CSE1500 - WEB AND DATABASE TECHNOLOGY

DB LECTURE 7

EVEN MORE ON SQL

Alessandro Bozzon

cse1500-ewi@tudelft.nl

AT THE END OF THIS LECTURE, YOU SHOULD BE ABLE TO….

- **Describe** and **design** SQL queries that make use of the WITH clause
- **Describe** and **design** SQL queries that make use of Views
- **Describe** and **design** constraints at application level with Triggers
- **Administer** users and permissions

EXAMPLE DATABASES

EXAMPLE DB1: EMPLOYEES

Employee

FirstName	Surname	Dept	Office	Salary	City
Mary	Brown	Administration	10	45	London
Charles	White	Production	20	36	Toulouse
Gus	Green	Administration	20	40	Oxford
Jackson	Neri	Distribution	16	45	Dover
Charles	Brown	Planning	14	80	London
Laurence	Chen	Planning	7	73	Worthing
Pauline	Bradshaw	Administration	75	40	Brighton
Alice	Jackson	Production	20	46	Toulouse

Department

DeptName	Address	City
Administration	Bond Street	London
Production	Rue Victor Hugo	Toulouse
Distribution	Pond Road	Brighton
Planning	Bond Street	London
Research	Sunset Street	San Joné

EXAMPLE DB2: PRODUCTS

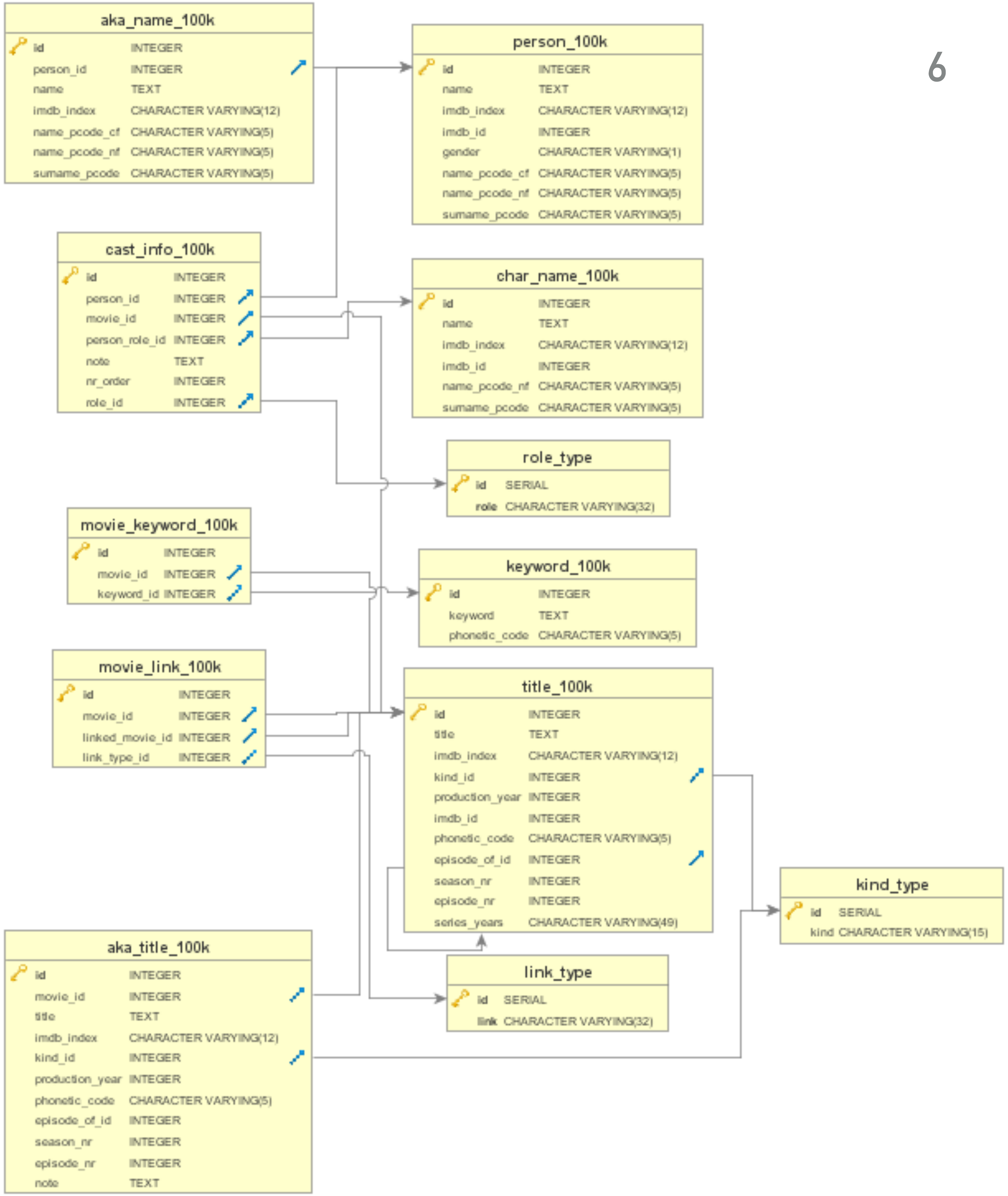
Supplier			
CodeS	NameS	Shareholders	Office
S1	John	2	Amsterdam
S2	Victor	1	Den Haag
S3	Anna	3	Den Haag
S4	Angela	2	Amsterdam
S5	Paul	3	Utrecht

