SQL Queries on a Single Table (I)

- Query 1: Determine the names of all professors.

  select name **from** professors;

name
Sokrates
Russel
Kopernikus
Popper
Augustinus
Curie
Kant

- The result table is temporary
- The result table gets an internal name
- The base table *professors* remains unchanged
- Needed values are copied from the base table into the result table in the same order
- Each SQL command is terminated by a ';' so that it can extend over several lines

SQL Queries on a Single Table (II)

- Query 2: Determine the different ranks of professors.

  select distinct rank **from** professors;

- Elimination of duplicates in a table is not automatically executed in SQL queries for two main reasons:
  - SQL has an underlying list model; lists may have duplicate elements (e.g., list ⟨4, 1, 6, 8, 1, 4, 4⟩)
  - Performance reasons (sorting necessary)

rank
C3
C4

- Omitting the keyword **distinct** will result in a copy of the complete *rank* column of the table professors with all duplicates and the same order of values (list model!)

SQL Queries on a Single Table (III)

- Query 3: Find all personnel identifiers and names of C4 professors.

  select pers-id, name **from** professors **where** rank = 'C4';

pers-id	name
2125	Sokrates
2126	Russel
2136	Curie
2137	Kant

- This query is a search query, that is, subject to a search condition, it filters out a subset of the data stored on disk in the database
- **where** clause – horizontal (row) restriction
  select clause – vertical (column) restriction

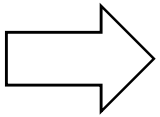
- Simple SQL queries are near to natural language formulations of commands; however, this does not hold for (more) complex SQL queries
- Another problem: SQL has simple data types only, applications with complex objects (e.g., life sciences dealing with enzyme and nucleotide sequences, multimedia systems dealing with videos, images, and audio) do not like SQL very much since the data types and operations provided are too low-level and not expressive enough

SQL Queries on a Single Table (IV)

- Query 4: Determine all students whose semester number is larger than 2 and whose registration identifier is less than 27000

```
select reg-id, name, sem
from students
where sem > 2 and reg-id < 27000;
```

students		
reg-id	name	sem
24002	Xenokrates	18
25403	Jonas	12
26120	Fichte	10
26830	Aristoxenos	08
27550	Schopenhauer	06
28106	Carnap	03
29120	Theophastros	02
29555	Feuerbach	02



reg-id	name	sem
24002	Xenokrates	18
25403	Jonas	12
26120	Fichte	10
26830	Aristoxenos	08

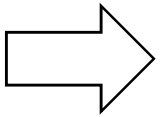
SQL Queries on a Single Table (V)

- Query 5: List all students whose semester number is larger than or equal to 8 and less than or equal to 18.

```
select *
from students
where sem >= 8 and sem <= 18;
```

```
select *
from students
where sem between 8 and 18;
```

students		
reg-id	name	sem
24002	Xenokrates	18
25403	Jonas	12
26120	Fichte	10
26830	Aristoxenos	08
27550	Schopenhauer	06
28106	Carnap	03
29120	Theophastros	02
29555	Feuerbach	02



reg-id	name	sem
24002	Xenokrates	18
25403	Jonas	12
26120	Fichte	10
26830	Aristoxenos	08

SQL Queries on a Single Table (VI)

- Query 6: Create a new table about students where the semester number of each student is incremented by 1.

  select reg-id, name, sem + 1
  from students;

- The **select** clause may contain arithmetic expressions involving the numerical operators +, –, *, and /
- The table *students* remains unchanged
- The third attribute has no name in the result table since it is a computed value
- To assign a new name (e.g., *new_sem*) to the third column, we can use the **as** clause and write

  select reg-id, name, sem + 1 **as** new_sem
  from students;

- The result table has the schema (reg-id, name, new_sem)

SQL Queries on Multiple Tables (I)

- Query 1: Determine the names of professors who hold lectures. Further, show the titles of the corresponding lectures.

```
select name, title
from professors, lectures
where pers-id = held_by;
```

- Example of an equi-join (as a special form of the theta-join) in SQL
- Join attributes are *pers-id* and *held_by* (find query result table yourself)
- Another formulation of the same query is

```
select name, title
from professors join lectures on pers-id = held_by;
```

