SQL

- After completing this chapter, you should be able to
  - ▷ write advanced SQL queries including, *e.g.*, multiple **tuple variables** over different/the same relation,
  - ▷ use **aggregation**, **grouping**, `UNION`,
  - ▷ be comfortable with the various **join variants,**
  - ▷ evaluate the **correctness** and **equivalence** of SQL queries,

    This includes the sometimes tricky issues of (deciding the presence) duplicate result tuples.
  - ▷ judge the **portability** of certain SQL constructs.

SQL

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  A1 , ... , An
FROM    R1 , ... , Rm
WHERE   C
```

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

Example Database (again)

STUDENTS

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

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

Tuple Variables (1)

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

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

▷ This may be executed as

```
foreach E ∈ EXERCISES do
  if E.CAT = 'H' then
    print E.ENO, E.TOPIC
  fi
od
```

▷ **Tuple variable** `E` represents a **single row** of table `EXERCISES` (the loop assigns each row in succession).

Tuple Variables (2)

- For each table in the `FROM` clause, a tuple variable is automatically created: if 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, stating `FROM EXERCISES` only is understood as

`FROM EXERCISES EXERCISES`

(the tuple variable `EXERCISES` ranges over the rows of the table `EXERCISES`).

Tuple Variables (3)

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

  `FROM EXERCISES E`

  then the implicit tuple variable `EXERCISES` is *not* declared, *i.e.*, `EXERCISES.ENO` will yield an error.

- A reference to an attribute A of a tuple variable R may be simply written as A (instead of $R.A$) if R is **the only variable** which is bound to tuples having an attribute named A.

Attribute References (1)

- In general, **attributes are referenced** in the form

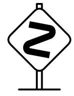
$$R.A$$

- If an attribute may be associated to a tuple variable in an unambiguous manner, the variable may be omitted, *e.g.*:

```
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 (2)



- 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'
```

▷ 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 in the SELECT clause.

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

Joins (1)

- Consider a query with two tuple variables:

```
SELECT  A1, ..., An
FROM    STUDENTS S, RESULTS R
WHERE   C
```

▷ S will range over 4 tuples in STUDENTS, R will range over 8 tuples in RESULTS. In principle, all $4 \times 8 = 32$ combinations will be considered in condition C:

```
foreach S ∈ STUDENTS do
  foreach R ∈ RESULTS do
    if C then
      print A1, ..., An
    fi
  od
od
```

Joins (2)

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

  This is the task of the **query optimizer.** For example, if C contains the join condition `S.SID = R.SID`, the DBMS might loop over the tuples in `RESULTS` and find the corresponding `STUDENTS` tuple by using an **index** over `STUDENT.SID` (many DBMS automatically create an index over the key attributes).

- In order to understand the **semantics** of a query, however, the simple nested `foreach` algorithm suffices.

  The query optimizer may use any algorithm that produces the **exact same output**, although possibly in different tuple order.

Joins (3)

- 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'
```

Output of this query?

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

Notes

Joins (4)

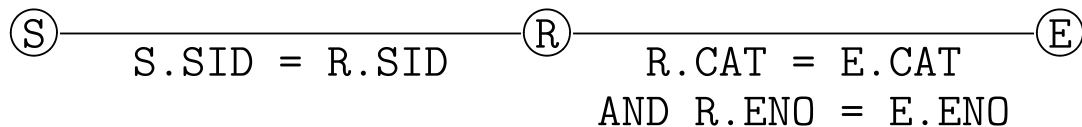
- Guideline: it is almost always an **error** if there are two tuples variables which are **not linked** (directly or indirectly) via join conditions.

