H' AND R.SID = S.SID
```

Let us assume that (FIRST, LAST) is a key for STUDENTS.

1. Initialize  $\mathcal{K} = \{S.FIRST, S.LAST, R.ENO, R.POINTS\}$ .
2.  $\mathcal{K} + \{R.CAT\}$  because of the conjunct  $R.CAT = 'H'$
2.  $\mathcal{K} + \{S.SID, S.EMAIL\}$  as  $\mathcal{K}$  contains a key of STUDENTS
2.  $\mathcal{K} + \{R.SID\}$  because of the conjunct  $S.SID = R.SID$

$\mathcal{K}$  contains a key of

- STUDENTS S (S.FIRST, S.LAST) and
- RESULTS R (R.SID, R.CAT, R.ENO)

Thus DISTINCT is superfluous.

# Duplicate Elimination

```
SELECT DISTINCT S.FIRST, S.LAST, R.ENO, R.POINTS
FROM STUDENTS S, RESULTS R
WHERE R.CAT = 'H' AND R.SID = S.SID
```

Let us assume that (FIRST, LAST) is a key for STUDENTS.

1. Initialize  $\mathcal{K} = \{S.FIRST, S.LAST, R.ENO, R.POINTS\}$ .
2.  $\mathcal{K} + \{R.CAT\}$  because of the conjunct  $R.CAT = 'H'$
2.  $\mathcal{K} + \{S.SID, S.EMAIL\}$  as  $\mathcal{K}$  contains a key of STUDENTS
2.  $\mathcal{K} + \{R.SID\}$  because of the conjunct  $S.SID = R.SID$

$\mathcal{K}$  contains a key of

- STUDENTS S (S.FIRST, S.LAST) and
- RESULTS R (R.SID, R.CAT, R.ENO)

Thus DISTINCT is superfluous.

If FIRST, LAST were no key of STUDENTS, this test would not succeed (and rightly so).

# Query Formulation Traps

- **Missing join conditions** (very common).



# Query Formulation Traps

- **Missing join conditions** (very common).
- **Unnecessary joins** (may slow query down significantly).

# Query Formulation Traps

- **Missing join conditions** (very common).
- **Unnecessary joins** (may slow query down significantly).
- **Self joins:** incorrect treatment of multiple tuple variables which range over the same relation (missing (in)equality conditions).

# Query Formulation Traps

- **Missing join conditions** (very common).
- **Unnecessary joins** (may slow query down significantly).
- **Self joins:** incorrect treatment of multiple tuple variables which range over the same relation (missing (in)equality conditions).
- **Unexpected duplicates**, often an indicator for faulty queries (adding `DISTINCT` is no cure here).

# Query Formulation Traps

- **Missing join conditions** (very common).
- **Unnecessary joins** (may slow query down significantly).
- **Self joins:** incorrect treatment of multiple tuple variables which range over the same relation (missing (in)equality conditions).
- **Unexpected duplicates**, often an indicator for faulty queries (adding DISTINCT is no cure here).
- **Unnecessary DISTINCT.**

*Although today's query optimizer are probably more "clever" than the average SQL user in proving the absence of duplicates.*

## Overview

1. SELECT-FROM-WHERE Blocks, Tuple Variables
2. Subqueries, Non-Monotonic Constructs
3. Aggregations I: Aggregation Functions
4. Aggregations II: GROUP BY, HAVING
5. UNION, Conditional Expressions
6. ORDER BY
7. SQL-92 Joins, Outer Join

# Non-Monotonic Behaviour

SQL queries using only the constructs introduced so far compute **monotonic functions** on the database state:

- if further rows gets **inserted**, these queries yield a **superset** of rows.

# Non-Monotonic Behaviour

SQL queries using only the constructs introduced so far compute **monotonic functions** on the database state:

- if further rows gets **inserted**, these queries yield a **superset** of rows.

However, not all queries behave monotonically in this way.

## Example of a non-monotonic query

Query: find students who have not submitted any homework.

# Non-Monotonic Behaviour

SQL queries using only the constructs introduced so far compute **monotonic functions** on the database state:

- if further rows gets **inserted**, these queries yield a **superset** of rows.

However, not all queries behave monotonically in this way.

## Example of a non-monotonic query

Query: find students who have not submitted any homework.

- In the current DB state, Maria Brown would be a correct answer.



# Non-Monotonic Behaviour

SQL queries using only the constructs introduced so far compute **monotonic functions** on the database state:

- if further rows gets **inserted**, these queries yield a **superset** of rows.

However, not all queries behave monotonically in this way.

## Example of a non-monotonic query

Query: find students who have not submitted any homework.

- In the current DB state, Maria Brown would be a correct answer.
- `INSERT INTO RESULTS VALUES (104, 'H', 1, 8)` would invalidate this answer.

# Non-Monotonic Behaviour

SQL queries using only the constructs introduced so far compute **monotonic functions** on the database state:

- if further rows gets **inserted**, these queries yield a **superset** of rows.

However, not all queries behave monotonically in this way.

## Example of a non-monotonic query

Query: find students who have not submitted any homework.

- In the current DB state, Maria Brown would be a correct answer.
- `INSERT INTO RESULTS VALUES (104, 'H', 1, 8)` would invalidate this answer.

Such queries **cannot** be formulated with the SQL constructs introduced so far.

# Non-Monotonic Behaviour

- In natural language, queries that contain formulations like
  - “**there is no**”,
  - “**does not exists**”,
  - ...

indicate non-monotonic behaviour.  
(**negated existential quantification**).

# Non-Monotonic Behaviour

- In natural language, queries that contain formulations like
  - **“there is no”**,
  - **“does not exists”**,
  - ...

indicate non-monotonic behaviour.  
(**negated existential quantification**).

- Furthermore,
  - **“for all”**,
  - **“the minimum/maximum”**

also indicate non-monotonic behaviour.  
(**universally quantification**).

# Non-Monotonic Behaviour

- In natural language, queries that contain formulations like
  - “**there is no**”,
  - “**does not exists**”,
  - ...

indicate non-monotonic behaviour.  
(**negated existential quantification**).

- Furthermore,
  - “**for all**”,
  - “**the minimum/maximum**”

also indicate non-monotonic behaviour.  
(**universally quantification**).

In an equivalent SQL formulation of such queries, this boils down to a test whether a **query yields a (non-)empty result**.

# NOT IN

With

- IN ( $\in$ ) and
- NOT IN ( $\notin$ )

it is possible to check whether an attribute value appears in a set of values computed by another SQL **subquery**.

# NOT IN

With

- IN ( $\in$ ) and
- NOT IN ( $\notin$ )

it is possible to check whether an attribute value appears in a set of values computed by another SQL **subquery**.

## Students without any homework result

```
SELECT FIRST, LAST
FROM STUDENTS
WHERE SID NOT IN (SELECT SID
                  FROM RESULTS
                  WHERE CAT = 'H')
```

FIRST	LAST
Maria	Brown

# NOT IN

```
SELECT  FIRST, LAST
FROM    STUDENTS
WHERE   SID NOT IN (SELECT  SID
                    FROM    RESULTS
                    WHERE   CAT = 'H')
```

At least conceptually, the **subquery** is evaluated before the evaluation of the **main query** starts:

STUDENTS			
<u>SID</u>	FIRST	LAST	EMAIL
101	Ann	Smith	...
102	Michael	Jones	(null)
103	Richard	Turner	...
104	Maria	Brown	...

Subquery result	
SID	
101	
101	
102	
102	
103	



# NOT IN

```
SELECT  FIRST, LAST
FROM    STUDENTS
WHERE   SID NOT IN (SELECT  SID
                    FROM    RESULTS
                    WHERE   CAT = 'H')
```

At least conceptually, the **subquery** is evaluated before the evaluation of the **main query** starts:

STUDENTS			
<u>SID</u>	FIRST	LAST	EMAIL
101	Ann	Smith	...
102	Michael	Jones	(null)
103	Richard	Turner	...
104	Maria	Brown	...

Subquery result	
SID	
	101
	101
	102
	102
	103

Then, for every STUDENTS tuple, a matching SID is searched in the subquery result. If there is none, the tuple is output.

# NOT IN

EXERCISES(CAT, ENO, TOPIC, MAXPT)

RESULTS(SID, CAT, ENO, POINTS)

**Topics of homeworks solved by at least one student.**

# NOT IN

EXERCISES(CAT, ENO, TOPIC, MAXPT)

RESULTS(SID, CAT, ENO, POINTS)

## Topics of homeworks solved by at least one student.

```
SELECT TOPIC
FROM EXERCISES
WHERE CAT = 'H' AND ENO IN (SELECT ENO
                             FROM RESULTS
                             WHERE CAT = 'H')
```

# NOT IN

```
EXERCISES(CAT, ENO, TOPIC, MAXPT)
```

```
RESULTS(SID, CAT, ENO, POINTS)
```

## Topics of homeworks solved by at least one student.

```
SELECT TOPIC
FROM EXERCISES
WHERE CAT = 'H' AND ENO IN (SELECT ENO
                             FROM RESULTS
                             WHERE CAT = 'H')
```

## Is there a difference to this query? (with or without DISTINCT)

```
SELECT DISTINCT TOPIC
FROM EXERCISES E, RESULTS R
WHERE E.CAT = 'H'
      AND E.ENO = R.ENO
      AND R.CAT = 'H'
```

# NOT IN

- In SQL-86,  
subquery is required to deliver a **single output column**
- In SQL-92,  
comparisons where extended to the tuple level.

It is thus valid to write, e.g.:

```
WHERE (A,B) NOT IN (SELECT C,D FROM . . . )
```

# NOT EXISTS

The construct NOT EXISTS enables the main (or outer) query to check whether the **subquery result is empty**.

# NOT EXISTS

The construct NOT EXISTS enables the main (or outer) query to check whether the **subquery result is empty**.

## Students that have not submitted any homework

```
SELECT  FIRST, LAST
FROM    STUDENTS S
WHERE   NOT EXISTS ( SELECT  *
                     FROM    RESULTS R
                     WHERE   R.CAT = 'H'
                     AND     R.SID = S.SID )
```

# NOT EXISTS

The construct NOT EXISTS enables the main (or outer) query to check whether the **subquery result is empty**.

## Students that have not submitted any homework

```
SELECT  FIRST, LAST
FROM    STUDENTS S
WHERE   NOT EXISTS ( SELECT  *
                    FROM    RESULTS R
                    WHERE   R.CAT = 'H'
                    AND     R.SID = S.SID )
```

- In the subquery, tuple variables declared in the FROM clause of the outer query may be referenced.

*You may also do so for IN subqueries.*



# NOT EXISTS

The construct NOT EXISTS enables the main (or outer) query to check whether the **subquery result is empty**.

## Students that have not submitted any homework

```
SELECT  FIRST, LAST
FROM    STUDENTS S
WHERE   NOT EXISTS ( SELECT  *
                     FROM    RESULTS R
                     WHERE   R.CAT = 'H'
                     AND     R.SID = S.SID)
```

- In the subquery, tuple variables declared in the FROM clause of the outer query may be referenced.  
*You may also do so for IN subqueries.*
- In this case, the outer query and subquery are **correlated**.

# NOT EXISTS

The construct NOT EXISTS enables the main (or outer) query to check whether the **subquery result is empty**.

## Students that have not submitted any homework

