Today's lecture

1. Transactions
2. Views
3. Access Rights in SQL

Transactions

- A logical unit of work consisting of one or more SQL statements
- Atomic transaction
  - **Fully executed** or
  - **Rolled back** as if it never occurred
- Isolation from concurrent transactions
  - Changes made by a transaction are not visible to other concurrently executing transactions until the transaction completes

- Transaction model based on two SQL statements:
  - **COMMIT**
  - **ROLLBACK**
- Transactions begin implicitly
  - Ended by commit work or rollback work
- Default on most databases: each SQL statement commits automatically
- Can turn off auto commit for a session (e.g. using API)

VIEWS

- Definition
- View Creation and Destruction
- Updating Views
- Types of Views

Views

- One database often supports multiple applications
  - Slightly different pictures of the world.
- **Views** help accommodate this variation without storing redundant data.

Views

- In some cases, it is not desirable for all users to see the entire logical model (that is, all the actual relations stored in the database.)

Example:

```
Employee(ssn, name, department, project, salary)
```

Consider a person who needs to know the name and project of employees in the 'Development' department, but not the salary. This person should see a relation described, in SQL, by

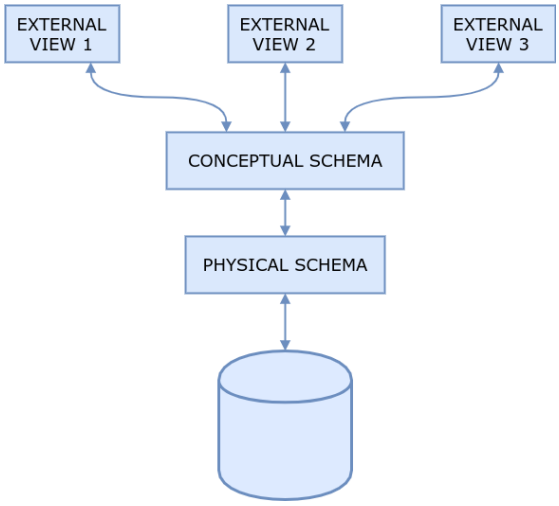
```
SELECT name, project
FROM   Employee
WHERE  department = 'Development'
```

Views

- Provide a mechanism to hide certain data from the view of certain users.
- Any relation that is not part of the conceptual model but is visible to a user as a *“virtual relation”* is called a **view**.
- Not physically stored.

Levels of Abstraction

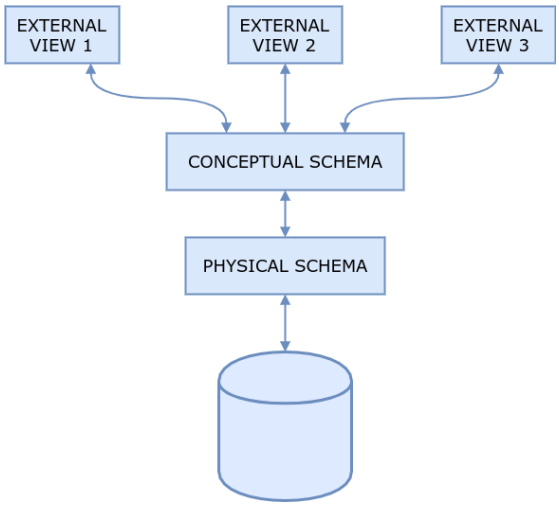
- Multiple views
- A single Conceptual (Logic) Schema
- A single Physical Schema



Levels of Abstraction

Physical Level

- Lowest level
- How the data is physically stored
- It includes
  - Where the data is located
  - File structures
  - Access methods
  - Indexes
- Managed by the **Database Administrator**