SQL Queries on Multiple Tables (II)

- Query 2: Determine the names of professors who hold the lecture titled "maieutics".

```
select name, title
from professors, lectures
where pers-id = held_by and
      title = 'maieutics';
```

```
select name, title
from professors, lectures
where (pers-id, title) =
      (held_by, 'maieutics');
```

name	title
Sokrates	maieutics

- The **where** clause can contain any logical expression that deploys logical connectives, e.g.,

… **where** (*a* **or** *b* **or** *c*) **and** ((*d* **or** *e*) **and** *f*) [*a*, *b*, *c*, *d*, *e*, *f* are conditions]

SQL Queries on Multiple Tables (III)

- Query 3: Which students attend which lecture? Output student names and lecture titles.

```
select name, title
from students, attends, lectures
where students.reg-id = attends.reg-id and
      attends.id = lectures.id;
```

- Three-table join
- Attribute names behind **select** and **where** must be always unique; if two table schemas share an attribute name, use dot-notation in the form of <table name>.<attribute name>
- The query above is a natural join and can be explicitly expressed as such:

```
select name, title
from students natural join attends natural join lectures
```

- Natural join is executed with respect to all shared attributes of *students* and *attends* as well as *attends* and *lectures*

SQL Queries on Multiple Tables (IV)

- Alternative formulation of the first version of the query by using tuple variables that are bound to tables:

```
select s.name, l.title
from students as s, attends as a, lectures as l
where s.reg-id = a.reg-id and a.id = l.id;
```

- The keyword **as** represents the “rename” operation in this context and is optional

SQL Queries on Multiple Tables (V)

- Query 4: Combine all professors with all students.

  select * **from** professors, students;

- Alternative formulation of this query using the explicit Cartesian product operator:

  select * **from** professors **cross join** students;

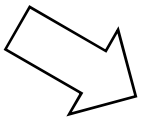
SQL Queries on Multiple Tables (VI)

- Query 5: List all students together with their performed tests by maintaining the complete student information.

```
select s.*, t.*
from students s left outer join test t on s.reg-id = t.reg-id;
```

students		
reg-id	name	sem
24002	Xenokrates	18
25403	Jonas	12
26120	Fichte	10
26830	Aristoxenos	08
27550	Schopenhauer	06
28106	Carnap	03
29120	Theophastros	02
29555	Feuerbach	02

tests			
reg-id	id	pers-id	grade
28106	5001	2126	1
25403	5041	2125	2
27550	4630	2137	2



reg-id	name	sem	id	pers-id	grade
24002	Xenokrates	18	null	null	null
25403	Jonas	12	5041	2125	2
26120	Fichte	10	null	null	null
26830	Aristoxenos	08	null	null	null
27550	Schopenhauer	06	4630	2137	2
28106	Carnap	03	5001	2126	1
29120	Theophastros	02	null	null	null
29555	Feuerbach	02	null	null	null

Right outer join (**right outer join**) and full outer join (**full outer join**) are similar

SQL Queries on Multiple Tables (VII)

- The asterisk symbol '*' used in the **select** clause means "all attributes"
- The use of "s.*" means that all attributes of the relations *students* are to be selected
- An equivalent formulation of the query is:

```
select *
from students natural left outer join test;
```

Set Operations (I)

- Query 6: Determine the names of all university employees, i.e., the names of all professors and all assistants

```
(select name from assistants)
union
(select name from professors);
```

- Alternative formulation:

```
(select * from assistants)
union corresponding by name
(select * from professors);
```

- All duplicates are automatically removed
- Assuming there would be professors and assistants with the same names and duplicates should be maintained, we would use the **all** clause and write:

```
(select name from assistants)
union all
(select name from professors);
```

```
(select * from assistants)
union all corresponding by name
(select * from professors);
```

Set Operations (II)

- Query 7: Find the identifiers of professors who teach “ethics” *or* “maieutics”.

```
(select p.pers-id from professors p, lectures l
 where p.pers-id = l.held_by and l.title = ‘ethics’)
union
(select p.pers-id from professors p, lectures l
 where p.pers-id = l.held_by and l.title = ‘maieutics’);
```