```
SELECT  FIRST, LAST
FROM    STUDENTS S
WHERE   NOT EXISTS ( SELECT  *
                    FROM    RESULTS R
                    WHERE   R.CAT = 'H'
                    AND    R.SID = S.SID)
```

- In the subquery, tuple variables declared in the FROM clause of the outer query may be referenced.  
*You may also do so for IN subqueries.*
- In this case, the outer query and subquery are **correlated**.
- The subquery is “**parameterized**”.

# NOT EXISTS

## Students that have not submitted any homework

```
SELECT  FIRST, LAST
FROM    STUDENTS S
WHERE   NOT EXISTS ( SELECT  *
                     FROM    RESULTS R
                     WHERE   R.CAT = 'H'
                     AND     R.SID = S.SID )
```

# NOT EXISTS

## Students that have not submitted any homework

```
SELECT  FIRST, LAST
FROM    STUDENTS S
WHERE   NOT EXISTS ( SELECT  *
                    FROM    RESULTS R
                    WHERE   R.CAT = 'H'
                    AND     R.SID = S.SID)
```

Tuple variable s loops over the four rows in STUDENTS.

# NOT EXISTS

## Students that have not submitted any homework

```
SELECT  FIRST, LAST
FROM    STUDENTS S
WHERE   NOT EXISTS ( SELECT  *
                     FROM    RESULTS R
                     WHERE   R.CAT = 'H'
                     AND     R.SID = S.SID )
```

Tuple variable *S* loops over the four rows in STUDENTS.

Conceptually, the subquery is evaluated four times (with *S.SID* bound to the current *SID* value).

# NOT EXISTS

## Students that have not submitted any homework

```
SELECT  FIRST, LAST
FROM    STUDENTS S
WHERE   NOT EXISTS ( SELECT  *
                     FROM    RESULTS R
                     WHERE   R.CAT = 'H'
                     AND     R.SID = S.SID )
```

Tuple variable *S* loops over the four rows in STUDENTS.

Conceptually, the subquery is evaluated four times (with *S.SID* bound to the current *SID* value).

*The DBMS is free to choose a more efficient equivalent evaluation strategy (cf. **query unnesting**).*

# NOT EXISTS

## Students that have not submitted any homework

```
SELECT  FIRST, LAST
FROM    STUDENTS S
WHERE   NOT EXISTS ( SELECT  *
                     FROM    RESULTS R
                     WHERE   R.CAT = 'H'
                     AND     R.SID = S.SID)
```

- “First,” s is bound to the STUDENTS tuple

SID	FIRST	LAST	EMAIL
101	Ann	Smith	...

# NOT EXISTS

## Students that have not submitted any homework

```
SELECT  FIRST, LAST
FROM    STUDENTS S
WHERE   NOT EXISTS ( SELECT  *
                    FROM    RESULTS R
                    WHERE   R.CAT = 'H'
                    AND     R.SID = S.SID)
```

- “First,” s is bound to the STUDENTS tuple

SID	FIRST	LAST	EMAIL
101	Ann	Smith	...

- In the subquery, S.SID is “replaced by” 101:

```
SELECT  *
FROM    RESULTS R
WHERE   R.CAT = 'H'
AND     R.SID = 101
```

SID	CAT	ENO	POINTS
101	H	1	10
101	H	2	8



# NOT EXISTS

## Students that have not submitted any homework

```
SELECT  FIRST, LAST
FROM    STUDENTS S
WHERE   NOT EXISTS ( SELECT  *
                     FROM    RESULTS R
                     WHERE   R.CAT = 'H'
                     AND     R.SID = S.SID)
```

- “First,” s is bound to the STUDENTS tuple

SID	FIRST	LAST	EMAIL
101	Ann	Smith	...

- In the subquery, S.SID is “replaced by” 101:

```
SELECT  *
FROM    RESULTS R
WHERE   R.CAT = 'H'
AND     R.SID = 101
```

SID	CAT	ENO	POINTS
101	H	1	10
101	H	2	8

- Since the result is non-empty, the NOT EXISTS in the outer query is not satisfied **for this S**.

# NOT EXISTS

## Students that have not submitted any homework

```
SELECT  FIRST, LAST
FROM    STUDENTS S
WHERE   NOT EXISTS ( SELECT  *
                     FROM    RESULTS R
                     WHERE   R.CAT = 'H'
                     AND     R.SID = S.SID)
```

- “Finally,” s is bound to the STUDENTS tuple

SID	FIRST	LAST	EMAIL
104	Maria	Brown	...

# NOT EXISTS

## Students that have not submitted any homework

```
SELECT  FIRST, LAST
FROM    STUDENTS S
WHERE   NOT EXISTS ( SELECT  *
                    FROM    RESULTS R
                    WHERE   R.CAT = 'H'
                    AND     R.SID = S.SID)
```

- “Finally,” s is bound to the STUDENTS tuple

SID	FIRST	LAST	EMAIL
104	Maria	Brown	...

- In the subquery, S.SID is “replaced by” 104:

```
SELECT  *
FROM    RESULTS R
WHERE   R.CAT = 'H'
AND     R.SID = 104
```

SID	CAT	ENO	POINTS
(no rows selected)			

# NOT EXISTS

## Students that have not submitted any homework

```
SELECT  FIRST, LAST
FROM    STUDENTS S
WHERE   NOT EXISTS ( SELECT  *
                     FROM    RESULTS R
                     WHERE   R.CAT = 'H'
                     AND     R.SID = S.SID )
```

- “Finally,” s is bound to the STUDENTS tuple

SID	FIRST	LAST	EMAIL
104	Maria	Brown	...

- In the subquery, S.SID is “replaced by” 104:

```
SELECT  *
FROM    RESULTS R
WHERE   R.CAT = 'H'
AND     R.SID = 104
```

SID	CAT	ENO	POINTS
(no rows selected)			

- Since the result is empty, the NOT EXISTS in the outer query is satisfied and Maria Brown is output.

# NOT EXISTS

While in the subquery tuple variables from outer query may be referenced, the **converse is illegal!**

# NOT EXISTS

While in the subquery tuple variables from outer query may be referenced, the **converse is illegal!**

## Wrong!

```
SELECT  FIRST, LAST, R.ENO
FROM    STUDENTS S
WHERE   NOT EXISTS ( SELECT  *
                     FROM    RESULTS R
                     WHERE   R.CAT = 'H'
                     AND     R.SID = S.SID)
```

# NOT EXISTS

While in the subquery tuple variables from outer query may be referenced, the **converse is illegal!**

## Wrong!

```
SELECT  FIRST, LAST, R.ENO
FROM    STUDENTS S
WHERE   NOT EXISTS ( SELECT  *
                    FROM    RESULTS R
                    WHERE   R.CAT = 'H'
                    AND     R.SID = S.SID)
```

- Compare this to **variable scoping** (global/local variables) in block-structured programming languages (Java, C). Subquery tuple variables declarations are “local.”

# NOT EXISTS

**Non-correlated subqueries** with NOT EXISTS are almost always an indication of an error!



# NOT EXISTS

**Non-correlated subqueries** with NOT EXISTS are almost always an indication of an error!

## Wrong!

```
SELECT  FIRST, LAST
FROM    STUDENTS S
WHERE   NOT EXISTS (SELECT  *
                    FROM    RESULTS R
                    WHERE   CAT = 'H')
```

# NOT EXISTS

**Non-correlated subqueries** with NOT EXISTS are almost always an indication of an error!

## Wrong!

```
SELECT  FIRST, LAST
FROM    STUDENTS S
WHERE   NOT EXISTS (SELECT *
                   FROM  RESULTS R
                   WHERE  CAT = 'H')
```

- If there is at least one tuple in RESULTS, the overall result will be empty.

# NOT EXISTS

**Non-correlated subqueries** with NOT EXISTS are almost always an indication of an error!

## Wrong!

```
SELECT  FIRST, LAST
FROM    STUDENTS S
WHERE   NOT EXISTS (SELECT *
                    FROM  RESULTS R
                    WHERE  CAT = 'H')
```

- If there is at least one tuple in RESULTS, the overall result will be empty.

Non-correlated subqueries evaluate to a set/relation **constant** and may make perfect sense (e.g., when used with (NOT) IN).

# NOT EXISTS

It is legal SQL syntax to use EXISTS without negation:

## Who has submitted at least one homework?

```
SELECT  SID, FIRST, LAST
FROM    STUDENTS S
WHERE   EXISTS (SELECT *
                FROM  RESULTS R
                WHERE  R.SID = S.SID
                AND    R.CAT = 'H')
```

# NOT EXISTS

It is legal SQL syntax to use EXISTS without negation:

## Who has submitted at least one homework?

```
SELECT  SID, FIRST, LAST
FROM    STUDENTS S
WHERE   EXISTS (SELECT *
                FROM  RESULTS R
                WHERE  R.SID = S.SID
                AND    R.CAT = 'H')
```

## Can we reformulate the above without using EXISTS?

# “For all”

EXERCISES(CAT,ENO, TOPIC, MAXPT)

RESULTS(SID,CAT,ENO, POINTS)

**Who got the best result for homework 1?**

# “For all”

EXERCISES(CAT,ENO,TOPIC,MAXPT)

RESULTS(SID,CAT,ENO,POINTS)

## Who got the best result for homework 1?

```
SELECT FIRST, LAST, POINTS
FROM STUDENTS S, RESULTS X
WHERE S.SID = X.SID
AND X.CAT = 'H' AND X.ENO = '1'
AND NOT EXISTS
  (SELECT *
   FROM RESULTS Y
   WHERE Y.CAT = 'H' AND Y.ENO = 1
   AND Y.POINTS > X.POINTS)
```

# “For all”

EXERCISES(CAT,ENO, TOPIC, MAXPT)

RESULTS(SID,CAT,ENO, POINTS)

## Who got the best result for homework 1?