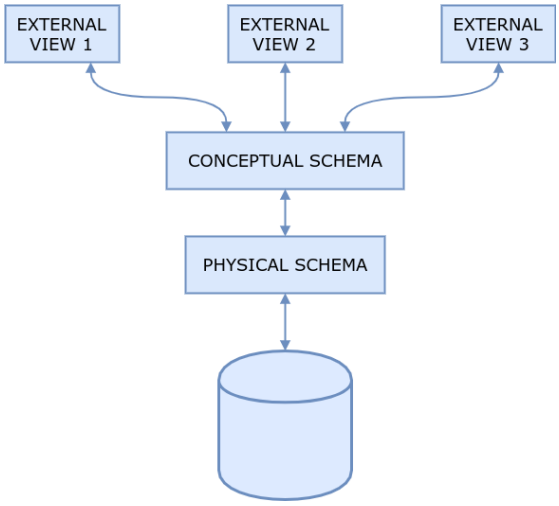


Levels of Abstraction

Conceptual or Logical Level

- Middle level
- What data is in the DB
- It consists of the schemas described with CREATE TABLE statements
  - Has all the data in the DB

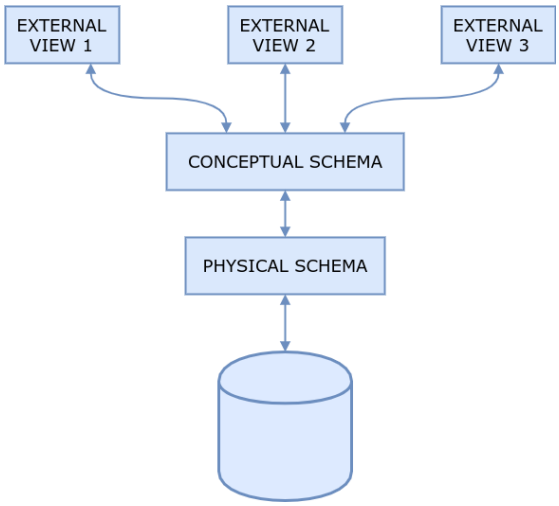
    - Has no information on what a user views at external level
- Managed by **Database Designers**



Levels of Abstraction

External or View Level

- Highest Level
- Combination of base tables and views
- Views define how certain Users/Groups see data:
    - Full or partial data based on the business requirement
    - Users have different views, based on their levels of access rights
- Exposed to **Users/Applications** and **Database Designers**.

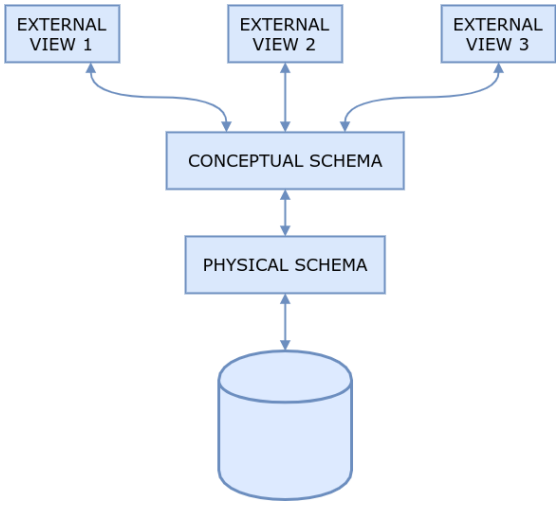


Data Independence

- A database model exhibits data independence if:

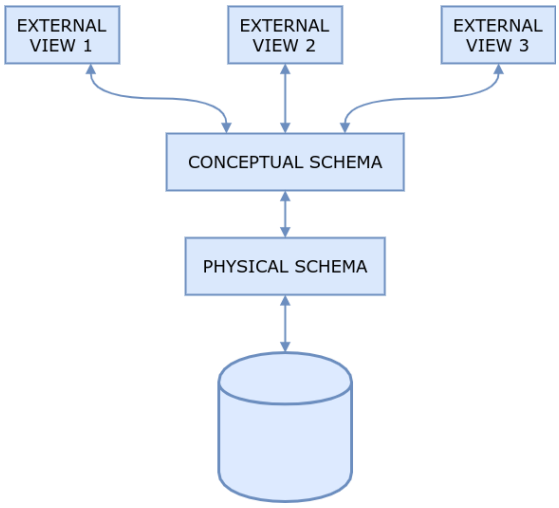
> Application programs are protected from changes in the conceptual and physical schemas.

- Why is this important?
  - *Everything changes.*
- Each higher level of the data architecture
  is immune to changes of the next lower level of the architecture.



Data Independence Types

- **Physical data independence**
  - Can modify the *physical schema* without causing application programs to be rewritten.
- **Logical data independence**
  - Can modify the *logical schema* without causing application program to be rewritten.