- Query 8: Find the identifiers of professors who teach “ethics” *and* “maieutics”.

```
(select p.pers-id from professors p, lectures l
 where p.pers-id = l.held_by and l.title = ‘ethics’)
intersect
(select p.pers-id from professors p, lectures l
 where p.pers-id = l.held_by and l.title = ‘maieutics’);
```

Set Operations (III)

- Query 9: Find the identifiers of professors who teach “ethics” *but not* “maieutics”.

```
(select p.pers-id from professors p, lectures l
 where p.pers-id = l.held_by and l.title = ‘ethics’)
except
(select p.pers-id from professors p, lectures l
 where p.pers-id = l.held_by and l.title = ‘maieutics’);
```

- Schema compliant table schemas required as operands

- Operations **union**, **except**, and **intersect** produce tables as sets of tuples, elimination of duplicates

- Operations **union all**, **except all**, and **intersect all** maintain duplicates in result table

Set Operations (IV)

- Semantics of the set operations extended by the clause **all**: Let $F(R, t)$ describe the frequency of tuple t in table R. Then the number of duplicates with respect to the tables R and S is defined as:
  - $\forall\, t : F(R \textbf{ union all } S, t) = F(R, t) + F(S, t)$
  - $\forall\, t : F(R \textbf{ except all } S, t) = \textbf{if } F(R, t) \geq F(S, t) \textbf{ then } F(R, t) - F(S, t) \textbf{ else } 0$
  - $\forall\, t : F(R \textbf{ intersect all } S, t) = \min(F(R, t), F(S, t))$

Null Values (I)

- Each SQL data types contains a special value **null**
- The special value **null** for an attribute value in a tuple indicates the value is *unknown* or *unclear* (e.g., the age of a person)
- Null values present special problems in relational operations including arithmetic operations, comparison operations, and set operations
- The result of an arithmetic expression (involving, e.g., +, –, *, /) is null if any of the input values is null
  - Example 1: Expression r.A + 7: If r.A is null for a particular tuple, the result of the expression is null
  - Example 2: Expression r.A * s.B: If any of the two arguments, or both arguments have the value null for particular tuples *r* and *s*, the result is null

Null Values (II)

- Comparisons involving null values are a bigger problem
- Does the comparison "5 < **null**" yield *true* or *false*?
  - To say this is true is wrong since we do not know what the null value represents
  - To say this is false is also wrong since then the expression "**not** (5 < **null**)" would evaluate to true, which does not make sense
  - Therefore, SQL adds a third truth value **unknown** that is the result of any comparison involving a *null* value
- SQL uses a *three-valued logic* with the values **true**, **false**, and **unknown**
- Boolean operations **and**, **or**, and **not** are extended:

not	
true	false
unknown	unknown
false	true

and	true	unknown	false
true	true	unknown	false
unknown	unknown	unknown	false
false	false	false	false

Null Values (III)

or	true	unknown	false
true	true	true	true
unknown	true	unknown	unknown
false	true	unknown	false

- In the **where** clause only those tuples are added to the result where the filter condition yields *true*
- The condition "**where** A **is null**" allows to select all tuples with a *null* value in attribute A
- The condition "**where** A **is not null**" allows to select all tuples that do not have a *null* value in attribute A
- Two copies of a tuple, such as t_1 = ('A', null) and t_2 = ('A', null), are considered to be equal, even if some attributes have a *null* value, although a comparison "null = null" would usually return *unknown*, rather than *true*
- This also holds for the set operations **union**, **intersect**, and **except**

String Operations (I)

- In SQL strings are enclosed in *single* quotes, for example, ‘database’
- Equality operation on strings is case sensitive, for example, the expression ‘database’ = ‘DATABASE’ yields false
- Examples of string operations
  - String concatenation performed by “||”, e.g., the expression ‘database ’ || ‘system’ yields ‘database system’
  - Extracting substrings with the method *substring*, e.g., the expression **substring**(‘database’, 3, 4) yields the string ‘taba’
  - Finding the length of strings with the method *length*, e.g., the expression **length**(‘database’) yields the number 8
  - Converting strings to uppercase (lowercase) with the method *upper* (*lower*), e.g., the expression **upper**(‘database’) yields the string ‘DATABASE’
- Unfortunately, different DBS offer different sets of string operations; further, the same string operations can have a slightly different syntax