```
SELECT FIRST, LAST, POINTS
FROM STUDENTS S, RESULTS X
WHERE S.SID = X.SID
AND X.CAT = 'H' AND X.ENO = '1'
AND NOT EXISTS
  (SELECT *
   FROM RESULTS Y
   WHERE Y.CAT = 'H' AND Y.ENO = 1
        AND Y.POINTS > X.POINTS)
```

In natural language:

- **A result x for homework 1 is selected if there is no result y for this exercise with more points than x.**



# “For all”

In mathematical logic there are **quantifiers**:

- $\exists X(\varphi)$       **existential quantifier**
  - Meaning: There is an  $X$  that satisfies formula  $\varphi$ .
- $\forall X(\varphi)$       **universal quantifier**
  - Meaning: For all  $X$ , formula  $\varphi$  is satisfied (true).

# “For all”

In mathematical logic there are **quantifiers**:

- $\exists X(\varphi)$       **existential quantifier**
  - Meaning: There is an  $X$  that satisfies formula  $\varphi$ .
- $\forall X(\varphi)$       **universal quantifier**
  - Meaning: For all  $X$ , formula  $\varphi$  is satisfied (true).

In **tuple relational calculus (TRC)** the maximum number of points for homework 1 reads:

$$\{ X.PPOINTS \mid X \in RESULTS \\ \wedge X.CAT = 'H' \\ \wedge X.ENO = 1 \\ \wedge \forall Y ((Y \in RESULTS \wedge Y.CAT = 'H' \wedge Y.ENO = 1) \\ \Rightarrow Y.PPOINTS \leq X.PPOINTS) \}$$

# “For all”

- SQL does **not** offer a universal quantifier ( $\forall$ , “for all”).

SQL offers only the existential quantifier EXISTS.

*However, see `>= ALL` below.*

# “For all”

- SQL does **not** offer a universal quantifier ( $\forall$ , “for all”).

SQL offers only the existential quantifier EXISTS.

*However, see  $\geq$  ALL below.*

This is no problem because

$$\forall X(\varphi) \iff \neg \exists X(\neg \varphi)$$

# “For all”

- SQL does **not** offer a universal quantifier ( $\forall$ , “for all”).

SQL offers only the existential quantifier EXISTS.

*However, see  $\geq$  ALL below.*

This is no problem because

$$\forall X (\varphi) \iff \neg \exists X (\neg \varphi)$$

The following two statements are equivalent:

- All cars are red.
- There exist no cars that not red.

# “For all”

SQL does not have  $\Rightarrow$ . Thus commonly used pattern

$$\forall X (\varphi_1 \Rightarrow \varphi_2)$$

becomes

$$\neg \exists X (\varphi_1 \wedge \neg \varphi_2)$$

# “For all”

SQL does not have  $\Rightarrow$ . Thus commonly used pattern

$$\forall X (\varphi_1 \Rightarrow \varphi_2)$$

becomes

$$\neg \exists X (\varphi_1 \wedge \neg \varphi_2)$$

The following example:

$$\{ X.PPOINTS \mid X \in \text{RESULTS} \wedge X.CAT = 'H' \wedge X.ENO = 1 \\ \wedge \forall Y ((Y \in \text{RESULTS} \wedge Y.CAT = 'H' \wedge Y.ENO = 1) \\ \Rightarrow Y.PPOINTS \leq X.PPOINTS) \}$$

is thus logically equivalent to:

$$\{ X.PPOINTS \mid X \in \text{RESULTS} \wedge X.CAT = 'H' \wedge X.ENO = 1 \\ \wedge \neg \exists Y ((Y \in \text{RESULTS} \wedge Y.CAT = 'H' \wedge Y.ENO = 1) \\ \wedge Y.PPOINTS > X.PPOINTS) \}$$

# “For all”

$$\{ X.PPOINTS \mid X \in \text{RESULTS} \wedge X.CAT = 'H' \wedge X.ENO = 1 \\ \wedge \neg \exists Y ((Y \in \text{RESULTS} \wedge Y.CAT = 'H' \wedge Y.ENO = 1) \\ \wedge Y.PPOINTS > X.PPOINTS) \}$$

Can be written in SQL as follows:

```
SELECT X.PPOINTS
FROM RESULTS X
WHERE X.CAT = 'H' AND X.ENO = 1
AND NOT EXISTS
  (SELECT *
   FROM RESULTS Y
   WHERE Y.CAT = 'H' AND Y.ENO = 1
   AND Y.PPOINTS > X.PPOINTS)
```



# Nested Subqueries

**Subqueries may be nested!**

# Nested Subqueries

**Subqueries may be nested!**

**List the students who solved all homeworks**

```
SELECT FIRST, LAST
FROM STUDENTS S
WHERE NOT EXISTS
      (SELECT *
       FROM EXERCISES E
       WHERE CAT = 'H'
       AND NOT EXISTS
            (SELECT *
             FROM RESULTS R
             WHERE R.SID = S.SID
             AND R.ENO = E.ENO
             AND R.CAT = 'H'))
```

# Nested Subqueries

**Subqueries may be nested!**

**List the students who solved all homeworks**

```
SELECT FIRST, LAST
FROM STUDENTS S
WHERE NOT EXISTS
      (SELECT *
       FROM EXERCISES E
       WHERE CAT = 'H'
       AND NOT EXISTS
            (SELECT *
             FROM RESULTS R
             WHERE R.SID = S.SID
             AND R.ENO = E.ENO
             AND R.CAT = 'H'))
```

- Inner query: all results for a given student and homework

# Nested Subqueries

**Subqueries may be nested!**

**List the students who solved all homeworks**

```
SELECT FIRST, LAST
FROM STUDENTS S
WHERE NOT EXISTS
      (SELECT *
       FROM EXERCISES E
       WHERE CAT = 'H'
       AND NOT EXISTS
            (SELECT *
             FROM RESULTS R
             WHERE R.SID = S.SID
             AND R.ENO = E.ENO
             AND R.CAT = 'H'))
```

- Inner query: all results for a given student and homework
- Middle query: homeworks of the student without results

# Nested Subqueries

**Subqueries may be nested!**

**List the students who solved all homeworks**

```
SELECT FIRST, LAST
FROM STUDENTS S
WHERE NOT EXISTS
      (SELECT *
       FROM EXERCISES E
       WHERE CAT = 'H'
       AND NOT EXISTS
            (SELECT *
             FROM RESULTS R
             WHERE R.SID = S.SID
             AND R.ENO = E.ENO
             AND R.CAT = 'H'))
```

- Inner query: all results for a given student and homework
- Middle query: homeworks of the student without results
- Outer query: students that have no homework without results

## Does this query compute the student with the best result for homework 1?

```
SELECT  DISTINCT S.FIRST, S.LAST, X.POINTS
FROM    STUDENTS S, RESULTS X, RESULTS Y
WHERE   S.SID = X.SID
AND     X.CAT = 'H' AND X.ENO = 1
AND     Y.CAT = 'H' AND Y.ENO = 1
AND     X.POINTS > Y.POINTS
```

- If not, what does the query compute?

# Common Errors

STUDENTS(SID, FIRST, LAST, EMAIL)

RESULTS(SID, CAT, ENO, POINTS)

## Return those students which did not solve homework 1

```
SELECT  FIRST, LAST
FROM    STUDENTS S
WHERE   NOT EXISTS
        (SELECT *
         FROM  RESULTS R, STUDENTS S
         WHERE R.SID = S.SID
         AND   R.CAT = 'H' AND R.ENO = 1)
```

## Quiz

What goes wrong here?

# Common Errors

```
STUDENTS(SID, FIRST, LAST, EMAIL)
```

```
RESULTS(SID, CAT, ENO, POINTS)
```

## Return those students which did not solve homework 1

```
SELECT  FIRST, LAST
FROM    STUDENTS S
WHERE   NOT EXISTS
        (SELECT *
         FROM  RESULTS R, STUDENTS S
         WHERE R.SID = S.SID
         AND   R.CAT = 'H' AND R.ENO = 1)
```

## Quiz

What goes wrong here?

- Subqueries bring up the concept of **variable scoping** (just like in programming languages) and related pitfalls.



# Common Errors

```
STUDENTS(SID, FIRST, LAST, EMAIL)
EXERCISES(CAT, ENO, TOPIC, MAXPT)
RESULTS(SID→STUDENTS, (CAT,ENO)→EXERCISES, POINTS)
```

## What is the error in this query?

**“Find those students who have neither submitted a homework nor participated in any exam.”**

```
SELECT  FIRST, LAST
FROM    STUDENTS
WHERE   SID NOT IN (SELECT  SID
                   FROM    EXERCISES)
```

1. Is this syntactically correct SQL?
2. What is the output of this query?
3. If the query is faulty, correct it.

# ALL, ANY, SOME

SQL allows to compare a **single value** with all values in a set (computed by a subquery). Such comparisons may be

- **universally** (ALL), or
- **existentially** (ANY)

quantified.

# ALL, ANY, SOME

SQL allows to compare a **single value** with all values in a set (computed by a subquery). Such comparisons may be

- **universally** (ALL), or
- **existentially** (ANY)

quantified.

**Which student(s) got the maximum number of points for homework 1?**

```
SELECT  S.FIRST, S.LAST, X.POINTS
FROM    STUDENTS S, RESULTS X
WHERE   S.SID = X.SID AND X.CAT='H' AND X.ENO=1
AND     X.POINTS >= ALL (SELECT Y.POINTS
                        FROM    RESULTS Y
                        WHERE   Y.CAT = 'H'
                        AND     Y.ENO = 1)
```

# ALL, ANY, SOME

SQL allows to compare a **single value** with all values in a set (computed by a subquery). Such comparisons may be

- **universally** (ALL), or
- **existentially** (ANY)

quantified.

**Which student(s) got the maximum number of points for homework 1?**

```
SELECT  S.FIRST, S.LAST, X.POINTS
FROM    STUDENTS S, RESULTS X
WHERE   S.SID = X.SID AND X.CAT='H' AND X.ENO=1
AND     X.POINTS >= ALL (SELECT Y.POINTS
                        FROM    RESULTS Y
                        WHERE   Y.CAT = 'H'
                        AND     Y.ENO = 1)
```

- Note: usage of >= is important here.

# ALL, ANY, SOME

This query is equivalent to the previous query:

## Using ANY.

```
SELECT  S.FIRST, S.LAST, X.POINTS
FROM    STUDENTS S, RESULTS X
WHERE   S.SID = X.SID AND X.CAT = 'H' AND X.ENO = 1
AND     NOT X.POINTS < ANY (SELECT Y.POINTS
                             FROM   RESULTS Y
                             WHERE  Y.CAT = 'H'
                             AND    Y.ENO = 1)
```

# ALL, ANY, SOME

This query is equivalent to the previous query:

## Using ANY.

```
SELECT  S.FIRST, S.LAST, X.POINTS
FROM    STUDENTS S, RESULTS X
WHERE   S.SID = X.SID AND X.CAT = 'H' AND X.ENO = 1
AND     NOT X.POINTS < ANY (SELECT Y.POINTS
                             FROM   RESULTS Y
                             WHERE  Y.CAT = 'H'
                             AND    Y.ENO = 1)
```

Note that ANY (ALL) do **not** extend SQL's expressiveness.

The ANY statement

$$A < \text{ANY} (\text{SELECT } B \text{ FROM } \dots \text{ WHERE } \dots)$$

is equivalent to the EXISTS statement

$$\text{EXISTS} (\text{SELECT } * \text{ FROM } \dots \text{ WHERE } \dots \text{ AND } A < B)$$

# ALL, ANY, SOME

Syntactical remarks on comparisons with subquery results:

# ALL, ANY, SOME

Syntactical remarks on comparisons with subquery results:

1. ANY and SOME are synonyms.



# ALL, ANY, SOME

Syntactical remarks on comparisons with subquery results:

1. ANY and SOME are synonyms.
2.  $x \text{ IN } S$  is equivalent to  $x = \text{ANY } S$ .

# ALL, ANY, SOME

Syntactical remarks on comparisons with subquery results:

1. ANY and SOME are synonyms.
2.  $x \text{ IN } S$  is equivalent to  $x = \text{ANY } S$ .
3. The subquery must yield a single result column.

# ALL, ANY, SOME

Syntactical remarks on comparisons with subquery results:

1. ANY and SOME are synonyms.
2.  $x \text{ IN } S$  is equivalent to  $x = \text{ANY } S$ .
3. The subquery must yield a single result column.
4. If none of the keywords ALL, ANY, SOME are present, the subquery must **yield at most one row**.
  - With (3), this ensures that the comparison is performed between atomic (non-set) values.
  - An empty subquery result is equivalent to NULL.

# Single Value Subqueries

```
STUDENTS(SID, FIRST, LAST, EMAIL)
EXERCISES(CAT, ENO, TOPIC, MAXPT)
RESULTS(SID→STUDENTS, (CAT,ENO)→EXERCISES, POINTS)
```

## Who got full points for homework 1?