View Creation and Destruction

- A view is defined using the create view statement which has the form

```
CREATE VIEW view_name AS
< QUERY >
[WITH CHECK OPTION]
```

  where < query > is any legal SQL expression.
- Once a view is defined, the view name can be used to refer to the **virtual relation** that the view generates.

View Creation and Destruction

- **CHECK OPTION**
  - Ensures that all UPDATE and INSERTs operations satisfy the condition(s) in the view definition.
  - Otherwise, the UPDATE or INSERT returns an error.
  - Not implemented in SQLite
- **Destruction**

```
DROP VIEW <view_name>
```

Views for Security

Example:

```
Student(studID, name, address, major, gpa)
```

- This is a view of the Student table without the gpa field.

```
CREATE VIEW SecStudent AS
SELECT studID, name, address, major
FROM   student
```

Views for Extensibility

Example:

- A company's database includes a relation:

```
Part (PartID, weight,...)
```

- Weight is stored in pounds
- The Company is purchased by a firm that uses metric weights
- Databases must be integrated and use Kg.
  - But old applications use pounds.

Views for Extensibility

- Solution:
  - Base table with kilograms becomes *MetricPart* for the integrated company

```
CREATE VIEW MetricPart AS
SELECT PartID, 2.2046*weight,       -- no
other changes
FROM Part
```

  - Old programs still call the table *Part*

Data Partitioning

- Sometimes the data of a database is partitioned.
- **Horizontal:** projection on certain attributes
  - Break up our table based on rows
  - Useful when some attributes are bulky or rarely used
  - Distributed databases
- **Vertical:** selection on certain values (ClientsParis, ClientsLyon)
  - Splitting out extra columns into their own table(s)

Another Example

- Consider the following relations

```
Person(name, city)
Purchase(buyer, seller, product, store)
Product(name, maker, category)
```

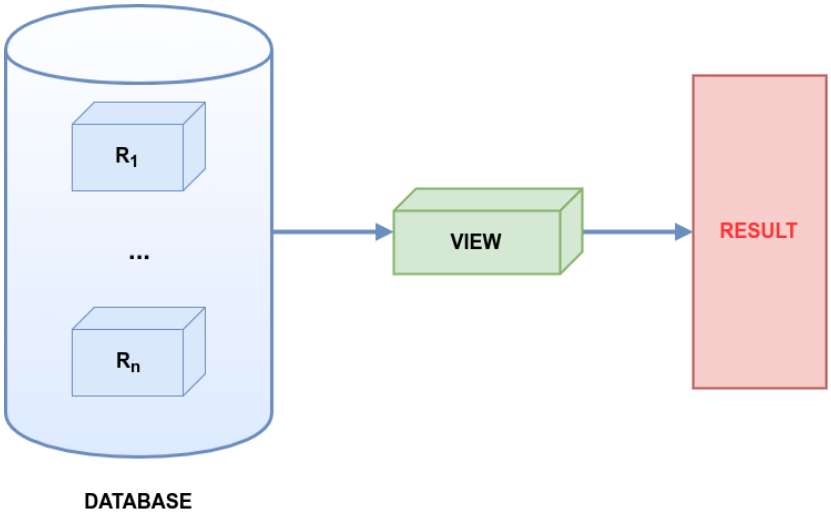
```
CREATE VIEW  SeattleView  AS
SELECT buyer, seller, product, store
FROM   Person, Purchase
WHERE  Person.city = 'Seattle'
       AND Person.name = Purchase.buyer
```

- We have a new 'virtual table':

```
SeattleView(buyer, seller, product, store)
```

Using (Querying) a View

- Transparency for the user
  - Handled as tables in the database
- Simplify the user's queries
- Useful in architectures client-server



Using (Querying) a View

Query using SeatleView

```
SeattleView(buyer, seller, product, store)
  Product(name, maker, category)

SELECT name, store
  FROM   SeattleView, Product
  WHERE  SeattleView.product = Product.name
         AND Product.category = 'shoes'
```

- When you enter a query that mentions a **view** in the FROM clause, the DBMS expands/rewrites your query to include the view definition.