  ▷ In this query, all three tuple variables are connected:

```
SELECT 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 (5)

- The tuple variable connection works as follows:



- The conditions correspond to the **foreign key relationships** between the tables.
- The omission of a join condition will usually lead to numerous duplicates in the query result.

  Usage of `DISTINCT` to get rid of duplicate does *not* fix the error in such a case, of course.

Query Formulation (1)

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, requiring a tuple variable, `S` say, over `STUDENTS` and the identifying condition `S.FIRST = 'Ann' AND S.LAST = 'Smith'`.
  - ▷ Exercise topics are of interest, so a tuple variable `E` over `EXERCISES` is needed, and the following piece of SQL can already be generated:

```
SELECT DISTINCT E.TOPIC
```

Several exercises may have the same topic.

Query Formulation (2)

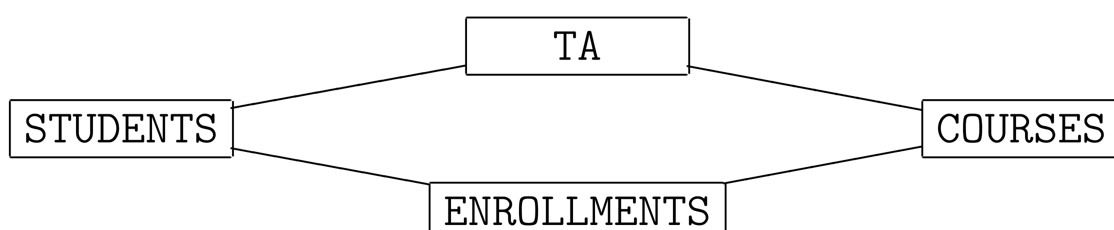
- Note: `S` and `E` are still **unconnected.**
- The **connection graph** of the tables in a database schema (edges correspond to foreign key relationships) helps in understanding the connection requirements:

▷ 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 (3)

- 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 TA ("*Hiwi*") assignments:



Unnecessary Joins (1)

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

  Query will run slowly if the optimizer overlooks the redundancy.

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 (2)

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
```

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
```

Notes

Self Joins (1)

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

Is there a student who got 9 points for both, homework 1 and homework 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 (2)

Find students who solved at least two exercises.[a]

[a] This may also be solved using aggregations (later).

```
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 make explicit that E1 and E2 refer to distinct exercises:

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


Duplicate Elimination (1)

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

Which exercises have already been solved by at least one student?

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

CAT	ENO
H	1
H	2
M	1
H	1
H	2
M	1
H	1
M	1

Duplicate Elimination (2)

- If a query might yield unwanted duplicate tuples, 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

- To emphasize that there will be duplicate rows (and that these are wanted in the result), SQL provides the `ALL` modifier.[11]

[11] `SELECT ALL` is the default.

Duplicate Elimination (3)

- **Sufficient condition for superfluous** `DISTINCT`:

① Let $\mathcal{K}$ be the set of attributes selected for output by the `SELECT` clause.

② Add to $\mathcal{K}$ attributes A such that $A = c$ (constant c) appears in the `WHERE` clause.

Here we assume that the `WHERE` clause specifies a conjunctive condition.

③ Add to $\mathcal{K}$ attributes A such that $A = B$ $(B \in \mathcal{K})$ appears in the `WHERE` clause. If $\mathcal{K}$ contains a key of a tuple variable, add all attributes of that variable.

Repeat ③ until $\mathcal{K}$ stable.

④ If $\mathcal{K}$ **contains a key of every tuple variable** listed under `FROM`, then `DISTINCT` is superfluous.