```
SELECT  S.FIRST, S.LAST
FROM    STUDENTS S, RESULTS R
WHERE   S.SID = R.SID AND R.CAT='H' AND R.ENO=1
AND     R.POINTS = (SELECT  MAXPT
                    FROM    EXERCISES
                    WHERE   CAT = 'H' AND ENO = 1)
```

# Single Value Subqueries

```
STUDENTS(SID, FIRST, LAST, EMAIL)
EXERCISES(CAT, ENO, TOPIC, MAXPT)
RESULTS(SID→STUDENTS, (CAT,ENO)→EXERCISES, POINTS)
```

## Who got full points for homework 1?

```
SELECT  S.FIRST, S.LAST
FROM    STUDENTS S, RESULTS R
WHERE   S.SID = R.SID AND R.CAT='H' AND R.ENO=1
AND     R.POINTS = (SELECT  MAXPT
                   FROM    EXERCISES
                   WHERE   CAT = 'H' AND ENO = 1)
```

- **Comparisons with subquery results** (note: no ANY, ALL) are possible if the subquery returns **at most one row**.

# Single Value Subqueries

```
STUDENTS(SID, FIRST, LAST, EMAIL)
EXERCISES(CAT, ENO, TOPIC, MAXPT)
RESULTS(SID→STUDENTS, (CAT,ENO)→EXERCISES, POINTS)
```

## Who got full points for homework 1?

```
SELECT  S.FIRST, S.LAST
FROM    STUDENTS S, RESULTS R
WHERE   S.SID = R.SID AND R.CAT='H' AND R.ENO=1
AND     R.POINTS = (SELECT  MAXPT
                   FROM    EXERCISES
                   WHERE   CAT = 'H' AND ENO = 1)
```

- **Comparisons with subquery results** (note: no ANY, ALL) are possible if the subquery returns **at most one row**.

Why is this guaranteed here?

# Single Value Subqueries

```
STUDENTS(SID, FIRST, LAST, EMAIL)
EXERCISES(CAT, ENO, TOPIC, MAXPT)
RESULTS(SID→STUDENTS, (CAT,ENO)→EXERCISES, POINTS)
```

## Who got full points for homework 1?

```
SELECT  S.FIRST, S.LAST
FROM    STUDENTS S, RESULTS R
WHERE   S.SID = R.SID AND R.CAT='H' AND R.ENO=1
AND     R.POINTS = (SELECT  MAXPT
                   FROM    EXERCISES
                   WHERE   CAT = 'H' AND ENO = 1)
```

- **Comparisons with subquery results** (note: no ANY, ALL) are possible if the subquery returns **at most one row**.

## Why is this guaranteed here?

- Use constraints to ensure this condition.

# Single Value Subqueries

```
STUDENTS(SID, FIRST, LAST, EMAIL)
EXERCISES(CAT, ENO, TOPIC, MAXPT)
RESULTS(SID→STUDENTS, (CAT,ENO)→EXERCISES, POINTS)
```

## Who got full points for homework 1?

```
SELECT  S.FIRST, S.LAST
FROM    STUDENTS S, RESULTS R
WHERE   S.SID = R.SID AND R.CAT='H' AND R.ENO=1
AND     R.POINTS = (SELECT  MAXPT
                   FROM    EXERCISES
                   WHERE   CAT = 'H' AND ENO = 1)
```

- **Comparisons with subquery results** (note: no ANY, ALL) are possible if the subquery returns **at most one row**.

## Why is this guaranteed here?

- Use constraints to ensure this condition.
- The DBMS will yield a runtime error if the subquery returns two or more rows.



# Single Value Subqueries

If the subquery has an **empty result**, the **null value** is returned.

## Bad style!

```
SELECT  FIRST, LAST
FROM    STUDENTS S
WHERE   (SELECT 1
        FROM    RESULTS R
        WHERE   R.SID = S.SID
        AND     R.CAT = 'H' AND R.ENO = 1) IS NULL
```

## Subqueries under FROM

Since the result of an SQL query is a **table**, it seems most natural to use a subquery result wherever a table might be specified, i.e., in the FROM clause.

## Subqueries under FROM

Since the result of an SQL query is a **table**, it seems most natural to use a subquery result wherever a table might be specified, i.e., in the FROM clause.

In the following example, the join of RESULTS and EXERCISES is computed in a subquery.

### Points (in %) achieved in homework exercise 1.

```
SELECT  X.SID, (X.POINTS * 100 / X.MAXPT) AS PCT
FROM    (SELECT  E.CAT, E.ENO, R.SID,
              R.POINTS, E.MAXPT
        FROM    EXERCISES E, RESULTS R
        WHERE   E.CAT = R.CAT AND E.ENO = R.ENO) X
WHERE   X.CAT = 'H' AND X.ENO = 1
```

## Subqueries under FROM

Since the result of an SQL query is a **table**, it seems most natural to use a subquery result wherever a table might be specified, i.e., in the FROM clause.

In the following example, the join of RESULTS and EXERCISES is computed in a subquery.

### Points (in %) achieved in homework exercise 1.

```
SELECT  X.SID, (X.POINTS * 100 / X.MAXPT) AS PCT
FROM    (SELECT  E.CAT, E.ENO, R.SID,
              R.POINTS, E.MAXPT
        FROM    EXERCISES E, RESULTS R
        WHERE   E.CAT = R.CAT AND E.ENO = R.ENO) X
WHERE   X.CAT = 'H' AND X.ENO = 1
```

One use of subqueries under FROM are **nested aggregations**.

## Subqueries under FROM

Inside the subquery, tuple variables introduced in the same FROM clause **may not be referenced**.



# Subqueries under FROM

Inside the subquery, tuple variables introduced in the same FROM clause **may not be referenced**.



## Not allowed in SQL!

```
SELECT  S.FIRST, S.LAST, X.ENO, X.POINTS
FROM    STUDENTS S, (SELECT  R.ENO, R.POINTS
                     FROM    RESULTS R
                     WHERE    R.CAT = 'H'
                     AND      R.SID = S.SID) X
```

## Subqueries under FROM

A **view declaration** registers a **query (not a query result)** under a given identifier in the database.

# Subqueries under FROM

A **view declaration** registers a **query (not a query result)** under a given identifier in the database.

## View: homework points

```
CREATE VIEW HW_POINTS AS
SELECT  S.FIRST, S.LAST, R.ENO, R.POINTS
FROM    STUDENTS S, RESULTS R
WHERE   S.SID = R.SID AND R.CAT = 'H'
```



# Subqueries under FROM

A **view declaration** registers a **query (not a query result)** under a given identifier in the database.

## View: homework points

```
CREATE VIEW HW_POINTS AS
SELECT  S.FIRST, S.LAST, R.ENO, R.POINTS
FROM    STUDENTS S, RESULTS R
WHERE   S.SID = R.SID AND R.CAT = 'H'
```

In queries, views may be used like stored tables:

```
SELECT  ENO, POINTS
FROM    HW_POINTS
WHERE   FIRST = 'Michael' AND LAST = 'Jones'
```

# Subqueries under FROM

A **view declaration** registers a **query (not a query result)** under a given identifier in the database.

## View: homework points

```
CREATE VIEW HW_POINTS AS
SELECT  S.FIRST, S.LAST, R.ENO, R.POINTS
FROM    STUDENTS S, RESULTS R
WHERE   S.SID = R.SID AND R.CAT = 'H'
```

In queries, views may be used like stored tables:

```
SELECT  ENO, POINTS
FROM    HW_POINTS
WHERE   FIRST = 'Michael' AND LAST = 'Jones'
```

- Views may be thought of as **subquery macros**

## Overview

1. SELECT-FROM-WHERE Blocks, Tuple Variables
2. Subqueries, Non-Monotonic Constructs
3. Aggregations I: Aggregation Functions
4. Aggregations II: GROUP BY, HAVING
5. UNION, Conditional Expressions
6. ORDER BY
7. SQL-92 Joins, Outer Join

# Aggregations

**Aggregation functions** are functions from a set or multiset to a single value, e.g.,

$$\min\{42, 57, 5, 13, 27\} = 5$$

# Aggregations

**Aggregation functions** are functions from a set or multiset to a single value, e.g.,

$$\min\{42, 57, 5, 13, 27\} = 5$$

- Used to summarize an entire set of values.

# Aggregations

**Aggregation functions** are functions from a set or multiset to a single value, e.g.,

$$\min\{42, 57, 5, 13, 27\} = 5$$

- Used to summarize an entire set of values.
- Aggregation functions are also known as
  - **group functions**, or
  - **column functions**

# Aggregations

**Aggregation functions** are functions from a set or multiset to a single value, e.g.,

$$\min\{42, 57, 5, 13, 27\} = 5$$

- Used to summarize an entire set of values.
- Aggregation functions are also known as
  - **group functions**, or
  - **column functions**
- The values of an entire column (or partitions of these values) form the input to such functions.

# Aggregations

**Aggregation functions** are functions from a set or multiset to a single value, e.g.,

$$\min\{42, 57, 5, 13, 27\} = 5$$

- Used to summarize an entire set of values.
- Aggregation functions are also known as
  - **group functions**, or
  - **column functions**
- The values of an entire column (or partitions of these values) form the input to such functions.

Typical use: statistics, data analysis, report generation.



# Aggregations

SQL-92 defines the five main aggregation functions

COUNT, SUM, AVG, MAX, MIN

# Aggregations

SQL-92 defines the five main aggregation functions

COUNT, SUM, AVG, MAX, MIN

**How many students in the current database state?**

```
SELECT COUNT(*)  
FROM STUDENTS
```

COUNT(*)
----------

4
---

# Aggregations

SQL-92 defines the five main aggregation functions

COUNT, SUM, AVG, MAX, MIN

**How many students in the current database state?**

```
SELECT COUNT(*)  
FROM STUDENTS
```

COUNT(*)
----------

4
---

- Some DBMS define further aggregation functions:

CORRELATION, STDDEV, VARIANCE, ...

# Aggregations

SQL-92 defines the five main aggregation functions

COUNT, SUM, AVG, MAX, MIN

**How many students in the current database state?**

```
SELECT COUNT(*)  
FROM STUDENTS
```

COUNT(\*)

4

- Some DBMS define further aggregation functions:

CORRELATION, STDDEV, VARIANCE, ...

- Some aggregation functions are sensitive to **duplicates**:

SUM, COUNT, AVG ,

some are insensitive:

MIN, MAX

# Aggregations

SQL-92 defines the five main aggregation functions

COUNT, SUM, AVG, MAX, MIN

## How many students in the current database state?