View Expansion

- Query using a **view**

```
SELECT name, SeattleView.store
FROM   SeattleView, Product
WHERE  SeattleView.product = Product.name
     AND Product.category = 'shoes'
```

- Expanded query

```
SELECT name, Purchase.store
FROM   Person, Purchase, Product
WHERE  Person.city = 'Seattle'
     AND Person.name = Purchase.buyer
     AND Purchase.product = Product.name
     AND Product.category = 'shoes'
```

Another Example

- Query using a **view**

```
SELECT  buyer, seller
FROM    SeattleView
WHERE   product= 'gizmo'
```

- Expanded query

```
SELECT buyer, seller
FROM   Person, Purchase
WHERE  Person.city = 'Seattle'
       AND Person.name = Purchase.buyer
       AND Purchase.product= 'gizmo'
```

Views Defined Using Other Views

- One view may be used in the expression defining another view
- A view relation v_1 is said to **depend directly on** a view relation v_2 if v_2 is used in the expression defining v_1
- A view relation v_1 is said to **depend on** view relation v_2 if either v_1 depends directly on v_2 or there is a path of dependencies from v_1 to v_2
- A view relation v is said to be **recursive** if it depends on itself.

Updating Views

- How can we insert a tuple into a table that *doesn't exist*?

```
Employee(ssn, name, department, project, salary)

CREATE VIEW  Developers AS
   SELECT name, project
   FROM  Employee
   WHERE department = 'Development'
```

Updating Views

- How can we insert a tuple into a table that *"doesn't exist"*?

Example:

```
Employee(ssn, name, department, project, salary)
Developers(name, project)
```

The following insertion:

```
INSERT INTO  Developers
VALUES('Joe', 'Optimizer')
```

becomes:

```
INSERT INTO  Employee
VALUES(NULL, 'Joe', NULL, 'Optimizer', NULL)
```

Non-Updateable Views

- Consider the relations

  ```
  Person(name, city)
  Purchase(buyer, seller, product, store)
  ```

  and the view

  ```
  CREATE VIEW  SeattleView  AS
  SELECT seller, product, store
  FROM   Person, Purchase
  WHERE  Person.city = 'Seattle'
         AND Person.name = Purchase.buyer
  ```

- How can we add the following tuple to the view?

  ```
  ('Joe',  'Shoe Model 12345',  'Nine West')
  ```

- We need to add 'Joe' to *Person* first.
  - How?!
  - One time?
  - Multiple times?

Updating Views

- Most SQL implementations allow updates **only on simple views**.
  - The `FROM` clause has only one database relation.
  - The `SELECT` clause contains only attribute names of the relation.
    - No expressions, aggregates, or distinct specification.
  - Any attribute not listed in the `SELECT` clause can be set to `NULL`.
  - The query does not have a `GROUP BY` or `HAVING` clause.

Updating Views

- SQLite views are read-only and thus you may not be able to execute a DELETE, INSERT or UPDATE statement on a view.
  - A workaround exists
  - Not in the scope of this class
  - This is why WITH CHECK OPTION is not implemented

Types of Views

- **Virtual views**
  - Used in databases
  - Computed only **on-demand** – *slow at runtime*
  - Always up to date

Types of Views

- **Materialized views**
  - A physical table containing all the tuples in the result of the query defining the view
  - Used in Data Warehouses (but recently also in DBMS)
  - Precomputed offline – *fast at runtime*

- If relations used in the query are updated, the materialized view result becomes out of date
  - Need to maintain the view, by updating the view whenever the underlying relations are updated.

Data Warehouse

- A relational database designed for query and analysis rather than for transaction processing.
- Usually contains historical data derived from transaction data.
- Separates analysis workload from transaction workload.
- Enables an organization to consolidate data from several sources.

Advantages/Disadvantages of Views

ADVANTAGES	DISADVANTAGES
Data independence	Update restriction
Currency	Structure restriction
Improved security	Performance
Reduced complexity	
Convenience	
Customization	
Data integrity	

Summary

- A view is a stored query definition
- Views can be very useful
  - Privacy

    - Easier query writing
    - Extensibility
- Not all views can be updated unambiguously
- Three levels of abstraction in a relational DBMS
    - Yields data independence, logical and physical

Access Rights in SQL

- The SQL security model
- Granting and revoking privileges

Discretionary Access Control

- Each user is given appropriate access rights (*privileges*) on specific DB objects
- Explicit grant of rights on objects to individuals.
- Users obtain certain privileges when they create an object
    - Can *pass* some or all of these privileges to other users **at their discretion**
- Although flexible, can be circumvented by devious unauthorized user tricking an authorized user into revealing sensitive data.

