Databases – SQL

Jörg Endrullis

VU University Amsterdam

SQL :: Running Example

Example Database

Students

<u>sid</u>	first	last	address
101	George	Orwell	London
102	Elvis	Presley	Memphis
103	Lisa	Simpson	Springfield
104	Bart	Simpson	Springfield
105	George	Washington	null

Exercises

<u>category</u>	<u>number</u>	topic	maxPoints
exam	1	SQL	14
homework	1	Logic	10
homework	2	SQL	10

Results

<u>sid</u> → Students	<u>category</u> → Exercises	<u>number</u> → Exercises	points
101	exam	1	12
101	homework	1	10
101	homework	2	8
102	exam	1	10
102	homework	1	9
102	homework	2	9
103	exam	1	7
103	homework	1	5

SQL :: Basic Syntax

Basic SQL Query Syntax

Basic SQL query (extensions follow)

```
select  A1, ..., An
from    R1, ..., Rm
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* `true`.

The **`select`** clause

- ... specifies the attributes of the **result**.
- *Here* $*$ *means output all attributes occurring in* $R_1, \ldots, R_m$.

The From Clause

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

Exercises			
category	**number**	**topic**	**maxPoints**
exam	1	SQL	14
homework	1	Logic	10
homework	2	SQL	10

```
select   E.number, E.topic
from     Exercises E
where    E.category = 'homework'
```

The From Clause

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

Exercises			
category	**number**	**topic**	**maxPoints**
exam	1	SQL	14
homework	1	Logic	10
homework	2	SQL	10

```
select  E.number, E.topic
from    Exercises E
where   E.category = 'homework'
```

The query may be thought of as

> **for all rows** `E` ∈ `Exercises` **do**
>   **if** `E.category = 'homework'` **then**
>   **print** `E.number, E.topic`
>   **end if**
> **end for**

Tuple variable `E` iterates over the rows of `Exercises`.

The From Clause

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

Exercises

category	number	topic	maxPoints
exam	1	SQL	14
homework	1	Logic	10
homework	2	SQL	10

```
select   E.number, E.topic
from     Exercises E
where    E.category = 'homework'
```

Query Result

number	topic
1	Logic
2	SQL

The query may be thought of as

for all rows `E` $\in$ `Exercises` **do**
  if `E.category = 'homework'` **then**
    print `E.number, E.topic`
  end if
end for

Tuple variable `E` iterates over the rows of `Exercises`.

The From Clause

For each table in the `from` clause there is a tuple variable.

The From Clause

For each table in the `from` clause there is a tuple variable.

- If the the name of the tuple variable is not given explicitly, the variable will have the name of the relation:

```
select  Exercises.number, Exercises.topic
from    Exercises
where   Exercises.category = 'homework'
```

The From Clause

For each table in the `from` clause there is a tuple variable.

- If the the name of the tuple variable is not given explicitly, the variable will have the name of the relation:

```
select  Exercises.number, Exercises.topic
from    Exercises
where   Exercises.category = 'homework'
```

In other words, `from Exercises` is understood as:

```
from Exercises Exercises
```

The From Clause

For each table in the `from` clause there is a tuple variable.

- If the the name of the tuple variable is not given explicitly, the variable will have the name of the relation:

```
select  Exercises.number, Exercises.topic
from    Exercises
where   Exercises.category = 'homework'
```

  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.number` will yield an error.

Attribute References

```
Students(sid, first, last, address)      Results(sid, category, number, points)
```

Let R be a tuple variable and A an attribute of R.

Attributes are referenced in the form

$$R\,.\,A$$

If R is the **only** tuple variable with attribute A, then it suffices

$$A$$

Attribute References

`Students(sid, first, last, address)`   `Results(sid, category, number, points)`

Let R be a tuple variable and A an attribute of R.

Attributes are referenced in the form

$$R.A$$

If R is the **only** tuple variable with attribute A, then it suffices

$$A$$

For example,

```
select  category, number, points
from    Students S, Results R
where   S.sid = R.sid
    and first = 'George' and last = 'Orwell'
```

- `first`, `last` can only refer to `S`
- `category`, `number`, `points` can only refer to `R`
- `sid` on its would be **ambiguous** (could refer to `S` or `R`)

Ambiguous Attribute References

```
Exercises(category, number, topic, maxPoints)

Results(sid, category, number, points)
```

Consider the following query:

```
select number, sid, points, maxPoints
from   Results R, Exercises E
where  R.number = E.number
   and R.category = 'homework' and E.category = 'homework'
```

Ambiguous Attribute References

```
Exercises(category, number, topic, maxPoints)

Results(sid, category, number, points)
```

Consider the following query:

```
select number, sid, points, maxPoints
from   Results R, Exercises E
where  R.number = E.number
   and R.category = 'homework' and E.category = 'homework'
```

In the `select` clause `number` is ambiguous.

Ambiguous Attribute References

Exercises(<u>category</u>, <u>number</u>, topic, maxPoints)

Results(<u>sid</u>, <u>category</u>, <u>number</u>, points)

Consider the following query:

```
select number, sid, points, maxPoints
from   Results R, Exercises E
where  R.number = E.number
   and R.category = 'homework' and E.category = 'homework'
```

In the `select` clause `number` is ambiguous.

Although forced to be equal by the join condition, SQL requires the user to specify whether `number` refers to `R` or `E`.

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

SQL :: Joins

Joins

```
Students(sid, first, last, address)      Results(sid, category, number, points)
```

Consider a query with two tuple variables:

select $A_1, \ldots, A_n$
from Students S, Results R
where C

Joins

`Students(sid, first, last, address)`   `Results(sid, category, number, points)`

Consider a query with two tuple variables:

select $A_1, \ldots, A_n$
from Students S, Results R
where C

- `S` ranges over 5 rows in `Students`,
- `R` ranges over 8 rows in `Results`.

Joins

`Students(sid, first, last, address)`   `Results(sid, category, number, points)`

Consider a query with two tuple variables:

```
select  A1, ..., An
from    Students S, Results R
where   C
```

- S ranges over 5 rows in `Students`,
- R ranges over 8 rows in `Results`.

Conceptually, all $5 \cdot 8 = 40$ combinations will be considered:

for all rows S $\in$ `Students` **do**
  for all rows R $\in$ `Results` **do**
    if C **then**
      print $A_1, \ldots, A_n$
    end if
  end for
end for

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 example, if *C* contains the join condition

```
S.sid = R.sid
```

then the DBMS might execute the query efficiently by:

- loop over the row in `Results`,
- find matching `Students` row via an **index** on `Students.sid`

DBMS typically create an index over the key attributes.

Joins

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

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

For example, if *C* contains the join condition

`S.sid = R.sid`

then the DBMS might execute the query efficiently by:

- loop over the row in `Results`,
- find matching `Students` row via an **index** on `Students.sid`

DBMS typically create an index over the key attributes.

For understanding the **semantics** of a query, the simple nested **foreach algorithm** suffices!

The query optimiser may use any algorithm that produces the **exact same output** (except possibly the tuple order).

Joins

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

```
select category, number, points
from   Students S, Results R
where  S.sid = R.sid       -- Join Condition
   and first = 'George' and last = 'Orwell'
```

Students			
sid	first	last	address
101	George	Orwell	London
102	Elvis	Presley	Memphis
103	Lisa	Simpson	Springfield
104	Bart	Simpson	Springfield
105	George	Washington	null

Results			
sid	category	number	points
101	exam	1	12
101	homework	1	10
101	homework	2	8
102	exam	1	10
102	homework	1	9
102	homework	2	9
103	exam	1	7
103	homework	1	5

Joins

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

```
select category, number, points
from   Students S, Results R
where  S.sid = R.sid        -- Join Condition
   and first = 'George' and last = 'Orwell'
```

Students			
sid	first	last	address
101	George	Orwell	London
102	Elvis	Presley	Memphis
103	Lisa	Simpson	Springfield
104	Bart	Simpson	Springfield
105	George	Washington	null

Results			
sid	category	number	points
101	exam	1	12
101	homework	1	10
101	homework	2	8
102	exam	1	10
102	homework	1	9
102	homework	2	9
103	exam	1	7
103	homework	1	5

Output of this query?