Duplicate Elimination (5)

```
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 declared as an alternative key for STUDENTS.

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

② $\mathcal{K} \leftarrow \mathcal{K} \cup$ {R.CAT} because of the conjunct R.CAT = 'H'.

continued ...

Duplicate Elimination (6)

... continued from last slide

③ $\mathcal{K} \leftarrow \mathcal{K} \cup$ {S.SID, S.EMAIL} because $\mathcal{K}$ contains a key of STUDENTS (S.FIRST, S.LAST).

③ $\mathcal{K} \leftarrow \mathcal{K} \cup$ {R.SID} because of the conjunct S.SID = R.SID.

④ $\mathcal{K}$ contains a key of STUDENTS (see above) and RESULTS (R.SID, R.CAT, R.ENO), thus DISTINCT is superfluous.

- Note: if FIRST, LAST were no key of STUDENTS, this test would not succeed (and rightly so).

Summary: 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.

SQL

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 (1)

- SQL queries using only the constructs introduced above 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 (remember: "*Find students who have not yet 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.
- Obviously, such queries *cannot* be formulated with the SQL constructs introduced so far.

Non-Monotonic Behaviour (2)

- Note: in natural lanugage, queries which contain formulations like "*there is no*", "*does not exists*", *etc.*, indicate non-monotonic behaviour (**existential quantification**).
- Furthermore, "*for all*", "*the minimum/maximum*" also indicate non-monotonic behaviour: in this case, a violation of a **universally quantified** condition must not exist.
- In an equivalent SQL formualtion of such queries, this ultimately leads to a test whether a certain **query yields a (non-)empty result.**

NOT IN (1)

- 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 (2)

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

STUDENTS

SID	FIRST	LAST	EMAIL
101	Ann	Smith	...
101	Michael	Jones	(null)
101	Richard	Turner	...
101	Turner	Brown	...

Subquery result

SID
101
101
102
102
103

- Then, for every STUDENTS tuple, a matching SID is searched for in the subquery result. If there is none (NOT IN), the tuple is output.

NOT IN (3)

- Since the **(non-)existence of particular tuples** does *not* depend on multiplicity, we may equivalently use DISTINCT in the subquery:

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

- The effect on the performance depends on the DBMS and the data (sizes).

  A reasonable optimizer will know about the NOT IN semantics and will decide on duplicate elimination/preservation itself, esp. because IN (NOT IN) may efficiently be implemented via **semijoin** and **antijoin** if duplicates are eliminated.

NOT IN (4)

Topics of homeworks that were 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'
```

468

NOT IN (5)

- In SQL-86, the subquery is required to deliver a **single output column.**

  This ensures that the subquery is a set (or multiset) and not an arbitrary relation.

- In SQL-92, comparisons where extended to the tuple level.[12] It is thus valid to write, *e.g.*:

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

[12] However, also see EXISTS below.

469

NOT EXISTS (1)

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

- 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 but this yields unnecessarily complicated query formulations (bad style).

- In this case, the outer query and subquery are **correlated.** In principle, the **subquery** has to be evaluated for every assignment of values to the outer tuple variables. (The subquery is "*parameterized*".)

NOT EXISTS (2)

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).

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

NOT EXISTS (3)

- "First," S is bound to the STUDENTS tuple

SID	FIRST	LAST	EMAIL
101	Ann	Smith	...

- In the subquery, S.SID is "replaced by" 101 and the following query is executed:

```
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 (4)

- "Finally," S is bound to the STUDENTS tuple

SID	FIRST	LAST	EMAIL
104	Maria	Brown	...

- In the subquery, S.SID is "replaced by" 104 and the following query is executed:

```
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 (5)

- 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 (6)

- **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 relation **constant** and may make perfect sense (*e.g.*, when used with (NOT) IN).

NOT EXISTS (7)

- Note: it is legal SQL syntax to specify an arbitrarily complex SELECT clause in the subquery, however, this does not affect the existential semantics of NOT EXISTS.

  SELECT * ... documents this quite nicely. Some SQL developers prefer SELECT 42 ... or SELECT null ... or similar SQL code.

  Again, the query optimizer will know the NOT EXISTS semantics such that the exact choice SELECT clause is of no importance.

NOT EXISTS (8)

- 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?

Notes

"For all" (1)

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" (2)

- Mathematical logic knows two **quantifiers:**
  - ▷ $\exists X(\varphi)$ **existential quantifier**

    There is an X that satfisfies formula φ.
  - ▷ $\forall X(\varphi)$ **universal quantifier**

    For all X, formula φ is satisfied (true).
- In **tuple relational calculus (TRC)** the maximum number of points for homework 1 reads

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

"For all" (3)

- For database queries, the pattern $\forall X\,(\varphi_1 \Rightarrow \varphi_2)$ is very typical.