```
SELECT COUNT(*)  
FROM STUDENTS
```

COUNT(\*)

4

- Some DBMS define further aggregation functions:

CORRELATION, STDDEV, VARIANCE, ...

- Some aggregation functions are sensitive to **duplicates**:

SUM, COUNT, AVG ,

some are insensitive:

MIN, MAX

- SQL allows to explicitly request to ignore duplicates, e.g.:

... COUNT(DISTINCT A) ...

# Simple Aggregations

**Simple aggregations** feed the value set of an **entire column** into an aggregation function.

*Below, we will discuss partitioning (or **grouping**) of columns.*

# Simple Aggregations

**Simple aggregations** feed the value set of an **entire column** into an aggregation function.

*Below, we will discuss partitioning (or **grouping**) of columns.*

**How many students in the current database state?**

```
SELECT COUNT(*)  
FROM STUDENTS
```

COUNT(*)
4

# Simple Aggregations

**Simple aggregations** feed the value set of an **entire column** into an aggregation function.

*Below, we will discuss partitioning (or **grouping**) of columns.*

**How many students in the current database state?**

```
SELECT COUNT(*)  
FROM STUDENTS
```

COUNT(*)
4

**Best and average result for homework 1?**

```
SELECT MAX(POINTS), AVG(POINTS)  
FROM RESULTS  
WHERE CAT = 'H' AND ENO = 1
```

MAX(POINTS)	AVG(POINTS)
10	8



# Simple Aggregations

STUDENTS(SID, FIRST, LAST, EMAIL)

EXERCISES(CAT, ENO, TOPIC, MAXPT)

RESULTS(SID→STUDENTS, (CAT,ENO)→EXERCISES, POINTS)

**How many students have submitted a homework?**

# Simple Aggregations

```
STUDENTS(SID, FIRST, LAST, EMAIL)
EXERCISES(CAT, ENO, TOPIC, MAXPT)
RESULTS(SID→STUDENTS, (CAT,ENO)→EXERCISES, POINTS)
```

## How many students have submitted a homework?

```
SELECT COUNT(DISTINCT SID)
FROM RESULTS
WHERE CAT = 'H'
```

COUNT(DISTINCT SID)
3

# Simple Aggregations

```
STUDENTS(SID, FIRST, LAST, EMAIL)
EXERCISES(CAT, ENO, TOPIC, MAXPT)
RESULTS(SID→STUDENTS, (CAT,ENO)→EXERCISES, POINTS)
```

**How many students have submitted a homework?**

```
SELECT COUNT(DISTINCT SID)
FROM RESULTS
WHERE CAT = 'H'
```

COUNT(DISTINCT SID)
3

**What is the total number of points student 101 got for her homeworks?**

# Simple Aggregations

```
STUDENTS(SID, FIRST, LAST, EMAIL)
EXERCISES(CAT, ENO, TOPIC, MAXPT)
RESULTS(SID→STUDENTS, (CAT,ENO)→EXERCISES, POINTS)
```

## How many students have submitted a homework?

```
SELECT COUNT(DISTINCT SID)
FROM RESULTS
WHERE CAT = 'H'
```

COUNT(DISTINCT SID)
3

## What is the total number of points student 101 got for her homeworks?

```
SELECT SUM(POINTS) AS "Total Points"
FROM RESULTS
WHERE SID = 101 AND CAT = 'H'
```

Total Points
18

# Simple Aggregations

**What average percentage of the maximum points did the students reach for homework 1?**

```
SELECT  AVG(R.POINTS / E.MAXPT) * 100
FROM    RESULTS R, EXERCISES E
WHERE   R.CAT = 'H' AND E.CAT = 'H'
AND     R.ENO = 1 AND E.ENO = 1
```

# Simple Aggregations

**What average percentage of the maximum points did the students reach for homework 1?**

```
SELECT  AVG(R.POINTS / E.MAXPT) * 100
FROM    RESULTS R, EXERCISES E
WHERE   R.CAT = 'H' AND E.CAT = 'H'
AND     R.ENO = 1 AND E.ENO = 1
```

**Homework points for student 101 plus 3 bonus points.**

```
SELECT  SUM(POINTS) + 3 AS "Total Homework Points"
FROM    RESULTS
WHERE   SID = 101 AND CAT = 'H'
```

# Restrictions

- Aggregations may not be nested (makes no sense).

# Restrictions

- Aggregations may not be nested (makes no sense).
- Aggregations may not be used in the WHERE clause:

**Wrong!**

```
... WHERE SUM(A) > 100 ...
```



# Restrictions

- Aggregations may not be nested (makes no sense).
- Aggregations may not be used in the WHERE clause:

## Wrong!

```
... WHERE SUM(A) > 100 ...
```

- If an aggregation function is used and no GROUP BY is used (simple aggregation), **no attributes** may appear in the SELECT clause.

## Wrong!

```
SELECT  CAT, ENO, AVG(POINTS)
FROM    RESULTS
```

# Null Values and Aggregations

Usually, null values are **ignored** (filtered out) before the aggregation operator is applied.

# Null Values and Aggregations

Usually, null values are **ignored** (filtered out) before the aggregation operator is applied.

Exception:

- COUNT(\*) counts null values
- COUNT(\*) counts rows, not attribute values

# Null Values and Aggregations

Usually, null values are **ignored** (filtered out) before the aggregation operator is applied.

Exception:

- COUNT(\*) counts null values
- COUNT(\*) counts rows, not attribute values

If the input set is empty, aggregation functions yield NULL.

- Exception: COUNT returns 0.

# Null Values and Aggregations

Usually, null values are **ignored** (filtered out) before the aggregation operator is applied.

Exception:

- COUNT(\*) counts null values
- COUNT(\*) counts rows, not attribute values

If the input set is empty, aggregation functions yield NULL.

- Exception: COUNT returns 0.
- This seems counter-intuitive, at least for SUM (where users might expect 0 in this case)

# Null Values and Aggregations

Usually, null values are **ignored** (filtered out) before the aggregation operator is applied.

Exception:

- COUNT(\*) counts null values
- COUNT(\*) counts rows, not attribute values

If the input set is empty, aggregation functions yield NULL.

- Exception: COUNT returns 0.
- This seems counter-intuitive, at least for SUM (where users might expect 0 in this case)
- However, allows to detect the difference between:
  - all column values NULL, or
  - values that sum up to 0.

## Overview

1. SELECT-FROM-WHERE Blocks, Tuple Variables
2. Subqueries, Non-Monotonic Constructs
3. Aggregations I: Aggregation Functions
4. Aggregations II: GROUP BY, HAVING
5. UNION, Conditional Expressions
6. ORDER BY
7. SQL-92 Joins, Outer Join

# GROUP BY

SQL's GROUP BY construct **partitions** the tuples of a table into **disjoint groups**.

- Aggregation functions may then be applied for each tuple group separately.



# GROUP BY

SQL's GROUP BY construct **partitions** the tuples of a table into **disjoint groups**.

- Aggregation functions may then be applied for each tuple group separately.

## Average points for each homework

```
SELECT ENO, AVG(POINTS)
FROM RESULTS
WHERE CAT = 'H'
GROUP BY ENO
```

ENO	AVG(POINTS)
1	8
2	8.5

All tuples agreeing in their ENO values (i.e., belonging to the same homework) form a group for aggregation.

# GROUP BY

The GROUP BY partitions the incoming tuples into groups:

- based on **value equality for the GROUP BY attributes**

(after evaluation of the FROM and WHERE clauses)

# GROUP BY

The GROUP BY partitions the incoming tuples into groups:

- based on **value equality for the GROUP BY attributes**

(after evaluation of the FROM and WHERE clauses)

```
SELECT  ENO, AVG(POINTS)
FROM    RESULTS
WHERE   CAT = 'H'
GROUP BY ENO
```

ENO-based groups formed by above example query:

SID	CAT	ENO	POINTS
101	H	1	10
102	H	1	9
103	H	1	5
101	H	2	8
101	H	2	9

# GROUP BY

The GROUP BY partitions the incoming tuples into groups:

- based on **value equality for the GROUP BY attributes**

(after evaluation of the FROM and WHERE clauses)

```
SELECT ENO, AVG(POINTS)
FROM RESULTS
WHERE CAT = 'H'
GROUP BY ENO
```

ENO-based groups formed by above example query:

SID	CAT	ENO	POINTS
101	H	1	10
102	H	1	9
103	H	1	5
101	H	2	8
101	H	2	9

Aggregation is subsequently done for every group (yielding as many rows as groups).

# GROUP BY

The GROUP BY construction can **never** produce empty groups.

- COUNT(\*) will never result in 0

Since the GROUP BY attributes have a **unique value for every group**, these attributes may be used in the SELECT clause.

- A reference to any other attribute is illegal.

## Wrong!

```
SELECT    E.ENO, E.TOPIC, AVG(R.POINTS)
FROM      EXERCISES E, RESULTS R
WHERE     E.CAT = 'H'
          AND R.CAT = 'H'
          AND E.ENO = R.ENO
GROUP BY  E.ENO
```

Wrong! Although E.ENO functionally determines E.TOPIC which thus is unique (for every group).

# GROUP BY

Grouping by E.ENO **and** E.TOPIC yields the desired result:

```
SELECT    E.ENO, E.TOPIC, AVG(R.POINTS)
FROM      EXERCISES E, RESULTS R
WHERE     E.CAT = 'H'
          AND R.CAT = 'H'
          AND E.ENO = R.ENO
GROUP BY  E.ENO, E.TOPIC
```

E.ENO	E.TOPIC	AVG(POINTS)
1	Rel.Alg.	8
2	SQL	8.5

Now the DBMS has a **syntactic clue** that E.TOPIC is unique.

## Is there any semantical difference between these queries?

1. SELECT TOPIC, AVG(POINTS / MAXPT)  
FROM EXERCISES E, RESULTS R  
WHERE E.CAT='H' AND R.CAT='H' AND E.ENO=R.ENO  
GROUP BY TOPIC
2. SELECT TOPIC, AVG(POINTS / MAXPT)  
FROM EXERCISES E, RESULTS R  
WHERE E.CAT='H' AND R.CAT='H' AND E.ENO=R.ENO  
GROUP BY TOPIC, E.ENO

# GROUP BY

- The sequence of the GROUP BY attributes is not important.



# GROUP BY

- The sequence of the GROUP BY attributes is not important.
- Duplicates should be eliminated with DISTINCT, although such elimination can also be realised via GROUP BY:

## Grouping without aggregation: DISTINCT.

```
SELECT    CAT, ENO
FROM      RESULTS
GROUP BY  CAT, ENO
```

This is an **abuse** of GROUP BY and should be avoided.

# HAVING

Aggregations may not be used in the WHERE clause.

# HAVING

Aggregations may not be used in the WHERE clause.

With GROUP BY, however, it may make sense to **filter out entire groups** based on some aggregated group property.

# HAVING

Aggregations may not be used in the WHERE clause.

With GROUP BY, however, it may make sense to **filter out entire groups** based on some aggregated group property.

For example, only groups of size greater than  $n$  tuples.

```
SELECT    ...           -- output columns
FROM      ...           -- what tuples
WHERE     ...           -- filter tuples
GROUP BY  ...           -- group tuples
HAVING    COUNT(*) > n  -- filter groups
```

# HAVING

Aggregations may not be used in the WHERE clause.

With GROUP BY, however, it may make sense to **filter out entire groups** based on some aggregated group property.

For example, only groups of size greater than  $n$  tuples.