```
select  S.first, S.last
from    Students S, Results R
where   category = 'homework' and number = 1
```

Joins

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

Joins

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

In this query, all three tuple variables are connected:

```
select S.first, S.last, E.category, E.number
from   Students S, Results R, Exercises E
where  S.sid = R.sid
   and R.category = E.category and R.number = E.number
```

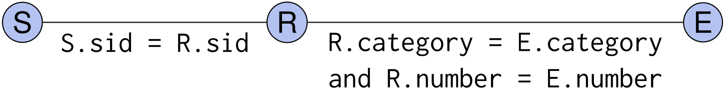
Joins

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

In this query, all three tuple variables are connected:

```
select S.first, S.last, E.category, E.number
from   Students S, Results R, Exercises E
where  S.sid = R.sid
   and R.category = E.category and R.number = E.number
```

The tuple variable are connected as follows:



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

Exercise

```
Students(sid, first, last, address)
Exercises(category, number, topic, maxPoints)
Results(sid → Students, (category, number) → Exercises, points)
```

Formulate the following query in SQL

The topics of all exercises solved by George Orwell?

Exercise

```
Students(sid, first, last, address)
Exercises(category, number, topic, maxPoints)
Results(sid → Students, (category, number) → Exercises, points)
```

Formulate the following query in SQL

The topics of all exercises solved by George Orwell?

We need tuple variables for `Students` and `Exercises`.

```
select E.topic
from   Students S, Exercises E
where  S.first = 'George' and S.last = 'Orwell'
```

Exercise

```
Students(sid, first, last, address)
Exercises(category, number, topic, maxPoints)
Results(sid → Students, (category, number) → Exercises, points)
```

Formulate the following query in SQL

The topics of all exercises solved by George Orwell?

We need tuple variables for `Students` and `Exercises`.

```
select E.topic
from   Students S, Exercises E
where  S.first = 'George' and S.last = 'Orwell'
```

Problem: `S` and `E` are **unconnected**!

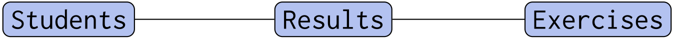
Exercise

Students(sid, first, last, address)
Exercises(category, number, topic, maxPoints)
Results(sid → Students, (category, number) → Exercises, points)

Formulate the following query in SQL

The topics of all exercises solved by George Orwell?

The **connection graph** with foreign key relations:



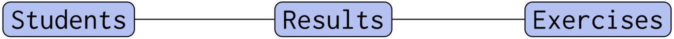
Exercise

```
Students(sid, first, last, address)
Exercises(category, number, topic, maxPoints)
Results(sid → Students, (category, number) → Exercises, points)
```

Formulate the following query in SQL

The topics of all exercises solved by George Orwell?

The **connection graph** with foreign key relations:



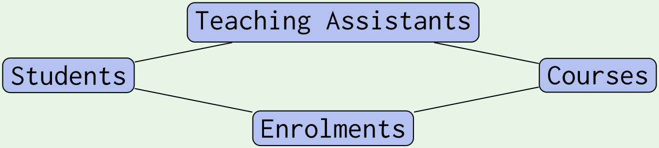
We establish the link via a tuple variable `R` over `Results`:

```
select E.topic
from   Students S, Exercises E, Results R
where  S.first = 'George' and S.last = 'Orwell'
   and S.sid = R.sid
   and R.category = E.category and R.number = E.number
```

Cycles in the Connection Graph

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

A database of course enrolments, could have the cycle:



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, address)
Exercises(category, number, topic, maxPoints)
Results(sid → Students, (category, number) → Exercises, points)
```

Results for homework 1

```
select  R.sid, R.points
from    Results R, Exercises E
where   R.category = E.category and R.number = E.number
    and E.category = 'homework' and E.number = 1
```

Unnecessary Joins

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

Query will run slowly if the optimizer overlooks the redundancy.

```
Students(sid, first, last, address)
Exercises(category, number, topic, maxPoints)
Results(sid → Students, (category, number) → Exercises, points)
```

Results for homework 1

```
select  R.sid, R.points
from    Results R, Exercises E
where   R.category = E.category and R.number = E.number
    and E.category = 'homework' and E.number = 1
```

Will the following query produce the same Results?

```
select  sid, points
from    Results R
where   R.category = 'homework' and R.number = 1
```

Unnecessary Joins

Exercises			
category	number	topic	maxPoints
exam	1	SQL	14
homework	1	Logic	10
homework	2	SQL	10

Results			
sid	category	number	points
101	exam	1	12
101	homework	1	10
101	homework	2	8
102	exam	1	10
102	homework	1	9
102	homework	2	9
103	exam	1	7
103	homework	1	5

What will be the result of this query?

```
select   R.sid, R.points
from     Results R, Exercises E
where    R.category = 'homework' and R.number = 1
```

Unnecessary Joins

Students			
sid	**first**	**last**	**address**
101	George	Orwell	London
102	Elvis	Presley	Memphis
103	Lisa	Simpson	Springfield
104	Bart	Simpson	Springfield
105	George	Washington	null

Results			
sid	**category**	**number**	**points**
101	exam	1	12
101	homework	1	10
101	homework	2	8
102	exam	1	10
102	homework	1	9
102	homework	2	9
103	exam	1	7
103	homework	1	5

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
```

SQL :: Self Joins

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.

```
Students(sid, first, last, address)
Exercises(category, number, topic, maxPoints)
Results(sid → Students, (category, number) → Exercises, points)
```

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.

```
Students(sid, first, last, address)
Exercises(category, number, topic, maxPoints)
Results(sid → Students, (category, number) → Exercises, points)
```

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.category = 'homework' and H1.number = 1
    and H2.category = 'homework' and H2.number = 2
    and H1.points = 9 and H2.points = 9
```

Self Joins

Students that solved at least two homework assignments

(This may also be solved using aggregations.)

```
select S.first, S.last
from   Students S, Results R1, Results R2
where  S.sid = R1.sid and R1.category = 'homework'
   and S.sid = R2.sid and R2.category = 'homework'
```

"Unexpected" result

What is going wrong here?

Self Joins

Students that solved at least two homework assignments

(This may also be solved using aggregations.)

```
select S.first, S.last
from   Students S, Results R1, Results R2
where  S.sid = R1.sid and R1.category = 'homework'
   and S.sid = R2.sid and R2.category = 'homework'
```

"Unexpected" result

What is going wrong here?

We need to ensure that `R1` and `R2` refer to distinct results:

```
⋮
and (R1.sid <> R2.sid or
     R1.category <> R2.category or
     R1.number <> R2.number)
```

SQL :: Duplicate 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 category, number
from   Results
```

category	number
exam	1
exam	1
exam	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 category, number
from   Results
```

category	number
exam	1
exam	1
exam	1
⋮	⋮

The **`distinct`** modifier may be applied to the `select` clause to request explicit duplicate row elimination

```
select distinct category, number
from   Results
```

category	number
exam	1
homework	1
homework	2

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 category, number
from   Results
```

category	number
exam	1
exam	1
exam	1
⋮	⋮

The **`distinct`** modifier may be applied to the `select` clause to request explicit duplicate row elimination

```
select distinct category, number
from   Results
```

category	number
exam	1
homework	1
homework	2

Unexpected duplicates in the result can be a sign of mistakes!

Avoid Unnecessary Duplicate Elimination

Sufficient condition for superfluous `distinct`

Assumption: `where` clause is a conjunction (`and`).

1. Let $\mathcal{K}$ be the set of attributes in the `select` clause.
2. If $A = c$ in the `where` clause and c a constant, add A to $\mathcal{K}$.
3. If $A = B$ in the `where` clause and $B \in \mathcal{K}$, add A to $\mathcal{K}$.
4. If $\mathcal{K}$ has a key of a variable X, add all attributes of X to $\mathcal{K}$.
5. Repeat 2, 3 and 4 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.

Avoid Unnecessary Duplicate Elimination