  Such a **bounded quantifier** (X is required to fulfill φ_1) is natural because TRC requires that all queries are **safe:** values *outside* the DB state do *not* influence the truth values. This is important to ensure that the truth value of a formula for some DB state may be determined with *finite effort* (it suffices to take into account the values that actually occur in database relations).

  Usually, formula φ_1 binds X to (some subset of) the tuples of a database relation. For these tuples φ_2 then checks further query conditions.

"For all" (4)

- SQL does *not* offer a universal quantifier (but only the existential quantifier `EXISTS`).

  However, see `>= ALL` below.

- Of course, this is no problem because

$$\forall X\,(\varphi) \Leftrightarrow \neg\exists X\,(\neg\varphi)\ .$$

- The above query pattern thus becomes (SQL lacks $\Rightarrow$)

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

“For all” (5)

- The last example above is thus logically equivalent to

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

- In SQL:

```
SELECT  X.POINTS
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.POINTS > X.POINTS)
```


Nested Subqueries

- **Subqueries may be nested** to any reasonable depth.

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'))
```

Common Errors (1)

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 (2)

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

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)
```

Common Errors (3)

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.

Notes

ALL, ANY, SOME (1)

- 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 (2)

- The following is equivalent to the above 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, since, *e.g.*

$$A \texttt{ < ANY (SELECT } B \texttt{ FROM } \cdots \texttt{ WHERE } \cdots \texttt{)}$$

$$\equiv$$

$$\texttt{EXISTS (SELECT 1 FROM } \cdots \texttt{ WHERE } \cdots \texttt{ AND } A < B\texttt{)}$$

ALL, ANY, SOME (3)

- Syntactical remarks on comparisons with subquery results:

  ① ANY and SOME are synonyms.

  ② x IN S is equivalent to x = 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 (1)

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 (non-data dependent) constraints to ensure this condition, the DBMS will yield a runtime error if the subquery returns two or more rows.

Single Value Subqueries (2)

- 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` (1)

- Since the result of an SQL query is a **table**, it seems most natural to use a subquery result whereever a table might be specified, *i.e.*, in the `FROM` clause.
- This principle of (query) language construction is known as **orthogonality**: language constructs may be combined in arbitrary fashion as long as the semantic/typing/...rules of the language are obeyed.

  Relational algebra is an orthogonal query language.

- SQL version up to SQL-86 were not orthogonal in this sense.

Subqueries under `FROM` (2)

- One use of subqueries under `FROM` are **nested aggregations** (see further below).
- In the following example, the join of `RESULTS` and `EXERCISES` is computed in a subquery (this might result from a **view definition**, see below).

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` (3)

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

  This is for historic reasons and could be considered an SQL "bug."

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 (4)

- A **view declaration** registers a query[13] 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.**

[13] Not a query result!

SQL

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 (1)

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

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

- Aggregation functions are used to summarize an entire set of values.

  In the DB literature, 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 (2)

- SQL-92 defines the five main aggregation functions
  `COUNT`, `SUM`, `AVG`, `MAX`, `MIN` .
  - ▷ Some DBMS define further aggregation functions:

    `CORRELATION`, `STDDEV`, `VARIANCE`, `FIRST`, `LAST`, ...

- Any **commutative** and **associative** binary operator with a neutral element can be extended ("*lifted*") to work on set-valued arguments (*e.g.*, `SUM` corresponds to +).

Commutative and associative, neutral element?

Why do we require these properties of the operators?