```
SELECT    ...           -- output columns
FROM      ...           -- what tuples
WHERE     ...           -- filter tuples
GROUP BY  ...           -- group tuples
HAVING    COUNT(*) > n  -- filter groups
```

This is possible with SQL's HAVING clause.

- The condition in the HAVING clause may (only) involve aggregation functions.

# HAVING

STUDENTS(SID, FIRST, LAST, EMAIL)

EXERCISES(CAT, ENO, TOPIC, MAXPT)

RESULTS(SID→STUDENTS, (CAT,ENO)→EXERCISES, POINTS)

Which students got at least 18 homework points?

# HAVING

```
STUDENTS(SID, FIRST, LAST, EMAIL)
EXERCISES(CAT, ENO, TOPIC, MAXPT)
RESULTS(SID→STUDENTS, (CAT,ENO)→EXERCISES, POINTS)
```

## Which students got at least 18 homework points?

```
SELECT    FIRST, LAST
FROM      STUDENTS S, RESULTS R
WHERE     S.SID = R.SID AND R.CAT = 'H'
GROUP BY  S.SID, FIRST, LAST
HAVING    SUM(POINTS) >= 18
```

FIRST	LAST
Ann	Smith
Michael	Jones

# HAVING

```
STUDENTS(SID, FIRST, LAST, EMAIL)
EXERCISES(CAT, ENO, TOPIC, MAXPT)
RESULTS(SID→STUDENTS, (CAT,ENO)→EXERCISES, POINTS)
```

## Which students got at least 18 homework points?

```
SELECT    FIRST, LAST
FROM      STUDENTS S, RESULTS R
WHERE     S.SID = R.SID AND R.CAT = 'H'
GROUP BY  S.SID, FIRST, LAST
HAVING    SUM(POINTS) >= 18
```

FIRST	LAST
Ann	Smith
Michael	Jones

The WHERE clause refers to single tuples, the HAVING condition applies to entire groups (in this case: all tuples containing the homework results of a student).



# WHERE vs. HAVING

GROUP BY should not contain direct attribute references, only aggregation functions.

## This is wrong

```
SELECT    FIRST, LAST
FROM      STUDENTS S, RESULTS R
GROUP BY  S.SID, R.SID, FIRST, LAST
HAVING    S.SID = R.SID AND SUM(POINTS) >= 18
```

## This is correct

```
SELECT    FIRST, LAST
FROM      STUDENTS S, RESULTS R
WHERE     S.SID = R.SID
GROUP BY  S.SID, FIRST, LAST
HAVING    SUM(POINTS) >= 18
```

# Aggregation Subqueries

## Who has the best result for homework 1?

```
SELECT  S.FIRST, S.LAST, R.POINTS
FROM    STUDENTS S, RESULTS R
WHERE   S.SID = R.SID AND R.CAT = 'H' AND R.ENO = 1
AND     R.POINTS = (SELECT  MAX(POINTS)
                   FROM    RESULTS
                   WHERE   CAT = 'H' AND ENO = 1)
```

## Who has the best result for homework 1?

```
SELECT  S.FIRST, S.LAST, R.POINTS
FROM    STUDENTS S, RESULTS R
WHERE   S.SID = R.SID AND R.CAT = 'H' AND R.ENO = 1
AND     R.POINTS = (SELECT  MAX(POINTS)
                   FROM    RESULTS
                   WHERE   CAT = 'H' AND ENO = 1)
```

- The aggregate in the subquery is guaranteed to yield exactly one row as required.

# Aggregation Subqueries

## Who has the best result for homework 1?

```
SELECT  S.FIRST, S.LAST, R.POINTS
FROM    STUDENTS S, RESULTS R
WHERE   S.SID = R.SID AND R.CAT = 'H' AND R.ENO = 1
AND     R.POINTS = (SELECT  MAX(POINTS)
                   FROM    RESULTS
                   WHERE   CAT = 'H' AND ENO = 1)
```

- The aggregate in the subquery is guaranteed to yield exactly one row as required.
- Remember: earlier we solved this using ANY/ALL.



# Nested Aggregations

**Nested aggregations** require a subquery in the FROM clause.

# Nested Aggregations

**Nested aggregations** require a subquery in the FROM clause.

What is the average number of homework points (excluding those students who did not submit anything)?

```
SELECT  AVG(X.HW_POINTS)
FROM    (SELECT  SID, SUM(POINTS) AS HW_POINTS
        FROM    RESULTS
        WHERE   CAT = 'H'
        GROUP BY SID) X
```

X	
SID	HW_POINTS
101	18
102	18
103	5

AVG(X.HW_POINTS)
13.67

# Maximizing Aggregations

**Who has the best overall homework result?  
(maximum sum of homework points)**



# Maximizing Aggregations

## Who has the best overall homework result? (maximum sum of homework points)

```
SELECT  FIRST, LAST, SUM(POINTS) AS TOTAL
FROM    STUDENTS S, RESULTS R
WHERE   S.SID = R.SID AND R.CAT = 'H'
GROUP BY S.SID, FIRST, LAST
HAVING  SUM(POINTS)
        >= ALL (SELECT  SUM (POINTS)
                FROM    RESULTS
                WHERE   CAT = 'H'
                GROUP BY SID)
```

# Maximizing Aggregations

## Who has the best overall homework result? (maximum sum of homework points)

```
SELECT    FIRST, LAST, SUM(POINTS) AS TOTAL
FROM      STUDENTS S, RESULTS R
WHERE     S.SID = R.SID AND R.CAT = 'H'
GROUP BY  S.SID, FIRST, LAST
HAVING    SUM(POINTS)
          >= ALL (SELECT    SUM (POINTS)
                  FROM      RESULTS
                  WHERE     CAT = 'H'
                  GROUP BY  SID)
```

- Alternatively, we could use a view to solve this problem (next slide).

# Maximizing Aggregations

**View: total number of homework points for each student.**

```
CREATE VIEW HW_TOTALS AS
SELECT  SID, SUM(POINTS) AS TOTAL
FROM    RESULTS
WHERE   CAT = 'H'
GROUP BY SID
```

# Maximizing Aggregations

**View: total number of homework points for each student.**

```
CREATE VIEW HW_TOTALS AS
SELECT  SID, SUM(POINTS) AS TOTAL
FROM    RESULTS
WHERE   CAT = 'H'
GROUP BY SID
```

**Alternative formulation of query on previous slide.**

# Maximizing Aggregations

**View: total number of homework points for each student.**

```
CREATE VIEW HW_TOTALS AS
SELECT    SID, SUM(POINTS) AS TOTAL
FROM      RESULTS
WHERE     CAT = 'H'
GROUP BY  SID
```

**Alternative formulation of query on previous slide.**

```
SELECT  S.FIRST, S.LAST, H.TOTAL
FROM    STUDENTS S, HW_TOTALS H
WHERE   S.SID = H.SID
AND     H.TOTAL = (SELECT MAX(TOTAL)
                  FROM    HW_TOTALS)
```

## Overview

1. SELECT-FROM-WHERE Blocks, Tuple Variables
2. Subqueries, Non-Monotonic Constructs
3. Aggregations I: Aggregation Functions
4. Aggregations II: GROUP BY, HAVING
5. UNION, Conditional Expressions
6. ORDER BY
7. SQL-92 Joins, Outer Join

# UNION

UNION allows to combine the results of two queries.

# UNION

UNION allows to combine the results of two queries.

- UNION is strictly needed since there is no other method to construct one result column that draws from different tables/columns.



# UNION

UNION allows to combine the results of two queries.

- UNION is strictly needed since there is no other method to construct one result column that draws from different tables/columns.
- This is necessary, for example, if specialisations of a concept (“subclasses”) are stored in separate tables.

For instance, if we have tables

- GRADUATE\_COURSES and
- UNDERGRADUATE\_COURSES

both of which are specialisations of the concept COURSE.

# UNION

UNION allows to combine the results of two queries.

- UNION is strictly needed since there is no other method to construct one result column that draws from different tables/columns.
- This is necessary, for example, if specialisations of a concept (“subclasses”) are stored in separate tables.

For instance, if we have tables

- GRADUATE\_COURSES and
- UNDERGRADUATE\_COURSES

both of which are specialisations of the concept COURSE.

- UNION is also commonly used for **case analysis** (cf., the if...then... cascades in programming languages).

**Total number of homework points for every students  
(or 0 if no homework submitted).**

```
SELECT    S.FIRST, S.LAST, SUM(R.POINTS) AS TOTAL
FROM      STUDENTS S, RESULTS R
WHERE     S.SID = R.SID AND R.CAT = 'H'
GROUP BY  S.SID, S.FIRST, S.LAST
UNION ALL
SELECT    S.FIRST, S.LAST, 0 AS TOTAL
FROM      STUDENTS S
WHERE     S.SID NOT IN (SELECT  SID
                        FROM    RESULTS
                        WHERE    CAT = 'H')
```

## Assign student grades based on homework 1.

```
SELECT  S.SID, S.FIRST, S.LAST, 'A' AS GRADE
FROM    STUDENTS S, RESULTS R
WHERE   S.SID = R.SID AND R.CAT = 'H' AND R.ENO=1
AND     R.POINTS >= 9
UNION ALL
SELECT  S.SID, S.FIRST, S.LAST, 'B' AS GRADE
FROM    STUDENTS S, RESULTS R
WHERE   S.SID = R.SID AND R.CAT = 'H' AND R.ENO=1
AND     R.POINTS >= 7 AND R.POINTS < 9
UNION ALL
...
```

# UNION

The UNION operand subqueries must return tables with the same number of columns and compatible data types.

# UNION

The UNION operand subqueries must return tables with the same number of columns and compatible data types.

*Columns correspondence is by column **position** (1st, 2nd, . . .). Column names need not be identical*

# UNION

The UNION operand subqueries must return tables with the same number of columns and compatible data types.

*Columns correspondence is by column **position** (1st, 2nd, ...). Column names need not be identical*

SQL distinguishes between

- UNION: with **duplicate elimination**, and
- UNION ALL: **concatenation** (duplicates retained).

# UNION

The UNION operand subqueries must return tables with the same number of columns and compatible data types.

*Columns correspondence is by column **position** (1st, 2nd, ...). Column names need not be identical*

SQL distinguishes between

- UNION: with **duplicate elimination**, and
- UNION ALL: **concatenation** (duplicates retained).

Other SQL-92 set operations:

- EXCEPT ( $-$ )
- INTERSECT ( $\cap$ )



# UNION

The UNION operand subqueries must return tables with the same number of columns and compatible data types.

*Columns correspondence is by column **position** (1st, 2nd, ...). Column names need not be identical*

SQL distinguishes between

- UNION: with **duplicate elimination**, and
- UNION ALL: **concatenation** (duplicates retained).

Other SQL-92 set operations:

- EXCEPT ( $-$ )
- INTERSECT ( $\cap$ )

These do **not** add to the expressivity of SQL.

# UNION

The UNION operand subqueries must return tables with the same number of columns and compatible data types.

*Columns correspondence is by column **position** (1st, 2nd, ...). Column names need not be identical*

SQL distinguishes between

- UNION: with **duplicate elimination**, and
- UNION ALL: **concatenation** (duplicates retained).

Other SQL-92 set operations:

- EXCEPT ( $-$ )
- INTERSECT ( $\cap$ )

These do **not** add to the expressivity of SQL.

How?

# Conditional Expressions

Whereas UNION is the **portable way** to conduct a case analysis, sometimes a **conditional expression** suffices and is more efficient.

*Conditional expression syntax varies between DBMSs. Oracle uses `DECODE( . . . )`, for example.*

# Conditional Expressions

Whereas UNION is the **portable way** to conduct a case analysis, sometimes a **conditional expression** suffices and is more efficient.

*Conditional expression syntax varies between DBMSs. Oracle uses `DECODE( . . . )`, for example.*

Here, we will use the SQL-92 syntax.

# Conditional Expressions

Whereas UNION is the **portable way** to conduct a case analysis, sometimes a **conditional expression** suffices and is more efficient.

*Conditional expression syntax varies between DBMSs. Oracle uses DECODE(···), for example.*

Here, we will use the SQL-92 syntax.

**Full exercise category name for the results of Ann Smith.**