```
select  distinct S.first, S.last, R.number, R.points
from    Students S, Results R
where   R.category = 'homework' and R.sid = S.sid
```

Let us assume that { `first`, `last` } is a key for `Students`.

Avoid Unnecessary Duplicate Elimination

```
select  distinct S.first, S.last, R.number, R.points
from    Students S, Results R
where   R.category = 'homework' and R.sid = S.sid
```

Let us assume that { `first`, `last` } is a key for `Students`.

1. Initialise $\mathcal{K} =$

Avoid Unnecessary Duplicate Elimination

```
select  distinct S.first, S.last, R.number, R.points
from    Students S, Results R
where   R.category = 'homework' and R.sid = S.sid
```

Let us assume that { `first`, `last` } is a key for `Students`.

1. Initialise $\mathcal{K} = \{$ `S.first`, `S.last`, `R.number`, `R.points` $\}$.

Avoid Unnecessary Duplicate Elimination

```
select  distinct S.first, S.last, R.number, R.points
from    Students S, Results R
where   R.category = 'homework' and R.sid = S.sid
```

Let us assume that $\{$`first`, `last`$\}$ is a key for `Students`.

1. Initialise $\mathcal{K} = \{$`S.first`, `S.last`, `R.number`, `R.points`$\}$.
2. $\mathcal{K} + \{$`R.category`$\}$ because of `R.category = 'homework'`

Avoid Unnecessary Duplicate Elimination

```
select  distinct S.first, S.last, R.number, R.points
from    Students S, Results R
where   R.category = 'homework' and R.sid = S.sid
```

Let us assume that { `first`, `last` } is a key for `Students`.

1. Initialise $\mathcal{K} =$ { `S.first`, `S.last`, `R.number`, `R.points` }.
2. $\mathcal{K}$ + { `R.category` } because of `R.category = 'homework'`
4. $\mathcal{K}$ + { `S.sid`, `S.address` } since $\mathcal{K}$ contains a key of `Students`

Avoid Unnecessary Duplicate Elimination

```
select  distinct S.first, S.last, R.number, R.points
from    Students S, Results R
where   R.category = 'homework' and R.sid = S.sid
```

Let us assume that { `first`, `last` } is a key for `Students`.

1. Initialise $\mathcal{K} = \{$ `S.first`, `S.last`, `R.number`, `R.points` $\}$.
2. $\mathcal{K} + \{$ `R.category` $\}$ because of `R.category = 'homework'`
4. $\mathcal{K} + \{$ `S.sid`, `S.address` $\}$ since $\mathcal{K}$ contains a key of `Students`
3. $\mathcal{K} + \{$ `R.sid` $\}$ because of the conjunct `S.sid = R.sid`

Avoid Unnecessary Duplicate Elimination

```
select  distinct S.first, S.last, R.number, R.points
from    Students S, Results R
where   R.category = 'homework' and R.sid = S.sid
```

Let us assume that { `first`, `last` } is a key for `Students`.

1. Initialise $\mathcal{K} =$ { `S.first`, `S.last`, `R.number`, `R.points` }.
2. $\mathcal{K} +$ { `R.category` } because of `R.category = 'homework'`
4. $\mathcal{K} +$ { `S.sid`, `S.address` } since $\mathcal{K}$ contains a key of `Students`
3. $\mathcal{K} +$ { `R.sid` } because of the conjunct `S.sid = R.sid`

Finally, $\mathcal{K}$ contains a key of

- `Students S` { `S.first`, `S.last` } and
- `Results R` { `R.sid`, `R.cat`, `R.eno` }

Avoid Unnecessary Duplicate Elimination

```
select  distinct S.first, S.last, R.number, R.points
from    Students S, Results R
where   R.category = 'homework' and R.sid = S.sid
```

Let us assume that { `first, last` } is a key for `Students`.

1. Initialise $\mathcal{K} =$ { `S.first, S.last, R.number, R.points` }.
2. $\mathcal{K} +$ { `R.category` } because of `R.category = 'homework'`
4. $\mathcal{K} +$ { `S.sid, S.address` } since $\mathcal{K}$ contains a key of `Students`
3. $\mathcal{K} +$ { `R.sid` } because of the conjunct `S.sid = R.sid`

Finally, $\mathcal{K}$ contains a key of

- `Students S` { `S.first, S.last` } and
- `Results R` { `R.sid, R.cat, R.eno` }

Thus `distinct` is superfluous.

Avoid Unnecessary Duplicate Elimination

```
select  distinct S.first, S.last, R.number, R.points
from    Students S, Results R
where   R.category = 'homework' and R.sid = S.sid
```

Let us assume that { `first, last` } is a key for `Students`.

1. Initialise $\mathcal{K} =$ { `S.first, S.last, R.number, R.points` }.
2. $\mathcal{K} +$ { `R.category` } because of `R.category = 'homework'`
4. $\mathcal{K} +$ { `S.sid, S.address` } since $\mathcal{K}$ contains a key of `Students`
3. $\mathcal{K} +$ { `R.sid` } because of the conjunct `S.sid = R.sid`

Finally, $\mathcal{K}$ 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` } is not key of `Students`, the test would fail.
Rightly so, since then the result could contain duplicates.

Query Formulation Traps

Typical mistakes

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

SQL :: Non-Monotonic Queries

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.

- Currently, `Bart Simpson` 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.

- Currently, `Bart Simpson` would be a correct answer.
- This answer is no longer valid after:

```
insert into Results values (104, 'homework', 1, 8)
```

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.

- Currently, `Bart Simpson` would be a correct answer.
- This answer is no longer valid after:

```
insert into Results values (104, 'homework', 1, 8)
```

Such **non-monotonic** queries **cannot** be formulated with the SQL constructs that we have seen so far.

Non-Monotonic Behaviour

In natural language, queries that contain formulations like

"**there is no**" "**does not exists**"	}	**negated existential quantification**
"**for all**" "**the minimum/maximum**"	}	**universally quantification**

indicate non-monotonic behaviour.

Non-Monotonic Behaviour

In natural language, queries that contain formulations like

"**there is no**" "**does not exists**"	**negated existential quantification**
"**for all**" "**the minimum/maximum**"	**universally quantification**

indicate non-monotonic behaviour.

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

SQL :: Not In

Not In

With

- `in`
- `not in`

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   category = 'homework')
```

Query Result	
first	**last**
Bart	Simpson
George	Washington

Not In

```
select  first, last
from    Students
where   sid not in (select  sid
                    from    Results
                    where   category = 'homework')
```

Conceptually . . .

The **subquery** is evaluated before the **main query**:

Students			
sid	**first**	**last**	**address**
101	George	Orwell	London
102	Elvis	Presley	Memphis
103	Lisa	Simpson	Springfield
104	Bart	Simpson	Springfield
105	George	Washington	null

Subquery result
sid
101
102
103

Then, for every tuple of `Students`, a matching `sid` is searched in the subquery result. If there is none, the tuple is output.