Supply		
CodeS	CodeP	Amount
S1	P1	300
S1	P2	200
S1	P3	400
S1	P4	200
S1	P5	100
S1	P6	100
S2	P1	300
S2	P2	400
S3	P2	200
S4	P3	200
S4	P4	300
S4	P5	400

Products				
CodeP	NameP	Color	Size	Storehouse
P1	Sweater	Red	40	Amsterdam
P2	Jeans	Green	48	Den Haag
P3	Shirt	Blu	48	Rotterdam
P4	Shirt	Blu	44	Amsterdam
P5	Skirt	Blu	40	Den Haag
P6	Coat	Red	42	Amsterdam

EXAMPLE DB3: IMDB

- A subset of the schema and data from the IMDB.com website
  - Actors (person_100k), Movies (title_100k), and Actors in Movies (cast_info_100k)
  - Plus aliases, keywords, movie genres, etc.
- We will use MongoDB and Neo4J implementations of the same database (obviously, with different schemas)
- Get it (with import instructions) here
  - https://docs.google.com/document/d/1jj3cMAnk6Rc0mHkkOAIYDzYLjKisCuyj4-3KF9l-_8o



aka_name_100k

id	INTEGER
person_id	INTEGER
name	TEXT
imdb_index	CHARACTER VARYING(12)
name_pcode_cf	CHARACTER VARYING(5)
name_pcode_nf	CHARACTER VARYING(5)
surname_pcode	CHARACTER VARYING(5)

person_100k

id	INTEGER
name	TEXT
imdb_index	CHARACTER VARYING(12)
imdb_id	INTEGER
gender	CHARACTER VARYING(1)
name_pcode_cf	CHARACTER VARYING(5)
name_pcode_nf	CHARACTER VARYING(5)
surname_pcode	CHARACTER VARYING(5)

cast_info_100k

id	INTEGER
person_id	INTEGER
movie_id	INTEGER
person_role_id	INTEGER
note	TEXT
nr_order	INTEGER
role_id	INTEGER

char_name_100k

id	INTEGER
name	TEXT
imdb_index	CHARACTER VARYING(12)
imdb_id	INTEGER
name_pcode_nf	CHARACTER VARYING(5)
surname_pcode	CHARACTER VARYING(5)

role_type

id	SERIAL
role	CHARACTER VARYING(32)

movie_keyword_100k

id	INTEGER
movie_id	INTEGER
keyword_id	INTEGER

keyword_100k

id	INTEGER
keyword	TEXT
phonetic_code	CHARACTER VARYING(5)