Notes

Aggregations (3)

- Note: some aggegration functions are sensitive to **duplicates** (*e.g.*, `SUM`, `COUNT`, `AVG`), some are insensitive (*e.g.*, `MIN`, `MAX`).
- For the first type, SQL allows to explicitly request to ignore duplicates, *e.g.*:

```
··· COUNT(DISTINCT A) ···
```

Simple Aggregations (1)

- **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 (2)

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 (3)

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'
```


Aggregation Queries

- Basically, there **three different types of queries** in SQL:
  1. Queries without aggregation functions and without `GROUP BY` and `HAVING`. (Discussed above.)
  2. Queries with aggregation functions in the `SELECT` clause but no `GROUP BY` (simple aggregations). Yield exactly one row.
  3. Queries with `GROUP BY`.
- Each type has different syntax restrictions and is **evaluated** in a different way.

  Again: we refer to the SQL semantics here. A DBMS is free to implement these semantics as it sees fit.

Evaluation (1)

① First, evaluate the FROM clause.

Conceptually, form all possible tuple combinations of the source tables (Cartesian product).

② Evaluate the WHERE clause.

The Cartesian product produced in ① is filtered (restricted) and only those tuple combinations saisfying the filter condition remain.

③ No aggregation, GROUP BY or HAVING: evaluate the SELECT clause.

Evaluate projection list (terms, scalar expressions) for each tuple combination produced in ② and print resulting tuples.

Evaluation (2)

③* Simpe aggregation: add column values received from phase ② to sets/multisets that will be the input to the aggregation function(s).

- If no DISTINCT is used or if the aggregation function is **idempotent** (MIN, MAX), the aggregation results may be **incrementally** computed, *no* temporary sets need to be maintained (see next slide).
- Print the single row of aggregated value(s).

Evaluation (3)

Simple aggregation, no DISTINCT

```
SELECT   SUM(MAXPT), COUNT(*)
FROM     EXERCISES E
WHERE    CAT = 'H'
```

Possible eval strategy (no intermediate storage required)

```
agg1 ← 0;        /* neutral element for + */
agg2 ← 0;        /* neutral element for + */
foreach E ∈ EXERCISES do
  if E.CAT = 'H' then
    agg1 ← agg1 + E.MAXPT;        /* incrementally maintain SUM */
    agg2 ← agg1 + 1;              /* incrementally maintain COUNT */
  fi
od
print agg1, agg2
```


Restrictions (1)

- 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
```

But see GROUP BY below.

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 aggregation 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, this way a query can detect the difference between two types of empty input: (1) all column values NULL, or (2) no tuple qualified in WHERE clause.

SQL

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 (1)

- 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 (2)

- The GROUP BY clause partitions the incoming tuples (after evaluation of the FROM and WHERE clauses) into groups based on **value equality for the** GROUP BY **attributes:**

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 (3)

- This construction can *never* produce empty groups (a COUNT(*) will never result in 0).

  See outer joins for queries that deal with "empty groups."

- 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![a]

[a] Although E.ENO is key and thus E.TOPIC would be unique.

```
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
```


GROUP BY (4)

- Grouping by E.ENO *and* E.TOPIC is possible and will yield 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

The DBMS now has a simple **syntactic clue** that the value of E.TOPIC will be unique.

GROUP BY (5)

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
```

Notes

GROUP BY (6)

- The sequence of the `GROUP BY` attributes is not important.
- Grouping makes no sense if the `GROUP BY` attributes contain a **key** (if only one table is listed in the `FROM` clause): **each group will contain a single row** only.
- Duplicates should be eliminated with `DISTINCT`, although such elimination can also be realized 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 (1)

- Remember: 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 may be significant.

- This is possible with SQL's `HAVING` clause.

▷ The condition in the `HAVING` clause may (only) involve aggregation functions and the `GROUP BY` attributes.

HAVING (2)

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

- If a condition refers to GROUP BY attributes only (but not aggregations), it may be placed under WHERE *or* HAVING.

How many groups are produced for these two queries?

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

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

Notes

Aggregation Subqueries (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.
- Remember: our earlier solution to this problem was using `ANY/ALL`.

Aggregation Subqueries (2)

- In SQL-92, aggregation subqueries may be placed into the SELECT clause. This may 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.CAT = 'H')
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.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
103	18
103	5

AVG(X.HW_POINTS)
13.67

Maximizing Aggregations (1)

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 (2)

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)
```

SQL

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 (1)

- In SQL, it is possible to combine the results of two queries by UNION.
- UNION is strictly needed since there is no other method to construct one result column that draws from different tables/columns.