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 In

Exercises(<u>category</u>, <u>number</u>, topic, maxPoints)

Results(<u>sid</u>, <u>category</u>, <u>number</u>, points)

Topics of homework tasks solved by at least one student

Not In

Exercises(<u>category</u>, <u>number</u>, topic, maxPoints)

Results(<u>sid</u>, <u>category</u>, <u>number</u>, points)

Topics of homework tasks solved by at least one student

```
select topic
from   Exercises
where  category = 'homework'
   and number in (select number
                  from   Results
                  where  category = 'homework')
```

Not In

```
Exercises(category, number, topic, maxPoints)

Results(sid, category, number, points)
```

Topics of homework tasks solved by at least one student

```
select topic
from   Exercises
where  category = 'homework'
   and number in (select number
                  from   Results
                  where  category = 'homework')
```

Is there a difference to this query?
(with or without `distinct`)

```
select distinct topic
from   Exercises E, Results R
where  E.category = 'homework'
   and R.category = 'homework'
   and E.number = R.number
```

SQL :: Not Exists

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.category = 'homework'
                     and      R.sid = S.sid)
```

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

- Then the outer query and subquery are **correlated.**
- The subquery is said to be "**parameterized**".

You may also do so for `in` *subqueries.*

Not Exists

Students that have not submitted any homework

```
select  first, last
from    Students S
where   not exists (select  *
                    from    Results R
                    where   R.category = 'homework'
                    and     R.sid = S.sid)
```

Query Result	
first	**last**
Bart	Simpson
George	Washington

Not Exists

Students that have not submitted any homework

```
select   first, last
from     Students S
where    not exists (select   *
                     from     Results R
                     where    R.category = 'homework'
                     and      R.sid = S.sid)
```

Query Result	
first	**last**
Bart	Simpson
George	Washington

Conceptually . . .

Tuple variable `S` loops over the 5 rows in `Students`.

The subquery is evaluated 5 times.

The DBMS is free to choose a more efficient equivalent evaluation strategy (for instance, 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.category = 'homework'
                     and     R.sid = S.sid)
```

First, `S` is the `Students` tuple

sid	first	last	address
101	George	Orwell	London

Not Exists

Students that have not submitted any homework

```
select   first, last
from     Students S
where    not exists (select   *
                     from     Results R
                     where    R.category = 'homework'
                     and      R.sid = S.sid)
```

First, S is the Students tuple

sid	first	last	address
101	George	Orwell	London

In the subquery, S.sid is instantiated by 101:

```
select   *
from     Results R
where    R.category = 'homework'
and      R.sid = 101
```

Query Result

sid	category	number	points
101	homework	1	10
101	homework	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.category = 'homework'
                      and     R.sid = S.sid )
```

First, S is the Students tuple

sid	first	last	address
101	George	Orwell	London

In the subquery, S.sid is instantiated by 101:

```
select   *
from     Results R
where    R.category = 'homework'
and      R.sid = 101
```

Query Result

sid	category	number	points
101	homework	1	10
101	homework	2	8

The result is non-empty. Thus the not exists is false 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.category = 'homework'
                      and      R.sid = S.sid)
```

Finally, `S` is the `Students` tuple

sid	first	last	address
105	George	Washington	null

Not Exists

Students that have not submitted any homework

```
select   first, last
from     Students S
where    not exists (select   *
                     from     Results R
                     where    R.category = 'homework'
                     and      R.sid = S.sid)
```

Finally, `S` is the `Students` tuple

sid	first	last	address
105	George	Washington	null

In the subquery, `S.sid` is instantiated by `105`:

```
select   *
from     Results R
where    R.category = 'homework'
and      R.sid = 105
```

Query Result

sid	category	number	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.category = 'homework'
                      and      R.sid = S.sid )
```

Finally, `S` is the `Students` tuple

sid	first	last	address
105	George	Washington	null

In the subquery, `S.sid` is instantiated by `105`:

```
select   *
from     Results R
where    R.category = 'homework'
and      R.sid = 105
```

Query Result

sid	category	number	points
(no rows selected)			

The result is empty. So the `not exists` is true for **this** `S`.

Not Exists

The subquery may use tuple variables from outer query.
The **converse is illegal**!

Wrong!

```
select   first, last, R.number
from     Students S
where    not exists (select  *
                     from    Results R
                     where   R.category = 'homework'
                     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!

Wrong!

```
select  first, last
from    Students S
where   not exists (select *
                    from   Results R
                    where  category = 'homework')
```

If there is at least one homework result, then 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  category = 'homework')
```

If there is at least one homework result, then 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`).

Exists

We can also 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.category = 'homework')
```

Query Result		
sid	**first**	**last**
101	George	Orwell
102	Elvis	Presley
103	Lisa	Simpson

Exists

We can also 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.category = 'homework')
```

Query Result		
sid	**first**	**last**
101	George	Orwell
102	Elvis	Presley
103	Lisa	Simpson

Can we reformulate the above without using `exists`?

SQL :: For All

For All

Existential quantifier: $\exists X(\varphi)$

- Meaning: There is an X that satisfies formula φ.

Universal quantifier: $\forall X(\varphi)$

- Meaning: For all X, formula φ is satisfied (true).

For All

Existential quantifier: $\exists X(\varphi)$

- Meaning: There is an X that satisfies formula φ.

Universal quantifier: $\forall X(\varphi)$

- Meaning: For all X, formula φ is satisfied (true).

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

SQL offers only the existential quantifier `exists`.

However, there is a restricted form: `>= all`.

For All

Existential quantifier: $\exists X(\varphi)$

- Meaning: There is an X that satisfies formula φ.

Universal quantifier: $\forall X(\varphi)$

- Meaning: For all X, formula φ is satisfied (true).

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

SQL offers only the existential quantifier `exists`.

However, there is a restricted form: `>= all`.

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 exists no car that is not red.

For All & Implication

SQL does also not have $\Rightarrow$. The commonly used pattern

$$\forall X\,(\alpha \Rightarrow \beta)$$

becomes

$$\begin{aligned} & \forall X\,(\alpha \Rightarrow \beta) \\ \equiv\; & \neg\exists X\,\neg(\alpha \Rightarrow \beta) \\ \equiv\; & \neg\exists X\,\neg(\neg\alpha \vee \beta) \\ \equiv\; & \neg\exists X\,(\alpha \wedge \neg\beta) \end{aligned}$$

Exercise: For All & Implication

```
Exercises(category, number, topic, maxPoints)

Results(sid, category, number, points)
```

Who got the best result for homework 1?

Construct the SQL query!

Exercise: For All & Implication

```
Exercises(category, number, topic, maxPoints)

Results(sid, category, number, points)
```

Who got the best result for homework 1?

Construct the SQL query!

In natural language: the students `S` that have a result `X` for homework 1 such that for all result `Y` for homework 1 it holds that `Y.points` is less or equal to `X.points`.

Exercise: For All & Implication

```
Exercises(category, number, topic, maxPoints)