```
SELECT  CASE WHEN CAT = 'H' THEN 'Homework'
          WHEN CAT = 'M' THEN 'Midterm Exam'
          WHEN CAT = 'F' THEN 'Final Exam'
          ELSE 'Unknown Category' END,
        ENO, POINTS
FROM    STUDENTS S, RESULTS R
WHERE   S.SID = R.SID
AND     S.FIRST = 'Ann' AND S.LAST = 'Smith'
```

# Conditional Expressions

A typical application is to **replace a null value** by a value  $Y$ :

```
... CASE WHEN  $X$  IS NOT NULL THEN  $X$  ELSE  $Y$  END ...
```

# Conditional Expressions

A typical application is to **replace a null value** by a value  $Y$ :

```
... CASE WHEN  $X$  IS NOT NULL THEN  $X$  ELSE  $Y$  END ...
```

In SQL-92, this may be abbreviated to

```
... COALESCE ( $X$ ,  $Y$ )...
```

# Conditional Expressions

A typical application is to **replace a null value** by a value  $Y$ :

```
... CASE WHEN  $X$  IS NOT NULL THEN  $X$  ELSE  $Y$  END ...
```

In SQL-92, this may be abbreviated to

```
... COALESCE ( $X$ ,  $Y$ )...
```

## List the e-mail addresses of all students

```
SELECT FIRST, LAST, COALESCE (EMAIL, '(none)')  
FROM STUDENTS
```



## Overview

1. SELECT-FROM-WHERE Blocks, Tuple Variables
2. Subqueries, Non-Monotonic Constructs
3. Aggregations I: Aggregation Functions
4. Aggregations II: GROUP BY, HAVING
5. UNION, Conditional Expressions
6. ORDER BY
7. SQL-92 Joins, Outer Join

# Sorting Output

If query output is to be read by humans, enforcing a certain **tuple order** greatly helps in interpreting the result.

ORDER BY allows to specify a **list of sorting criteria**.

# Sorting Output

If query output is to be read by humans, enforcing a certain **tuple order** greatly helps in interpreting the result.

ORDER BY allows to specify a **list of sorting criteria**.

Without such an ordering, the order is **unpredictable**:

- Depends on the internal algorithms of the query optimiser.
- Order may change even query to query.

# Sorting Output

An ORDER BY clause may specify multiple attribute names.

- The second attribute is used for tuple ordering if they agree on the first attribute, and so on (**lexicographic ordering**).

# Sorting Output

An ORDER BY clause may specify multiple attribute names.

- The second attribute is used for tuple ordering if they agree on the first attribute, and so on (**lexicographic ordering**).
- Sort in **ascending** order (default): ASC,
- Sort in **descending** order: DESC.

# Sorting Output

An ORDER BY clause may specify multiple attribute names.

- The second attribute is used for tuple ordering if they agree on the first attribute, and so on (**lexicographic ordering**).
- Sort in **ascending** order (default): ASC,
- Sort in **descending** order: DESC.

**Homework results sorted by exercise (best result first).  
In case of a tie, sort alphabetically by student name.**

```
SELECT  R.ENO, R.POINTS, S.FIRST, S.LAST
FROM    STUDENTS S, RESULTS R
WHERE   S.SID = R.SID AND R.CAT = 'H'
ORDER BY R.ENO, R.POINTS DESC, S.LAST, S.FIRST
```

# Sorting Output

An ORDER BY clause may specify multiple attribute names.

- The second attribute is used for tuple ordering if they agree on the first attribute, and so on (**lexicographic ordering**).
- Sort in **ascending** order (default): ASC,
- Sort in **descending** order: DESC.

**Homework results sorted by exercise (best result first).  
In case of a tie, sort alphabetically by student name.**

```
SELECT    R.ENO, R.POINTS, S.FIRST, S.LAST
FROM      STUDENTS S, RESULTS R
WHERE     S.SID = R.SID AND R.CAT = 'H'
ORDER BY  R.ENO, R.POINTS DESC, S.LAST, S.FIRST
```

- First, compare R.ENO.

# Sorting Output

An ORDER BY clause may specify multiple attribute names.

- The second attribute is used for tuple ordering if they agree on the first attribute, and so on (**lexicographic ordering**).
- Sort in **ascending** order (default): ASC,
- Sort in **descending** order: DESC.

**Homework results sorted by exercise (best result first).  
In case of a tie, sort alphabetically by student name.**

```
SELECT  R.ENO, R.POINTS, S.FIRST, S.LAST
FROM    STUDENTS S, RESULTS R
WHERE   S.SID = R.SID AND R.CAT = 'H'
ORDER BY R.ENO, R.POINTS DESC, S.LAST, S.FIRST
```

- First, compare R.ENO.
- If the first criterion leads to a tie, compare POINTS **DESC**.



# Sorting Output

An ORDER BY clause may specify multiple attribute names.

- The second attribute is used for tuple ordering if they agree on the first attribute, and so on (**lexicographic ordering**).
- Sort in **ascending** order (default): ASC,
- Sort in **descending** order: DESC.

**Homework results sorted by exercise (best result first).  
In case of a tie, sort alphabetically by student name.**

```
SELECT  R.ENO, R.POINTS, S.FIRST, S.LAST
FROM    STUDENTS S, RESULTS R
WHERE   S.SID = R.SID AND R.CAT = 'H'
ORDER BY R.ENO, R.POINTS DESC, S.LAST, S.FIRST
```

- First, compare R.ENO.
- If the first criterion leads to a tie, compare POINTS **DESC**.
- If we still have a tie, compare S.LAST.

# Sorting Output

An ORDER BY clause may specify multiple attribute names.

- The second attribute is used for tuple ordering if they agree on the first attribute, and so on (**lexicographic ordering**).
- Sort in **ascending** order (default): ASC,
- Sort in **descending** order: DESC.

**Homework results sorted by exercise (best result first).  
In case of a tie, sort alphabetically by student name.**

```
SELECT    R.ENO, R.POINTS, S.FIRST, S.LAST
FROM      STUDENTS S, RESULTS R
WHERE     S.SID = R.SID AND R.CAT = 'H'
ORDER BY  R.ENO, R.POINTS DESC, S.LAST, S.FIRST
```

- First, compare R.ENO.
- If the first criterion leads to a tie, compare POINTS **DESC**.
- If we still have a tie, compare S.LAST.
- If we still have a tie, compare S.FIRST.

# Sorting Output

Result of last example query.

ENO	POINTS	FIRST	LAST
1	10	Ann	Smith
1	9	Michael	Jones
1	5	Richard	Turner
2	9	Michael	Jones
2	8	Ann	Smith

## Sorting Output

- In some application scenarios it is necessary to **add columns** to a table to obtain suitable **sorting criteria**.

## Sorting Output

- In some application scenarios it is necessary to **add columns** to a table to obtain suitable **sorting criteria**.

If the students names were stored in the form 'Ann\_Smith', sorting by last name is more or less impossible. Having separate columns for first and last name is better.

## Sorting Output

- In some application scenarios it is necessary to **add columns** to a table to obtain suitable **sorting criteria**.

If the students names were stored in the form 'Ann\_Smith', sorting by last name is more or less impossible. Having separate columns for first and last name is better.

- **Null values** are all listed first or all listed last in the sort sequence (depending on the database).

## Sorting Output

- In some application scenarios it is necessary to **add columns** to a table to obtain suitable **sorting criteria**.

If the students names were stored in the form 'Ann\_Smith', sorting by last name is more or less impossible. Having separate columns for first and last name is better.

- **Null values** are all listed first or all listed last in the sort sequence (depending on the database).
- Since the effect of ORDER BY is purely “cosmetic”, ORDER BY may **not** be applied to a subquery.

## Overview

1. SELECT-FROM-WHERE Blocks, Tuple Variables
2. Subqueries, Non-Monotonic Constructs
3. Aggregations I: Aggregation Functions
4. Aggregations II: GROUP BY, HAVING
5. UNION, Conditional Expressions
6. ORDER BY
7. SQL-92 Joins, Outer Join



# Example Database

STUDENTS

<u>SID</u>	FIRST	LAST	EMAIL
101	Ann	Smith	...
102	Michael	Jones	(null)
103	Richard	Turner	...
104	Maria	Brown	...

EXERCISES

<u>CAT</u>	<u>ENO</u>	TOPIC	MAXPT
H	1	Rel.Alg.	10
H	2	SQL	10
M	1	SQL	14

RESULTS

<u>SID</u>	<u>CAT</u>	<u>ENO</u>	POINTS
101	H	1	10
101	H	2	8
101	M	1	12
102	H	1	9
102	H	2	9
102	M	1	10
103	H	1	5
103	M	1	7

# Joins

Up to version SQL-86, there were no explicit joins in queries.

- Instead, Cartesian products of relations (`FROM`) are specified and then filtered via `WHERE`.

Up to version SQL-86, there were no explicit joins in queries.

- Instead, Cartesian products of relations (FROM) are specified and then filtered via WHERE.

## Natural join of RESULTS and EXERCISES

```
SELECT  R.CAT AS CAT, R.ENO AS ENO,  
        SID, POINTS, TOPIC, MAXPT  
FROM    RESULTS R, EXERCISES E  
WHERE   R.CAT = E.CAT AND R.ENO = E.ENO
```

# Joins

Since SQL-92 there are explicit join operations.

Since SQL-92 there are explicit join operations.

## **“Natural join” in SQL-92**

```
SELECT  SID, ENO, (POINTS / MAXPT) * 100
FROM    RESULTS NATURAL JOIN EXERCISES
WHERE   CAT = 'H'
```

Since SQL-92 there are explicit join operations.

## “Natural join” in SQL-92

```
SELECT  SID, ENO, (POINTS / MAXPT) * 100
FROM    RESULTS NATURAL JOIN EXERCISES
WHERE   CAT = 'H'
```

The keywords `NATURAL JOIN` lead the DBMS to automatically add the join predicate to the query:

```
RESULTS.CAT = EXERCISES.CAT
AND RESULTS.ENO = EXERCISES.ENO
```

# Joins

Since SQL-92 there are explicit join operations.

## “Natural join” in SQL-92

