The WITH Clause for Defining Temporary Tables

- A way of defining a temporary table whose definition is available only to the query in which the **with** clause is used

- Query 14: Output the titles of those lectures that are attended by more than 20 students.

```
with attendance_rate(id, number) as
        (select id, count(*)
         from attends
         group by id)
select l.title
from lectures as l, attendance_rate as a
where l.id = a.id and a.number > 20;
```

Scalar Subqueries in the WHERE, SELECT, or HAVING Clause (I)

- A scalar subquery returns a single tuple that consists of a single column and a single row and that is interpreted and used as a *single value*
- A scalar subquery can occur in the **select**, **where**, or **having** clause
- Query 15: Determine the name and semester number of those students with a semester number less than the average.

```
select name, sem
from students
where sem < (select avg(sem) from students);
```

- It must be guaranteed that the subquery only returns a single value; otherwise, a runtime error occurs
- A scalar subquery can be used immediately following a relational comparison operator (=, <, >, <=, >=, <>)

Scalar Subqueries in the WHERE, SELECT, or HAVING Clause (II)

- Query 16: List the name and semester number of all students whose semester number is greater than the average semester number, and compute by how much their semester number is greater than the average.

```
select name, sem, sem – (select avg(sem) from students) as semDiff
from students
where sem > (select avg(sem) from students);
```

- The subquery computes the average semester number and is replaced by that number both in the **where** clause and the **select** clause

- One is not allowed to write "**where** sem > **avg**(sem)" since aggregation functions cannot be used in the **where** clause

Views (I)

- Users are usually not allowed to see the complete conceptual schema since
  - a large part of the schema is not of interest to them

    Example: An assistant is usually not interested in the information which student attends which class
  - authorization and security considerations forbid access to sensitive data

    Example: A student should get access to an instructor's name and department but not have the authorization to see her salary amount

- A view
  - is a virtual relation that does not necessarily exist in the database but can be produced or derived from stored base tables by a particular user (group) at the time of request
  - is not precomputed and stored but instead is computed by executing its *specifying query* whenever the virtual relation is used
  - is not part of the conceptual schema
  - is defined as an SQL query on one or more base tables or views

Views (II)

- A view (*continued*)
  - determines which data a user may access and which data a user must not access
  - implements an external view in the three-level architecture of a DBS

- DBMS stores the definition (i.e., specifying query) in the system catalog

- View resolution: When the query processor encounters a view name in an SQL query, it
  - looks up the view definition in the system catalog (step 1)
  - textually replaces the view name in the SQL query by its specifying query (step 2)
  - recursively performs steps 1 and 2 if there should be still view names in the modified query
  - executes the corresponding, resolved, and equivalent query

Views (III)

- Example: A view on table *tests* shall express the restriction that not each user is allowed to see the results of an exam.

  ```
  create view tests_view(reg-id, id, pers-id) as
  select reg-id, id, pers-id from tests;
  ```

- Query 17: Output the names of all students who have taken a test.

  SQL query

  ```
  select s.name
  from students as s, test_view as t
  where s.reg-id = t.reg-id;
  ```

  SQL query after view resolution

  ```
  select s.name
  from students as s,
          (select reg-id, id, pers-id
           from tests) as t
  where s.reg-id = t.reg-id;
  ```

Views (IV)

- All updates to a base table are immediately reflected in all views that access that base table; however, this is not the case vice versa for updates on views
- Example 1

  ```
  create view test_situations(reg-id, id, pers-id) as
  select reg-id, id, pers-id
  from tests;
  ```

  The following update on *test_situations* is accepted by the DBMS:

  ```
  insert into test_situations values(26120, 5259, 2133);
  ```

  - The effect is that the tuple (26120, 5259, 2133, **null**) is inserted into the base table *tests* since the grade is unknown and since a null value is allowed for the attribute grade
  - The grade would have to be later updated in the base table *tests* (by another user).

Views (V)

- Problem of view updatability: Views have the inherent problem that they frequently cannot be modified through insertions, deletions, and updates

- Example 2

  ```
  create view grading(pers-id, avg-grade) as
  select pers-id, avg(grade)
  from tests
  group by pers-id;
  ```

  This view is not changeable since it contains the computed attribute avg-grade. An update operation cannot be transferred to the original base relation *tests*. The following operation is rejected by the DBMS:

  ```
  update grading
  set avg-grade = 3.0
  where pers-id = (select pers-id from professors where name = “Sokrates”);
  ```

Views (VI)

- Example 3

  ```
  create view lecture-view as
  select title, credits, name
  from lectures, professors
  where held_by = pers-id;
  ```

  The insertion of a new lecture is impossible as the following attempt shows:

  ```
  insert into lecture-view
  values (‘nihilism’, 2, ‘Nobody’);
  ```

  - In order to enter a tuple into a view, the DBMS must be able to assign the values to the original base tables.
  - But this is not always possible (for a number of possible reasons).
  - Here the view removes the keys of both base tables *lectures* and *professors*.