Results(sid, category, number, points)
```

Who got the best result for homework 1?

Construct the SQL query!

In natural language: the students S that have a result X for homework 1 such that for all result Y for homework 1 it holds that Y.points is less or equal to X.points.

In predicate logic (tuple relational calculus):

$$\{ S \mid S \in \texttt{Students} \wedge X \in \texttt{Results} \wedge \texttt{S.sid = X.sid} \\ \wedge \texttt{X.category} = \texttt{'homework'} \wedge \texttt{X.number} = 1 \\ \wedge \forall Y \big((Y \in \texttt{Results} \\ \wedge \texttt{Y.category} = \texttt{'homework'} \wedge \texttt{Y.number} = 1) \\ \Rightarrow \texttt{Y.points} \leqslant \texttt{X.points} \big) \}$$

Exercise: For All & Implication

$$\forall X\,(\varphi_1 \Rightarrow \varphi_2) \;\equiv\; \neg\exists X\,(\varphi_1 \wedge \neg\varphi_2)$$

The formula

$$\{\, \mathtt{S} \mid \mathtt{S} \in \mathtt{Students} \;\wedge\; \mathtt{X} \in \mathtt{Results} \;\wedge\; \mathtt{S.sid = X.sid}$$
$$\wedge\; \mathtt{X.category} = \mathtt{'homework'} \;\wedge\; \mathtt{X.number} = 1$$
$$\wedge\; \forall \mathtt{Y}\, \big((\mathtt{Y} \in \mathtt{Results}$$
$$\wedge\, \mathtt{Y.category} = \mathtt{'homework'} \wedge \mathtt{Y.number} = 1)$$
$$\Rightarrow \mathtt{Y.points} \leqslant \mathtt{X.points}\big) \,\}$$

Exercise: For All & Implication

$$\forall X\,(\varphi_1 \Rightarrow \varphi_2) \;\equiv\; \neg\exists X\,(\varphi_1 \wedge \neg\varphi_2)$$

The formula

$$\begin{aligned}
\{\, S \mid\; & S \in \texttt{Students} \;\wedge\; X \in \texttt{Results} \;\wedge\; \texttt{S.sid = X.sid} \\
& \wedge\; \texttt{X.category} = \texttt{'homework'} \;\wedge\; \texttt{X.number} = 1 \\
& \wedge\; \forall Y \big((Y \in \texttt{Results} \\
& \qquad \wedge\, \texttt{Y.category} = \texttt{'homework'} \wedge \texttt{Y.number} = 1) \\
& \qquad \Rightarrow \texttt{Y.points} \leqslant \texttt{X.points} \big) \,\}
\end{aligned}$$

is **logically equivalent to**

$$\begin{aligned}
\{\, S \mid\; & S \in \texttt{Students} \;\wedge\; X \in \texttt{Results} \;\wedge\; \texttt{S.sid = X.sid} \\
& \wedge\; \texttt{X.category} = \texttt{'homework'} \;\wedge\; \texttt{X.number} = 1 \\
& \wedge\; \neg\exists Y \big((Y \in \texttt{Results} \\
& \qquad \wedge\, \texttt{Y.category} = \texttt{'homework'} \wedge \texttt{Y.number} = 1) \\
& \qquad \wedge\, \texttt{Y.points} > \texttt{X.points} \big) \,\}
\end{aligned}$$

Exercise: For All & Implication

We translate the formula into an SQL query:

$$\{\, \mathtt{S} \mid \mathtt{S} \in \mathtt{Students} \;\wedge\; \mathtt{X} \in \mathtt{Results} \;\wedge\; \mathtt{S.sid} = \mathtt{X.sid}$$
$$\wedge\; \mathtt{X.category} = \mathtt{'homework'} \;\wedge\; \mathtt{X.number} = 1$$
$$\wedge\; \neg\exists\, \mathtt{Y} \big((\mathtt{Y} \in \mathtt{Results}$$
$$\wedge\, \mathtt{Y.category} = \mathtt{'homework'} \wedge \mathtt{Y.number} = 1)$$
$$\wedge\, \mathtt{Y.points} > \mathtt{X.points} \big) \,\}$$

Who got the best result for homework 1?

```
select  first, last, points
from    Students S, Results X
where   S.sid = X.sid
    and X.category = 'homework' and X.number = 1
    and not exists
        (select *
         from   Results Y
         where  Y.category = 'homework' and Y.number = 1
            and Y.points > X.points)
```

SQL :: Nested Subqueries

Nested Subqueries

Subqueries may be nested!

Nested Subqueries

Subqueries may be nested!

List the students who solved all homework assignments

Nested Subqueries

Subqueries may be nested!

List the students who solved all homework assignments

Inner query: all results for student S and homework E.
Middle query: homework of student S for which no result exists.
Outer query: students that have no homework without results.

Nested Subqueries

Subqueries may be nested!

List the students who solved all homework assignments

```
select first, last
from   Students S
where  not exists
       (select *
        from  Exercises E
        where category = 'homework'
          and not exists
              (select *
               from   Results R
               where  R.sid = S.sid
                  and R.number = E.number
                  and R.category = E.category))
```

Inner query: all results for student `S` and homework `E`.
Middle query: homework of student `S` for which no result exists.
Outer query: students that have no homework without results.

Common Errors

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.category = 'homework' and X.number = 1
   and Y.category = 'homework' and Y.number = 1
   and X.points > Y.points
```

If not, what does the query compute?

Common Errors

```
Students(sid, first, last, address)      Results(sid, category, number, 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.category = 'homework' and R.number = 1)
```

Quiz

What goes wrong here?

Common Errors

Students(sid, first, last, address)   Results(sid, category, number, 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.category = 'homework' and R.number = 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, address)
Exercises(category, number, topic, maxPoints)
Results(sid → Students, (category, number) → Exercises, points)
```

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)
```

What is the error in this query?

What is the output of this query? Fix the query.

SQL :: All, Any, Some

All, Any, Some

SQL allows to compare **single value** with all values computed by a **single-column** subquery. Such comparisons may be

- **universally** (`all`), or
- **existentially** (`any`, or equivalently `some`)

quantified.

All, Any, Some

SQL allows to compare **single value** with all values computed by a **single-column** subquery. Such comparisons may be

- **universally** (`all`), or
- **existentially** (`any`, or equivalently `some`)

quantified.

Who got the best result for homework 1?

```
select S.first, S.last, X.points
from   Students S, Results X
where  S.sid = X.sid
   and X.category = 'homework' and X.number = 1
   and X.points >= all (select Y.points
                        from   Results Y
                        where  Y.category = 'homework'
                        and    Y.number = 1)
```

The subquery must yield a single result column.

All, Any, Some

This query is equivalent to the previous query (but uses `any`):

```
select S.first, S.last, X.points
from   Students S, Results X
where  S.sid = X.sid
   and X.category = 'homework' and X.number = 1
   and not X.points < any (select Y.points
                           from   Results Y
                           where  Y.category = 'homework'
                              and Y.number = 1)
```

All, Any, Some

This query is equivalent to the previous query (but uses `any`):

```
select S.first, S.last, X.points
from   Students S, Results X
where  S.sid = X.sid
   and X.category = 'homework' and X.number = 1
   and not X.points < any (select Y.points
                           from   Results Y
                           where  Y.category = 'homework'
                              and Y.number = 1)
```

Note: { `all`, `any`, `some` } do **not** extend SQL's expressiveness.

The statement

$\underline{A\ \texttt{< any}}$ `(select` $\underline{B}$ `from ... where ...)`

is equivalent to

$\underline{\texttt{exists}}$ `(select` B `from ... where ...` $\underline{\texttt{and}\ A < B}$`)`

All, Any, Some

This query is equivalent to the previous query (but uses `any`):

```
select S.first, S.last, X.points
from   Students S, Results X
where  S.sid = X.sid
   and X.category = 'homework' and X.number = 1
   and not X.points < any (select Y.points
                           from   Results Y
                           where  Y.category = 'homework'
                              and Y.number = 1)
```

Note: { `all`, `any`, `some` } do **not** extend SQL's expressiveness.

The statement

<u>*A* `< any`</u> `(select` <u>*B*</u> `from ... where ...)`

is equivalent to

<u>`exists`</u> `(select` *B* `from ... where ...` <u>`and` *A* `<` *B*</u>`)`

The statement *x* `in` *S* is equivalent to *x* `= any` *S*.

SQL :: Single Value Subqueries

Single Value Subqueries

If **none of the keywords** `all`, `any`, `some` are present, i.e.

```
. . . where x = (select A from . . . ) ,
```

the subquery must yield **single column and at most one row**.
So the comparison is between atomic values.

Single Value Subqueries

If **none of the keywords** `all`, `any`, `some` are present, i.e.

`. . . where x = (select A from . . . ) ,`

the subquery must yield **single column and at most one row**.
So the comparison is between atomic values.

```
Students(sid, first, last, address)
Exercises(category, number, topic, maxPoints)
Results(sid → Students, (category, number) → 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.category = 'homework' and R.number = 1
   and R.points = (select maxPoints
                   from   Exercises
                   where  category = 'homework' and number = 1)
```

Single Value Subqueries

If **none of the keywords** `all`, `any`, `some` are present, i.e.

```
. . . where x = (select A from . . . ) ,
```

the subquery must yield **single column and at most one row**.
So the comparison is between atomic values.

```
Students(sid, first, last, address)
Exercises(category, number, topic, maxPoints)
Results(sid → Students, (category, number) → 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.category = 'homework' and R.number = 1
   and R.points = (select maxPoints
                   from   Exercises
                   where  category = 'homework' and number = 1)
```

Why is this query guaranteed to return a single column & row?

Single Value Subqueries

Use constraints to ensure that the query returns only one row!