movie_link_100k

id	INTEGER
movie_id	INTEGER
linked_movie_id	INTEGER
link_type_id	INTEGER

title_100k

id	INTEGER
title	TEXT
imdb_index	CHARACTER VARYING(12)
kind_id	INTEGER
production_year	INTEGER
imdb_id	INTEGER
phonetic_code	CHARACTER VARYING(5)
episode_of_id	INTEGER
season_nr	INTEGER
episode_nr	INTEGER
series_years	CHARACTER VARYING(49)

kind_type

id	SERIAL
kind	CHARACTER VARYING(15)

aka_title_100k

id	INTEGER
movie_id	INTEGER
title	TEXT
imdb_index	CHARACTER VARYING(12)
kind_id	INTEGER
production_year	INTEGER
phonetic_code	CHARACTER VARYING(5)
episode_of_id	INTEGER
season_nr	INTEGER
episode_nr	INTEGER
note	TEXT

link_type

id	SERIAL
link	CHARACTER VARYING(32)

WITH QUERIES

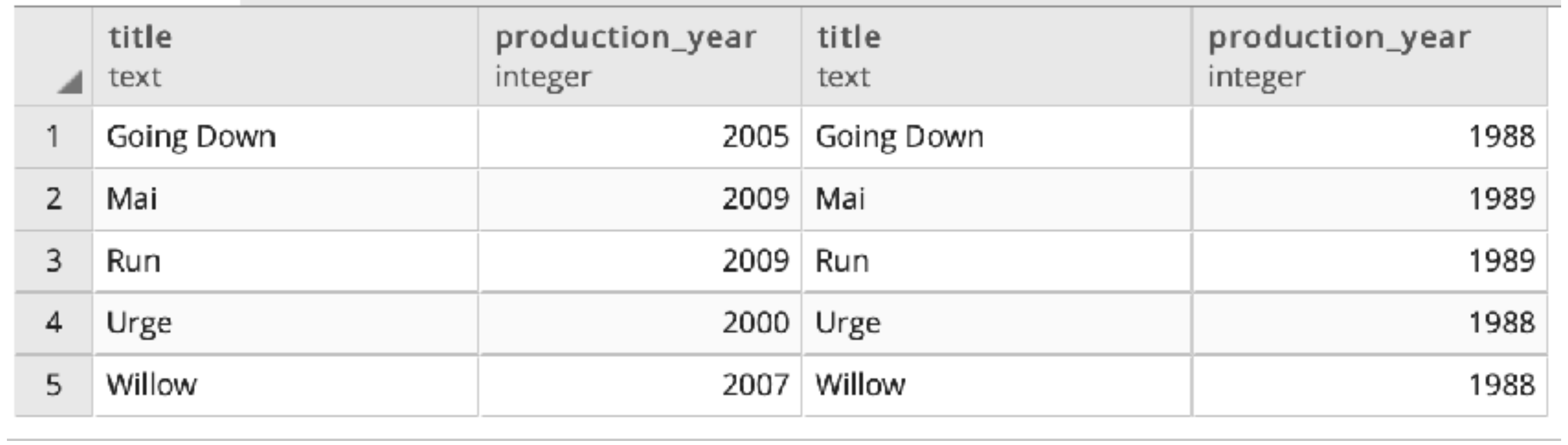
THE `WITH` CLAUSE

- A.k.a. **Common Table Expressions** (CTE)
- Allows the definition of a table (or multiple tables) that can be used only within a specific query
  - Similar to a `VIEW` (see later)
- But also the execution of `INSERT/UPDATE/DELETE` operations within the same query
  - `RETURNING` clauses give access to processed rows