Views (VII)

- An SQL view is said to be updatable (i.e., insertions, updates, and deletions can be applied to the view) if the following conditions are all satisfied by the specifying query:
  - The **distinct** keyword is *not* used, i.e., duplicate rows are not removed
  - The **from** clause has only one table, which can be a base table or another updatable view (this excludes views based on a join, union (**union**), intersection (**intersect**), or difference (**except**))
  - The **select** clause contains only attributes of the table in the **from** clause and does not have any expressions, aggregates, and constants, and no attribute name appears more than once
  - Any attribute not listed in the **select** clause can be set to *null*, that is, it does not have a **not null** constraint and is not part of a primary key
  - The **where** clause does not include any nested **select** clauses that reference the table in the **from** clause
  - The query does not have a **group by** or **having** clause

Views (VIII)

- Advantages
  - *Data independence*: A view can present a consistent, unchanging picture of the database structure, even if the underlying base tables are changed (e.g., addition of new attributes, removal of attributes not required by the view, changed relationships, tables split, restructured)
  - *Currency*: Changes to any of the base tables in the specifying query are immediately reflected in the view
  - *Improved security*: Each user gets restricted and controlled access to a specified part of the database through a small set of views
  - *Reduced complexity*: A view can simplify queries by merging data from several tables into a single table and thereby transforming multi-table queries into single-table queries
  - *Convenience*: Users are presented with only that part of the database that they need to see
  - *Customization*: Views allow the use of the same underlying base tables by different users in different ways

Views (IX)

- Disadvantages
  - *Update restrictions*: Problem of view updatability discussed before
  - *Structure restriction*
    - The structure of a view is determined at the time of its creation
    - If the specifying query is of the form **select** * **from** …, the * refers to the attributes of the base table present when the view was created.
    - If attributes are added later to the base table, the new attributes will not appear in the view, unless the view is dropped and recreated
  - Performance
    - View resolution that requires the computation of joins and other expensive database operations may take a long time and has to be performed every time the view is accessed
    - View materialization as a solution: Store the view result as a temporary table in the database when the view is first queried, afterwards make use of the materialized view that is much faster

Authorization (I)

- Authorization refers to the permission to perform an operation on data in the database
- Authorizations on data include permissions to read data, insert new data, update data, and delete data
- Each of these types of authorizations is called a privilege
- Each object that is created in SQL has an owner
- The owner is initially the only person who may perform any operations on the object
- When a user submits a query or an update, SQL first checks whether the query or update is authorized, based on the authorizations that the user has been granted; otherwise, the query or update is rejected
- Users may grant privileges to other users as well as revoke privileges from users

Authorization (II)

- SQL statement to grant privileges

```
grant {<privilege list> | all privileges}
on <relation/view name>
to {<user/role list> | public}
[with grant option];
```

- Example 1: Grant the users “Russel” and “Kant” (who are professors) **select** (i.e., read) authorization on the table *tests*.

```
grant select on tests to Russel, Kant;
```

- Example 2: Grant the professors “Russel” and “Kant” **update** (i.e., modification) authorization on the attribute *grade* of the table *tests* as well as **delete** authorization on the same table.

```
grant update(grade), delete on tests to Russel, Kant;
```

Authorization (III)

- Example 3: Give all users the privilege **select** to the table *lectures*.

  grant select on lectures **to public**;

  All current and future users are allowed to retrieve data from the table lectures

- Use of roles to assign privileges to groups of users, e.g.,

  create role C4_professor;

  create role C3_professor;

  create role assistant;

  create role student;

  grant C4_professor **to** Sokrates, Russel, Curie, Kant;

  grant C3_professor **to** Kopernikus, Popper, Augustinus;

  grant assistant **to** Platon, Aristoteles, Wittgenstein, Rhetikus, Newton, Spinoza;

  grant student **to** Xenokrates, Jonas, Fichte, Aristoxenos, Schopenhauer, Carnap, Theophastros, Feuerbach

Authorization (IV)

- Example 4: Grant all C4 professors the **select** privilege on the table *tests*.

  grant select on tests **to** C4_professor;

- Example 5: Give C4 professors and C3 professors all privileges on the table *students*.

  grant all privileges
  on students **to** C4_professor, C3_professor
  with grant option;

  Any C4 professor and any C3 professor can retrieve tuples from the table *students* and also insert, update, and delete from this table

  The keyword **with grant option** means that C4 professors and C3 professors can pass these privileges on to other users or roles (e.g., assistants)

  This can lead to authorization-grant graphs showing the transfer of privileges

Authorization (V)

- The **revoke** statement can take away all or some of the privileges that were granted with the **grant** statement to a user or a role

- Format:

  revoke [**grant option for**] {<privilege list> | **all privileges**}
  on <relation/view name>
  from {<user/role list> | **public**} [**restrict** | **cascade**];

- Keyword **all privileges** refers to all privileges granted to a user or a role; these are all revoked

- Example 6: Revoke the **select** privilege (all privileges) on the table *tests* from the users "Russel" and "Kant".

  revoke select on tests **from** Russel, Kant;

  revoke all privileges on tests **from** Russel, Kant;

Authorization (VI)

- Revocation of a privilege from a user/role may cause other users/roles to lose that privilege too, called cascading revocation
- Cascading revocation is the default and specifically indicated by the optional keyword **cascade**
- Example 7: Revoke all privileges on the table *students* from C4 professors and C3 professors.

```
revoke all privileges
on students
from C4_professor, C3_professor;
```

```
revoke all privileges
on students
from C4_professor, C3_professor
cascade;
```

- Use the keyword **restrict** to prevent cascading revocation

```
revoke all privileges on students from C4_professor, C3_professor restrict;
```

If there are any cascading revocations, the system returns an error, and the revoke action is not carried out

Authorization (VII)

- The optional **grant option for** clause allows privileges passed on via the **with grant option** clause of the **grant** statement to be revoked separately from the privileges themselves
- Example 8: The statement

  revoke grant option for all privileges
  on students
  from C4_professor, C3_professor;

  only revokes the grant option, rather than the actual **all privileges** privilege