The DBMS will yield a runtime error if the subquery returns two or more rows.

Single Value Subqueries

Use constraints to ensure that the query returns only one row!

The DBMS will yield a runtime error if the subquery returns two or more rows.

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

Single Value Subqueries

Use constraints to ensure that the query returns only one row!

The DBMS will yield a runtime error if the subquery returns two or more rows.

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.category = 'homework'
           and R.number = 1) is null
```

SQL :: Views & Subqueries under From

Subqueries under From

Since an **SQL query returns a table**, it makes sense to use a subquery wherever a table might be specified.

SQL allows subqueries in the `from` clause.

Subqueries under From

Since an **SQL query returns a table**, it makes sense to use a subquery wherever a table might be specified.

SQL allows subqueries in the `from` clause.

Points (in %) achieved in homework exercise 1

```
select X.sid, (X.points * 100 / X.maxPoints) as percent
from   (select E.category, E.number, R.sid, R.points, E.maxPoints
        from   Exercises E, Results R
        where  E.category = R.category and E.number = R.number) X
where  X.category = 'homework' and X.number = 1
```

Note: join of `Results` and `Exercises` is computed in a subquery.

Subqueries under From

Since an **SQL query returns a table**, it makes sense to use a subquery wherever a table might be specified.

SQL allows subqueries in the `from` clause.

Points (in %) achieved in homework exercise 1

```
select X.sid, (X.points * 100 / X.maxPoints) as percent
from   (select E.category, E.number, R.sid, R.points, E.maxPoints
        from   Exercises E, Results R
        where  E.category = R.category and E.number = R.number) X
where  X.category = 'homework' and X.number = 1
```

Note: join of `Results` and `Exercises` is computed in a subquery.

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.**

Not allowed in SQL!

```
select S.first, S.last, X.number, X.points
from   Students S, (select R.number, R.points
                    from  Results R
                    where R.category = 'homework'
                      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 HomeworkPoints as
select S.first, S.last, R.number, R.points
from   Students S, Results R
where  S.sid = R.sid and R.category = 'homework'
```

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 HomeworkPoints as
select S.first, S.last, R.number, R.points
from   Students S, Results R
where  S.sid = R.sid and R.category = 'homework'
```

In queries, views may be used like stored tables:

```
select number, points
from   HomeworkPoints
where  first = 'George' and last = 'Orwell'
```

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

SQL :: Aggregation Functions

Aggregations

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

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

They take as input the values of an entire column.

Aggregations

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

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

They take as input the values of an entire column.

Aggregation functions are also known as

- **group functions**, or
- **column functions**

Typical use: statistics, data analysis, report generation.

Aggregations

SQL-92 defines the five main aggregation functions

`count`, `sum`, `avg`, `max`, `min`

Some DBMS define more functions:

`correlation`, `stddev`, `variance`, `...`

Aggregations

SQL-92 defines the five main aggregation functions

`count`, `sum`, `avg`, `max`, `min`

Some DBMS define more functions:

`correlation`, `stddev`, `variance`, `...`

How many Students in the current database state?

```
select count(*)
from   Students
```

count(*)
5

Aggregations

SQL-92 defines the five main aggregation functions

`count`, `sum`, `avg`, `max`, `min`

Some DBMS define more functions:

`correlation`, `stddev`, `variance`, `...`

How many Students in the current database state?

```
select count(*)
from   Students
```

count(*)
5

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.:

$\cdots$ `count(distinct A)` $\cdots$

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.*

Best and average result for homework 1?

```
select max(points), avg(points)
from   Results
where  category = 'homework' and number = 1
```

max(points)	avg(points)
10	8

Simple Aggregations

```
Students(sid, first, last, address)
Exercises(category, number, topic, maxPoints)
Results(sid → Students, (category, number) → Exercises, points)
```

How many Students have submitted a homework?

Simple Aggregations

```
Students(sid, first, last, address)
Exercises(category, number, topic, maxPoints)
Results(sid → Students, (category, number) → Exercises, points)
```

How many Students have submitted a homework?

```
select count(distinct sid)
from   Results
where  category = 'homework'
```

count(distinct sid)
3

Simple Aggregations

```
Students(sid, first, last, address)
Exercises(category, number, topic, maxPoints)
Results(sid → Students, (category, number) → Exercises, points)
```

How many Students have submitted a homework?

```
select count(distinct sid)
from   Results
where  category = 'homework'
```

count(distinct sid)
3

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

Simple Aggregations

Students(sid, first, last, address)
Exercises(category, number, topic, maxPoints)
Results(sid → Students, (category, number) → Exercises, points)

How many Students have submitted a homework?

```
select count(distinct sid)
from   Results
where  category = 'homework'
```

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 category = 'homework'
```

total points
18

Simple Aggregations

SQL also allows to write formulas

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

```
select avg(R.points / E.maxPoints) * 100
from   Results R, Exercises E
where  R.category = 'homework' and R.number  = 1
   and E.category = 'homework' and E.number = 1
```

Simple Aggregations

SQL also allows to write formulas

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

```
select avg(R.points / E.maxPoints) * 100
from   Results R, Exercises E
where  R.category = 'homework' and R.number  = 1
   and E.category = 'homework' and E.number = 1
```

Homework points for student 101 plus 3 bonus points.

Simple Aggregations

SQL also allows to write formulas

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

```
select avg(R.points / E.maxPoints) * 100
from   Results R, Exercises E
where  R.category = 'homework' and R.number  = 1
   and E.category = 'homework' and E.number = 1
```

Homework points for student 101 plus 3 bonus points.

```
select  sum(points) + 3 as "total homework points"
from    Results
where   sid = 101 and category = 'homework'
```

Restrictions

The following are not allowed:

- Simple aggregations may not be nested (makes no sense):

Wrong!

$\cdots$ `sum(avg(A))` $\cdots$

Restrictions

The following are not allowed:

- Simple aggregations may not be nested (makes no sense):

Wrong!

```
··· sum(avg(A)) ···
```

- Aggregations may not be used in the `where` clause:

Wrong!

```
··· where sum(A) > 100 ···
```

Restrictions

The following are not allowed:

- Simple aggregations may not be nested (makes no sense):

  Wrong!

  ```
  ··· sum(avg(A)) ···
  ```

- Aggregations may not be used in the `where` clause:

  Wrong!

  ```
  ··· where sum(A) > 100 ···
  ```

- If an aggregation function is used without `group by`, **no attributes** may appear in the `select` clause:

  Wrong!

  ```
  select  category, number, avg(points)
  from    Results
  ```

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`.

A bit counter-intuitive for `sum` as one might expect `0`.

However, allows to detect the difference between:

- all column values `null`, or
- values that sum up to 0.

SQL :: Aggregations with Group By and Having

Group By

"Group by" **partitions** the rows of a table into **disjoint groups**:

- based on **value equality for the `group by` attributes**.

Aggregation functions applied for each group separately.

Group By

"Group by" **partitions** the rows of a table into **disjoint groups**:

- based on **value equality for the `group by` attributes**.

Aggregation functions applied for each group separately.

Average points for <u>each</u> homework

```
select   number, avg(points)
from     Results
where    category = 'homework'
group by number
```

number	avg(points)
1	8
2	8.5

All tuples agreeing in their `number` values for a group:

sid	category	number	points
101	homework	1	10
102	homework	1	9
103	homework	1	5
101	homework	2	8
101	homework	2	9

Group By

The groups are formed **after** the evaluation of the `from` and `where` clauses. Aggregation is subsequently done for every group (yielding as many rows as groups).

The `group by` **never** produces empty groups.

The `group by` attributes may be used in the `select` clause since they have a **unique value for every group**.

- A reference to any other attribute is illegal.

Wrong!

```
select    E.number, E.topic, avg(R.points)
from      Exercises E, Results R
where     E.category = 'homework'
      and R.category = 'homework' and E.number = R.number
group by  E.number
```

Wrong, although `E.number` *functionally determines* `E.topic` *which thus is unique (for every group).*

Group By

Grouping by `E.number` **and** `E.topic` yields the desired result!