  This is necsessay, for example, if specializations of a concept ("subclasses") are stored in separate tables. For instance, there may be GRADUATE_COURSES and UNDERGRADUATE_COURSES tables (both of which are specializations of the general concept COURSE).

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

UNION (2)

- 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 (IBM DB2, for example, creates artifical column names 1, 2, . . . , if necessary. Use column renaming via AS if column names matter.

- SQL distinguishes between
  - ▷ UNION: like RA $\cup$ with **duplicate elimination**, and
  - ▷ UNION ALL: **concatenation** (duplicates retained).
- Other SQL-92 set operations: EXCEPT ($-$), INTERSECT ($\cap$).[14]

[14] These do *not* add to the epxressivity of SQL. Proof?

UNION (3)

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')
```

UNION (4)

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
  ...
```


Conditional Expressions (1)

- Whereas UNION is the **portable way** to conduct a case analysis, sometimes a **conditional expression** suffices and is more efficient.
  - ▷ Here, we will use the SQL-92 (and, *e.g.*, DB2) syntax. Conditional expression syntax varies between DBMSs. Oracle uses DECODE(···), for example.

Conditional Expressions (2)

Print the 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 (3)

- A typical application of a conditional expression is to **replace a null value** by another (non-null) 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
```

- Conditional expressions are regular terms, so they may be input for other functions, comparisons, or aggregate functions.

SQL

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 (1)

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

  Without such an ordering, the sequence of output rows is meaningless, depends on the internal algorithms selected by the query optimizer to evaluate the query and may change from version to version of even query to query.

- In a SQL RDBMS, however, the query logic and the subsequent output formatting are completely **independent** processes.

  DBMS front-ends offer a variety of formatting options (page breaks, colorization of column values, *etc.*).

Sorting Output (2)

- Specify a **list of sorting criteria** in an `ORDER BY` clause.

  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**).

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 (3)

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

- The second sort criterion (`POINTS DESC`) is used, when the first criterion (`ENO`) leads to a tie.
  - ▷ Sort in **ascending** order (default): `ASC`,
  - ▷ sort in **descending** order: `DESC`.

Sorting Output (4)

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

  Examples:

  ▷ Print homework results in the order homeworks (`CAT = 'H'`), midterm exam (`'M'`), and final exam (`'F'`).

  ▷ In a list of universities, `'TU Clausthal'` should be listed under `C`, not `T`.

  ▷ If the students names were stored in the form `'Ann␣Smith'`, sorting by last name is more or less impossible.

  This is related to an important **DB design time question:** "*What do I need to do with the query outputs?*"

Sorting Output (5)

- **Null values** are all listed first or all listed last in the sort sequence (IBM DB2: all first).
- Since the effect of `ORDER BY` is purely "cosmetic", `ORDER BY` may *not* be applied to a subquery.

  ▷ This also applies if multiple queries are combined via `UNION`. Place `ORDER BY` at the bottom of the query to sort all tuples.

SQL

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 (recap)

STUDENTS

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

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

Joins in SQL-92 (1)

- Up to version SQL-86, SQL users were not able to explicitly specify joins in queries. Instead, Cartesian products of relations (`FROM`) are specified and then filtered via `WHERE`.

  The query optimizer automatically derives the opportunity to deploy one of the several **join variants** (**declarative query language**).

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 in SQL-92 (2)

- In SQL-92, users may write, *e.g.*,

“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

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

  to the query.
- SQL-92 permits joins in the `FROM` clause as well as at the outermost query level (like `UNION`). This comes close to RA.

Outer Join (recap)

- Join ($\bowtie$) **eliminates tuples without partner:**

A	B
a_1	b_1
a_1	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, even if a tuple does not team up with a partner in the join:

A	B
a_1	b_1
a_1	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

Outer Join in SQL-92 (1)

Number of submission per homework (if no submission, return 0**).**