Terminology

- **Privacy** Users should not be able to see and use data they are not supposed to.

> e.g., A student can't see other students' grades.

- **Security** No one should be able to enter the system and / or impact its behavior without being authorized to do so.

> e.g., Delete or change data without being authorized

- **Integrity** Authorized users should not be able to modify things they are not supposed to.

> e.g., Only instructors can assign grades.

- **Availability** Users should be able to see and modify things they are allowed to.

> e.g. The DB should always be operational

SQL Security Related Terminology

- **User**
  - Not the schema object, just a name for a session of an individual user
  - Identification by Authorization ID (user name)
- **Role**
  - Name for a role, to which rights may be assigned
  - May be granted to users / applications
- **Privileges** (Rights)
  - System privileges
  - Object (data) privileges: creator has all privileges
- **Operations**
  - GRANT < privilege >
  - REVOKE < privilege >

Roles and Users

- **Roles** define a set of privileges for a (potentially) large set of **Users**

```
CREATE ROLE sales_people;
-- grant some privileges to sales_people
-- grant sales_people role to users
```

- Much more economic than direct privileges
- Roles may be assigned to roles
- Often assigned to applications instead of individual users

Privileges

- Right to perform SQL statement type on objects
- Assigned to users or roles (authorization IDs)
- *Creator of object*: all privileges for that object
- *Administrator*: management of system privileges

Privileges

- The privileges defined by the ISO standard:
  - SELECT - retrieve data from a table
  - INSERT - insert new rows into a table
  - UPDATE - modify rows of data in a table
  - DELETE - delete rows of data from a table
  - REFERENCE - reference columns of a named table in integrity constraints
  - USAGE - use domains, collations, character sets, and translations

Grant Privileges

- Syntax

```
GRANT <privileges> ON <object>
TO [<users>|<role>]
[WITH GRANT OPTION]
```

- GRANT OPTION: Right to pass privilege on to other users
  - Only owner can execute CREATE, ALTER, and DROP

Example: Privilege to INSERT particular columns in a table

```
GRANT INSERT
ON <tablename(<attributenames>)>
TO <users>
[WITH GRANT OPTION]
```

- Access matrix : < user > has < right > on < object >

Examples

```
GRANT INSERT, SELECT ON Movie TO Klaus
```

- Klaus can query **'Movie'** or insert tuples into it.

```
GRANT DELETE ON Movie TO shop_owner WITH GRANT OPTION
```

- Anna can delete **'Movie'** tuples, and also authorize others to do so

---

```
GRANT UPDATE (price_Day) ON Movie TO movie_staff
```

- Staff can update (only) the price field of **'Movie'** tuples

---

```
GRANT SELECT ON MovieView TO Customers
```

- This does NOT allow the customers to query **'Movie'** directly!

Revoke Privileges

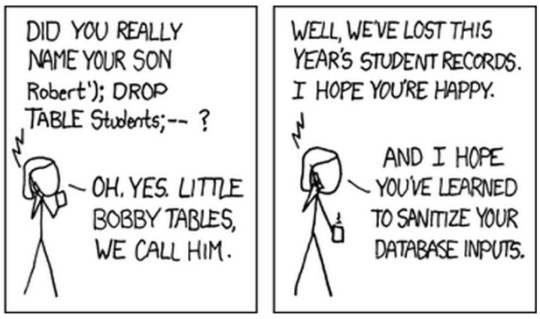
- Syntax

```
REVOKE <privileges>
ON <object>
FROM <users>
[RESTRICT | CASCADE]
```

- RESTRICT: only revoke if none of the privileges have been granted by these users.
- CASCADE: revoke from all users that have been granted the privilege by these users.
- Privilege given from different users must be revoked from all users to loose privilege.

Summary

- Security of DB and their applications is extremely important.
- Roles make privileges with many users manageable.
- Views also play an important role.
- Fine granular access restriction on objects is very important.



https://www.xkcd.com/327/