```
select    E.number, E.topic, avg(R.points)
from      Exercises E, Results R
where     E.category = 'homework'
      and R.category = 'homework' and E.number = R.number
group by  E.number, E.topic
```

E.number	E.topic	avg(points)
1	Rel.Alg.	8
2	SQL	8.5

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

Group By

Grouping by `E.number` **and** `E.topic` yields the desired result!

```
select    E.number, E.topic, avg(R.points)
from      Exercises E, Results R
where     E.category = 'homework'
      and R.category = 'homework' and E.number = R.number
group by  E.number, E.topic
```

E.number	**E.topic**	**avg(points)**
1	Rel.Alg.	8
2	SQL	8.5

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

The order of the `group by` attributes is not important.

Group By

Is there any semantical difference between these queries?

```
select   topic, avg(points / maxPoints)
from     Exercises E, Results R
where    E.category = 'homework' and R.category= 'homework'
     and E.number = R.number
group by topic

select   topic, avg(points / maxPoints)
from     Exercises E, Results R
where    E.category = 'homework' and R.category= 'homework'
     and E.number = R.number
group by topic, E.number
```

Having

Aggregations may not be used in the `where` clause.

With `group by`, however, it makes 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 makes sense to **filter out entire groups** based on some aggregated group property.

This is possible with SQL's `having` clause.

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
```

The condition in the `having` clause may (only) involve aggregation functions.

Having

```
Students(sid, first, last, address)
Exercises(category, number, topic, maxPoints)
Results(sid → Students, (category, number) → Exercises, points)
```

Which Students got at least 18 homework points?

Having

Students(<u>sid</u>, first, last, address)
Exercises(<u>category</u>, <u>number</u>, topic, maxPoints)
Results(<u>sid</u> → <u>Students</u>, (<u>category</u>, <u>number</u>) → <u>Exercises</u>, points)

Which Students got at least 18 homework points?

```
select     first, last
from       Students S, Results R
where      S.sid = R.sid and R.category = 'homework'
group by   S.sid, first, last
having     sum(points) >= 18
```

first	last
George	Orwell
Elvis	Presley

Having

```
Students(sid, first, last, address)
Exercises(category, number, topic, maxPoints)
Results(sid → Students, (category, number) → 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.category = 'homework'
group by  S.sid, first, last
having    sum(points) >= 18
```

first	last
George	Orwell
Elvis	Presley

- The `where` clause refers to single tuples.
- The `having` clause applies to entire groups.

Where vs. Having

The `having` clause 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
```

Where vs. Having

The `having` clause 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
```

SQL :: Aggregation Subqueries

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.category = 'homework' and R.number = 1
   and R.points = (select max(points)
                   from   Results
                   where  category = 'homework'
                      and number = 1)
```

Remember: earlier we solved this using `any`/`all`*.*

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

Aggregation Subqueries in Select

Aggregation subqueries may be used in the `select` clause.

This sometimes can be used to replace `group by`*.*

The homework points of the individual Students.

```
select first, last, (select sum(points)
                     from   Results R
                     where  R.sid = S.sid
                        and R.category = 'homework'
                    ) as homeworkPoints
from   Students S
```

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.homeworkPoints)
from     (select   sid, sum(points) as homeworkPoints
          from     Results
          where    category = 'homework'
          group by sid) X
```

X	
sid	**homeworkPoints**
101	18
102	18
103	5

avg(X.homeworkPoints)
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.category = 'homework'
group by S.sid, first, last
having   sum(points)
            >= all (select   sum(points)
                    from     Results
                    where    category = 'homework'
                    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.category = 'homework'
group by S.sid, first, last
having   sum(points)
            >= all (select   sum(points)
                    from     Results
                    where    category = 'homework'
                    group by sid)
```

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

Maximizing Aggregations

View: total number of homework points for each student.

```
create view HomeworkPoints as
select   sid, sum(points) as total
from     Results
where    category = 'homework'
group by sid
```

Alternative formulation of query on previous slide.

```
select  S.first, S.last, H.total
from    Students S, HomeworkPoints H
where   S.sid = H.sid
    and H.total = (select max(total)
                   from   HomeworkPoints)
```

SQL :: Union & Case & Coalesce

Union

"Union" allows to combine the results of two queries.

This is 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.

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

"Union" 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.

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

"Union" 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).

Union

Total number of homework points for every student

```
  select    S.first, S.last, sum(R.points) as total
  from      Students S, Results R
  where     S.sid = R.sid and R.category = 'homework'
  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  category = 'homework')
```

Union

Assigning 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.category = 'homework' and R.number = 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.category = 'homework' and R.number = 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.

*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` ($A - B$)
- `intersect` ($A \cap B$)

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

How?

Conditional Expressions

"Union" is the **portable way** to conduct a case analysis.

Sometimes a **conditional expression** suffices & more efficient.

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

Conditional Expressions

"Union" is the **portable way** to conduct a case analysis.

Sometimes a **conditional expression** suffices & more efficient.

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

Here, we will use the SQL-92 syntax.

Assigning student grades based on homework 1

```
select  S.sid, case when points >= 9 then 'A'
                     when points >=7 and points < 9 then 'B'
                     when points >=5 and points < 7 then 'C'
                  else 'F' end as 'grade'
from    Students S, Results R
where   S.sid = R.sid
    and R.category = 'homework' and R.number = 1
```

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)···
```

List the addresses of all students

```
select  first, last, coalesce(address, '(unknown)')
from    Students
```

SQL :: Order By

Sorting Output

If query output is to be read by humans, enforcing a certain **tuple order** 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** 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

If query output is to be read by humans, enforcing a certain **tuple order** 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.

```
order by attribute1 [asc|desc], attribute2 [asc|desc], . . .
```

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

If query output is to be read by humans, enforcing a certain **tuple order** 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.

```
order by attribute1 [asc|desc], attribute2 [asc|desc], . . .
```

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

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

```
select   R.number, R.points, S.first, S.last
from     Students S, Results R
where    S.sid = R.sid and R.category = 'homework'
order by R.number, R.points desc, S.last, S.first
```

Sorting Output

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

```
select   R.number, R.points, S.first, S.last
from     Students S, Results R
where    S.sid = R.sid and R.category = 'homework'
order by R.number, R.points desc, S.last, S.first
```

- First, compare `R.number`.

Sorting Output

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

```
select   R.number, R.points, S.first, S.last
from     Students S, Results R
where    S.sid = R.sid and R.category = 'homework'
order by R.number, R.points desc, S.last, S.first
```

- First, compare `R.number`.
- If the first criterion leads to a tie, compare `points` **desc**.

Sorting Output

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

```
select   R.number, R.points, S.first, S.last
from     Students S, Results R
where    S.sid = R.sid and R.category = 'homework'
order by R.number, R.points desc, S.last, S.first
```

- First, compare `R.number`.
- If the first criterion leads to a tie, compare `points` **desc**.
- If we still have a tie, compare `S.last`.

Sorting Output

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

```
select   R.number, R.points, S.first, S.last
from     Students S, Results R
where    S.sid = R.sid and R.category = 'homework'
order by R.number, R.points desc, S.last, S.first
```

- First, compare `R.number`.
- 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

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

```
select   R.number, R.points, S.first, S.last
from     Students S, Results R
where    S.sid = R.sid and R.category = 'homework'
order by R.number, R.points desc, S.last, S.first
```

- First, compare `R.number`.
- 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`.

number	points	first	last
1	10	George	Orwell
1	9	Elvis	Presley
1	5	Lisa	Simpson
2	9	Elvis	Presley
2	8	George	Orwell

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 `'George␣Orwell'`, 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 `'George␣Orwell'`, 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 sorted 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 `'George␣Orwell'`, 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 sorted sequence (depending on the database).

Since the effect of `order by` is purely "cosmetic", `order by` may **not** be applied to a subquery.