```
SELECT  SID, ENO, (POINTS / MAXPT) * 100
FROM    RESULTS NATURAL JOIN EXERCISES
WHERE   CAT = 'H'
```

The keywords `NATURAL JOIN` lead the DBMS to automatically add the join predicate to the query:

```
RESULTS.CAT = EXERCISES.CAT
AND RESULTS.ENO = EXERCISES.ENO
```

In a **natural join**, the join predicate arises implicitly by comparing all columns that have the same name in both tables.

# Inner and Outer Joins

SQL-92 supports the following **join types** ([. .] is optional):

- [INNER] JOIN: Usual join.
- LEFT [OUTER] JOIN: Preserves rows of left table.
- RIGHT [OUTER] JOIN: Preserves rows of right table.
- FULL [OUTER] JOIN: Preserves rows of both tables.
- CROSS JOIN: Cartesian product (all combinations).



# Inner and Outer Joins

SQL-92 supports the following **join types** ([. . .] is optional):

- [INNER] JOIN: Usual join.
- LEFT [OUTER] JOIN: Preserves rows of left table.
- RIGHT [OUTER] JOIN: Preserves rows of right table.
- FULL [OUTER] JOIN: Preserves rows of both tables.
- CROSS JOIN: Cartesian product (all combinations).

A (natural) join ( $\bowtie$ ) eliminates tuples without partner.

<table border="1"><thead><tr><th>A</th><th>B</th></tr></thead><tbody><tr><td>a<sub>1</sub></td><td>b<sub>1</sub></td></tr><tr><td>a<sub>2</sub></td><td>b<sub>2</sub></td></tr></tbody></table>	A	B	a <sub>1</sub>	b <sub>1</sub>	a <sub>2</sub>	b <sub>2</sub>	$\bowtie$	<table border="1"><thead><tr><th>B</th><th>C</th></tr></thead><tbody><tr><td>b<sub>2</sub></td><td>c<sub>2</sub></td></tr><tr><td>b<sub>3</sub></td><td>c<sub>3</sub></td></tr></tbody></table>	B	C	b <sub>2</sub>	c <sub>2</sub>	b <sub>3</sub>	c <sub>3</sub>	=	<table border="1"><thead><tr><th>A</th><th>B</th><th>C</th></tr></thead><tbody><tr><td>a<sub>2</sub></td><td>b<sub>2</sub></td><td>c<sub>2</sub></td></tr></tbody></table>	A	B	C	a <sub>2</sub>	b <sub>2</sub>	c <sub>2</sub>
A	B																					
a <sub>1</sub>	b <sub>1</sub>																					
a <sub>2</sub>	b <sub>2</sub>																					
B	C																					
b <sub>2</sub>	c <sub>2</sub>																					
b <sub>3</sub>	c <sub>3</sub>																					
A	B	C																				
a <sub>2</sub>	b <sub>2</sub>	c <sub>2</sub>																				

# Inner and Outer Joins

SQL-92 supports the following **join types** ([. . .] is optional):

- [INNER] JOIN: Usual join.
- LEFT [OUTER] JOIN: Preserves rows of left table.
- RIGHT [OUTER] JOIN: Preserves rows of right table.
- FULL [OUTER] JOIN: Preserves rows of both tables.
- CROSS JOIN: Cartesian product (all combinations).

A (natural) join ( $\bowtie$ ) eliminates tuples without partner.

A	B		B	C		A	B	C
a <sub>1</sub>	b <sub>1</sub>	$\bowtie$	b <sub>2</sub>	c <sub>2</sub>	=	a <sub>2</sub>	b <sub>2</sub>	c <sub>2</sub>
a <sub>2</sub>	b <sub>2</sub>		b <sub>3</sub>	c <sub>3</sub>				

The **left outer join** preserves all tuples in its **left** argument:

A	B		B	C		A	B	C
a <sub>1</sub>	b <sub>1</sub>	$\bowtie$	b <sub>2</sub>	c <sub>2</sub>	=	a <sub>1</sub>	b <sub>1</sub>	(null)
a <sub>2</sub>	b <sub>2</sub>		b <sub>3</sub>	c <sub>3</sub>		a <sub>2</sub>	b <sub>2</sub>	c <sub>2</sub>

## Inner and Outer Joins

The **join predicate** may be specified as follows:

# Inner and Outer Joins

The **join predicate** may be specified as follows:

- Keyword `NATURAL` prepended to join operator name.  
Yields comparison of columns with the same name.

# Inner and Outer Joins

The **join predicate** may be specified as follows:

- Keyword `NATURAL` prepended to join operator name.  
Yields comparison of columns with the same name.
- `USING (A1, ..., An)` appended to join operator name.

The  $A_i$  must be columns appearing in both tables. The join predicate then is  $R.A_1=S.A_1$  AND ... AND  $R.A_n=S.A_n$ .

# Inner and Outer Joins

The **join predicate** may be specified as follows:

- Keyword `NATURAL` prepended to join operator name.  
Yields comparison of columns with the same name.
- `USING (A1, ..., An)` appended to join operator name.  
The  $A_i$  must be columns appearing in both tables. The join predicate then is  $R.A_1=S.A_1 \text{ AND } \dots \text{ AND } R.A_n=S.A_n$ .
- `ON Condition` appended to join operator name.

# Inner and Outer Joins

The **join predicate** may be specified as follows:

- Keyword **NATURAL** prepended to join operator name.  
Yields comparison of columns with the same name.
- **USING** ( $A_1, \dots, A_n$ ) appended to join operator name.  
The  $A_i$  must be columns appearing in both tables. The join predicate then is  $R.A_1=S.A_1$  AND ... AND  $R.A_n=S.A_n$ .
- **ON**  $\langle Condition \rangle$  appended to join operator name.
- **CROSS JOIN** has no join predicate.

# Inner and Outer Joins

```
STUDENTS(SID, FIRST, LAST, EMAIL)
EXERCISES(CAT, ENO, TOPIC, MAXPT)
RESULTS(SID→STUDENTS, (CAT,ENO)→EXERCISES, POINTS)
```

## Number of submission per homework (0 if no submission)

```
SELECT    E.ENO, COUNT(SID)
FROM      EXERCISES E LEFT OUTER JOIN RESULTS R
          ON E.CAT = R.CAT AND E.ENO = R.ENO
WHERE     E.CAT = 'H'
GROUP BY  E.ENO
```

- All exercises are present in the result of the left outer join.



# Inner and Outer Joins

```
STUDENTS(SID, FIRST, LAST, EMAIL)
EXERCISES(CAT, ENO, TOPIC, MAXPT)
RESULTS(SID→STUDENTS, (CAT,ENO)→EXERCISES, POINTS)
```

## Number of submission per homework (0 if no submission)

```
SELECT    E.ENO, COUNT(SID)
FROM      EXERCISES E LEFT OUTER JOIN RESULTS R
          ON E.CAT = R.CAT AND E.ENO = R.ENO
WHERE     E.CAT = 'H'
GROUP BY  E.ENO
```

- All exercises are present in the result of the left outer join.
- In exercises without solutions, columns SID and POINTS will contain NULL.

# Inner and Outer Joins

```
STUDENTS(SID, FIRST, LAST, EMAIL)
EXERCISES(CAT, ENO, TOPIC, MAXPT)
RESULTS(SID→STUDENTS, (CAT,ENO)→EXERCISES, POINTS)
```

## Number of submission per homework (0 if no submission)

```
SELECT    E.ENO, COUNT(SID)
FROM      EXERCISES E LEFT OUTER JOIN RESULTS R
          ON E.CAT = R.CAT AND E.ENO = R.ENO
WHERE     E.CAT = 'H'
GROUP BY  E.ENO
```

- All exercises are present in the result of the left outer join.
- In exercises without solutions, columns SID and POINTS will contain NULL.
- COUNT(SID) ignores rows where SID IS NULL.

# Inner and Outer Joins

## Equivalent query without OUTER JOIN (12 vs. 5 lines).

```
SELECT    E.ENO, COUNT(*)
FROM      EXERCISES E, RESULTS R
WHERE     E.CAT = 'H' AND R.CAT = 'H'
AND       E.ENO = R.ENO
GROUP BY  E.ENO
UNION ALL
SELECT    E.ENO, 0
FROM      EXERCISES E
WHERE     E.CAT = 'H'
AND       E.ENO NOT IN (SELECT  R.ENO
                        FROM      RESULTS R
                        WHERE     R.CAT = 'H')
```

# Inner and Outer Joins

**Is there a problem with the following query?**

**“Number of homeworks solved per student (including 0).”**

```
SELECT    FIRST, LAST, COUNT(ENO)
FROM      STUDENTS S LEFT OUTER JOIN RESULTS R
          ON S.SID = R.SID
WHERE     R.CAT = 'H'
GROUP BY  S.SID, FIRST, LAST
```

## Inner and Outer Joins

Is there a problem with the following query?

“Number of homeworks solved per student (including 0).”

```
SELECT    FIRST, LAST, COUNT(ENO)
FROM      STUDENTS S LEFT OUTER JOIN RESULTS R
          ON S.SID = R.SID
WHERE     R.CAT = 'H'
GROUP BY  S.SID, FIRST, LAST
```

It is generally wise to restrict the outer join inputs **before** the outer join is performed (or move restrictions into the ON clause).

# Inner and Outer Joins

## Is there a problem with the following query?

**“Number of homeworks solved per student (including 0).”**

```
SELECT    FIRST, LAST, COUNT(ENO)
FROM      STUDENTS S LEFT OUTER JOIN RESULTS R
          ON S.SID = R.SID
WHERE     R.CAT = 'H'
GROUP BY  S.SID, FIRST, LAST
```

It is generally wise to restrict the outer join inputs **before** the outer join is performed (or move restrictions into the ON clause).

## Corrected version of last query.

```
SELECT    FIRST, LAST, COUNT(ENO)
FROM      STUDENTS S LEFT OUTER JOIN
          (SELECT  SID, ENO
           FROM    RESULTS
           WHERE   CAT = 'H') R
          ON S.SID = R.SID
GROUP BY  S.SID, FIRST, LAST
```

# Inner and Outer Joins

**Will tuples with CAT = 'M' appear in the output?**

```
SELECT  E.CAT, E.ENO, R.SID, R.POINTS
FROM    EXERCISES E LEFT OUTER JOIN RESULTS R
      ON  E.CAT = 'H'
        AND R.CAT = 'H'
        AND E.ENO = R.ENO
```

# Inner and Outer Joins

**Will tuples with CAT = 'M' appear in the output?**

```
SELECT  E.CAT, E.ENO, R.SID, R.POINTS
FROM    EXERCISES E LEFT OUTER JOIN RESULTS R
      ON  E.CAT = 'H'
        AND R.CAT = 'H'
        AND E.ENO = R.ENO
```

Conditions filtering the **left table** make little sense in a **left outer join predicate**.

The left outer join will make the “filtered” tuples appear anyway (as join partners for unmatched RESULTS tuples).



# SQL: Objectives

After completing this chapter, you should be able to:

- write **advanced SQL** queries
  - with multiple tuple variables over different/the same relation
  - with nested queries
  - ...
- use **aggregation, grouping, union**
- be comfortable with the various **join variants**
- evaluate the **correctness** and **equivalence** of SQL queries
  - this includes sometimes tricky issues of deciding the presence of duplicate result tuples