```
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
```

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

Outer Join in SQL-92 (2)

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')
```


Outer Join in SQL-92 (3)

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
```

Notes

Outer Join in SQL-92 (4)

- It is generally wise to restrict the outer join inputs **before** then outer join is performed.[15]

Corrected version of last query.

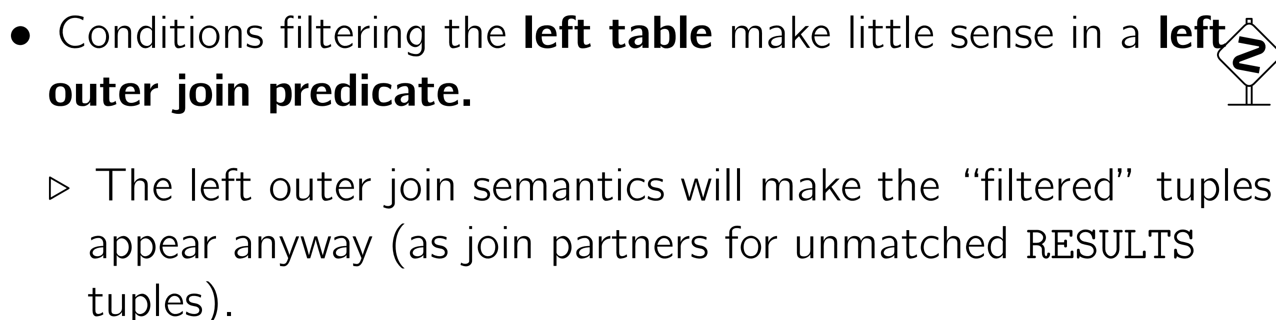
```
SELECT    FIRST, LAST, COUNT(ENO)
FROM      STUDENTS S LEFT OUTER JOIN
          (SELECT  SID, ENO
             FROM  RESULTS
            WHERE  CAT = 'H') R
          ON S.SID = R.SID
GROUP BY  S.SID, FIRST, LAST
```

[15] Or move restrictions into the ON clause.

Outer Join in SQL-92 (5)

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 semantics will make the "filtered" tuples appear anyway (as join partners for unmatched `RESULTS` tuples).

Outer Join SQL-92 (6)

- MySQL (prior to V4.1), for example, does not support subqueries. Sometimes, outer join can make up for this.

Students who did not submit any homework.

```
SELECT  S.SID, S.FIRST, S.LAST
FROM    STUDENTS S LEFT OUTER JOIN RESULTS R
        ON S.SID = R.SID AND R.CAT = 'H'
WHERE   R.SID IS NULL
```

- ▷ Instead of `R.SID` any attribute of `RESULTS` which may not be null (*e.g.*, a part of a key) could be tested for being `NULL`.

  The `IS NULL` test finds those `STUDENTS` tuples which do not find a matching join partner.

Join Syntax in SQL-92 (1)

- SQL-92 supports the following **join types** (parts in [ ] 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.
  - ▷ UNION JOIN: Pads columns of both tables with NULL.[16]
- Note: **semi-join** is easily expressed in the SELECT clause (SELECT DISTINCT *R*.* FROM *R*,*S* WHERE *C*).

[16] *SQL-92 Intermediate Level*: rarely found implemented in DBMSs.

Join Syntax in SQL-92 (2)

- The **join predicate** may be specified as follows:
  - ▷ Keyword NATURAL prepended to join operator name.
  - ▷ ON ⟨*Condition*⟩ appended to join operator name.
  - ▷ USING (A_1, . . . ,A_n) appended to join operator name.

    USING specified columns A_i appearing in both join inputs R and S. The effective join predicate then is $R.A_1 = S.A_1$ AND $\cdots$ AND $R.A_n = S.A_n$.
- CROSS JOIN and UNION JOIN have no join predicate.

UNION JOIN not implemented in today's DBMS products.

Simulate *R* UNION JOIN *S*.

Notes