SQL :: Left & Right Outer and Inner Joins

Joins

Up to version SQL-86, there were no explicit joins in queries.
Instead, Cartesian products of relations filtered via `where`.

```
select  R.category, R.number, sid, points, topic, maxPoints
from    Results R, Exercises E
where   R.category = E.category and R.number = E.number
```

Joins

Up to version SQL-86, there were no explicit joins in queries.
Instead, Cartesian products of relations filtered via `where`.

```
select  R.category, R.number, sid, points, topic, maxPoints
from    Results R, Exercises E
where   R.category = E.category and R.number = E.number
```

Since SQL-92 there are explicit join operations.

Natural Joins

“Natural join” in SQL-92

```
select  sid, number, (points / maxPoints) * 100
from    Results natural join Exercises
where   category = 'homework'
```

Note the use of **natural join**!

Natural Joins

“Natural join” in SQL-92

```
select  sid, number, (points / maxPoints) * 100
from    Results natural join Exercises
where   category = 'homework'
```

Note the use of **natural join**!

DBMS to automatically add the join predicate to the query:

```
Results.category = Exercises.category
and Results.number = Exercises.number
```

Natural Joins

"Natural join" in SQL-92

```
select  sid, number, (points / maxPoints) * 100
from    Results natural join Exercises
where   category = 'homework'
```

Note the use of **natural join**!

DBMS to automatically add the join predicate to the query:

```
Results.category = Exercises.category
and Results.number = Exercises.number
```

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

Specifying the Join Predicate

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

- **`natural`** prepended to join operator name.

  ```
  Results natural join Exercises
  ```

  Yields comparison of columns with the same name.

- **`using (A1, ..., An)`** after the second table.

  ```
  Results join Exercises using (category,number)
  ```

  The A_i must be columns appearing in both tables. The join predicate then is `R.A1 = S.A1 and ...and R.An = S.An`.

- **`on (condition)`** after the second table.

  ```
  Students S join Results R on (S.sid = R.sid)
  ```

  The matching condition works similar to the where clause, but is important in combination with left/right joins.

The `cross join` operator (next slide) has no join predicate.

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 **join (⋈) eliminates tuples without partner.**

A	B
a_1	b_1
a_2	b_2

⋈

B	C
b_2	c_2
b_3	c_3

=

A	B	C
a_2	b_2	c_2

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 **join (⋈) eliminates tuples without partner.**

A	B
a_1	b_1
a_2	b_2

$\bowtie$

B	C
b_2	c_2
b_3	c_3

$=$

A	B	C
a_2	b_2	c_2

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

A	B
a_1	b_1
a_2	b_2

$⟕$

B	C
b_2	c_2
b_3	c_3

$=$

A	B	C
a_1	b_1	(null)
a_2	b_2	c_2

Inner and Outer Joins

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

A	B
a_1	b_1
a_2	b_2

$⋈|$

B	C
b_2	c_2
b_3	c_3

$=$

A	B	C
a_2	b_2	c_2
(null)	b_3	c_3

Inner and Outer Joins

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

A	B
a_1	b_1
a_2	b_2

$\bowtie|$

B	C
b_2	c_2
b_3	c_3

$=$

A	B	C
a_2	b_2	c_2
(null)	b_3	c_3

The **full outer join** preserves all tuples in **both** arguments:

A	B
a_1	b_1
a_2	b_2

$|\bowtie|$

B	C
b_2	c_2
b_3	c_3

$=$

A	B	C
a_1	b_1	(null)
a_2	b_2	c_2
(null)	b_3	c_3

Inner and Outer Joins

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

A	B
a_1	b_1
a_2	b_2

$\bowtie|$

B	C
b_2	c_2
b_3	c_3

$=$

A	B	C
a_2	b_2	c_2
(null)	b_3	c_3

The **full outer join** preserves all tuples in **both** arguments:

A	B
a_1	b_1
a_2	b_2

$|\bowtie|$

B	C
b_2	c_2
b_3	c_3

$=$

A	B	C
a_1	b_1	(null)
a_2	b_2	c_2
(null)	b_3	c_3

The **cross join** is the **Cartesian product**:

A	B
a_1	b_1
a_2	b_2

$\times$

B	C
b_2	c_2
b_3	c_3

$=$

A	B	B	C
a_1	b_1	b_2	c_2
a_1	b_1	b_3	c_3
a_2	b_2	b_2	c_2
a_2	b_2	b_3	c_3

Inner and Outer Joins

```
Students(sid, first, last, address)
Exercises(category, number, topic, maxPoints)
Results(sid → Students, (category, number) → Exercises, points)
```

Number of submission per homework (0 if no submission)

```
select   E.number, count(sid)
from     Exercises E left outer join Results R
      on E.category = R.category and E.number = R.number
where    E.category = 'homework'
group by E.number
```

Inner and Outer Joins

```
Students(sid, first, last, address)
Exercises(category, number, topic, maxPoints)
Results(sid → Students, (category, number) → Exercises, points)
```

Number of submission per homework (0 if no submission)

```
select   E.number, count(sid)
from     Exercises E left outer join Results R
      on E.category = R.category and E.number = R.number
where    E.category = 'homework'
group by E.number
```

All Exercises are present in the result of the left (outer) join.

- for exercises without solutions, `sid` and `points` will be `null`
- `count(sid)` ignores rows where `sid is null`.

Inner and Outer Joins

```
Students(sid, first, last, address)
Exercises(category, number, topic, maxPoints)
Results(sid → Students, (category, number) → Exercises, points)
```

Number of submission per homework (0 if no submission)

```
select   E.number, count(sid)
from     Exercises E left outer join Results R
      on E.category = R.category and E.number = R.number
where    E.category = 'homework'
group by E.number
```

All Exercises are present in the result of the left (outer) join.

- for exercises without solutions, `sid` and `points` will be `null`
- `count(sid)` ignores rows where `sid is null`.

Could also be solved using **union**, but less elegant (longer).

Inner and Outer Joins

```
Students(sid, first, last, address)
Exercises(category, number, topic, maxPoints)
Results(sid → Students, (category, number) → Exercises, points)
```

Exercises with corresponding submissions in different ways...

Join with on

```
select *
from Exercises E left outer join Results R
     on E.category = R.category and E.number = R.number
```

Join with using

```
select *
from Exercises E left outer join Results R
     using (category, number)
```

Join with natural

```
select *
from Exercises E natural left outer join Results R
```

Inner and Outer Joins

Is there a problem with the following query?

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

```
select   first, last, count(number)
from     Students S left outer join Results R
         on S.sid = R.sid
where    R.category = 'homework'
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(number)
from     Students S left outer join Results R
         on S.sid = R.sid
where    R.category = 'homework'
group by S.sid, first, last
```

Correction:

- restrict the join inputs **before** the outer join is performed, or
- **move** restrictions into the on clause (warning: next slide).

Corrected version of last query

```
select   first, last, count(number)
from     Students S left outer join Results R
         on (S.sid = R.sid and R.category = 'homework')
group by S.sid, first, last
```

Inner and Outer Joins

Will exams appear in the output?

```
select E.category, E.number, R.sid, R.points
from   Exercises E left outer join Results R
       on    E.category = 'homework'
         and R.category = 'homework'
         and E.number = R.number
```

Inner and Outer Joins

Will exams appear in the output?

```
select E.category, E.number, R.sid, R.points
from   Exercises E left outer join Results R
       on    E.category = 'homework'
         and R.category = 'homework'
         and E.number = R.number
```

Yes, exams will appear!

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.

Inner and Outer Joins

Will exams appear in the output?

```
select E.category, E.number, R.sid, R.points
from   Exercises E left outer join Results R
       on    E.category = 'homework'
         and R.category = 'homework'
         and E.number = R.number
```

Yes, exams will appear!

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.

Corrected version of last query

```
select E.category, E.number, R.sid, R.points
from  (select * from Exercises where category = 'homework') E
      left outer join Results R
      on (R.category = 'homework' and E.number = R.number)
```