EXAMPLES /1

- Retrieve pairs of titles from the 00s that have the same name of a title from the 80s

```
WITH NinetiesMovies AS(
    SELECT title, production_year
    FROM title_100k, kind_type
    WHERE production_year BETWEEN 1990 AND 1999
    AND title_100k.kind_id = kind_type.id
    AND kind = 'movie'
), EightiesMovies AS (
    SELECT title, production_year
    FROM title_100k, kind_type
    WHERE production_year BETWEEN 1980 AND 1989
    AND title_100k.kind_id = kind_type.id
    AND kind = 'movie'
)

SELECT *
FROM NinetiesMovies, EightiesMovies
WHERE NinetiesMovies.title = EightiesMovies.title
```



	title text	production_year integer	title text	production_year integer
1	Going Down	2005	Going Down	1988
2	Mai	2009	Mai	1989
3	Run	2009	Run	1989
4	Urge	2000	Urge	1988
5	Willow	2007	Willow	1988

EXAMPLES /2

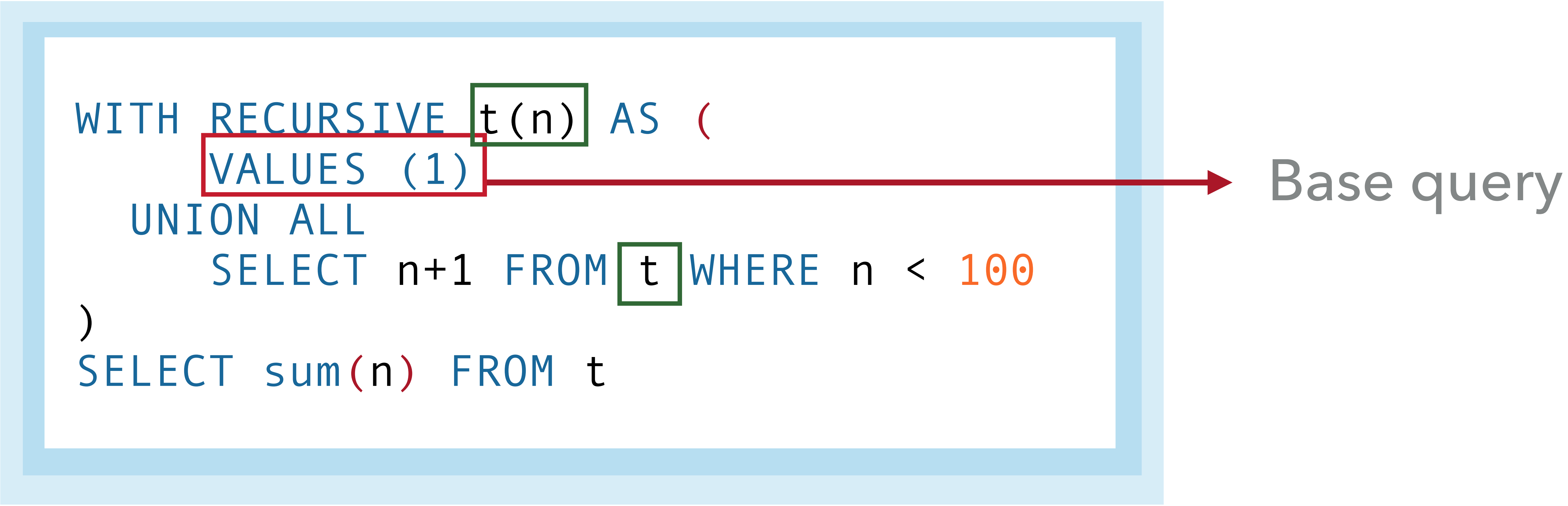
- Remove all aliases containing the name *Alessandro*, but copy them in a log table

```
create TABLE aka_name_log(
     like aka_name_100k
     including all
     including constraints
     including indexes
)
```

```
WITH moved_rows AS (
     DELETE FROM aka_name_100k
     WHERE name LIKE ‘%Alessandro%’
     RETURNING *
)
INSERT INTO aka_name_log
SELECT * FROM moved_rows
```

RECURSIVE QUERIES

- The WITH clause enables the definition of recursive queries
  - Useful to navigate recursive relationships



```
WITH RECURSIVE t(n) AS (
      VALUES (1)
  UNION ALL
      SELECT n+1 FROM t WHERE n < 100
)
SELECT sum(n) FROM t
```

Base query

EXAMPLE

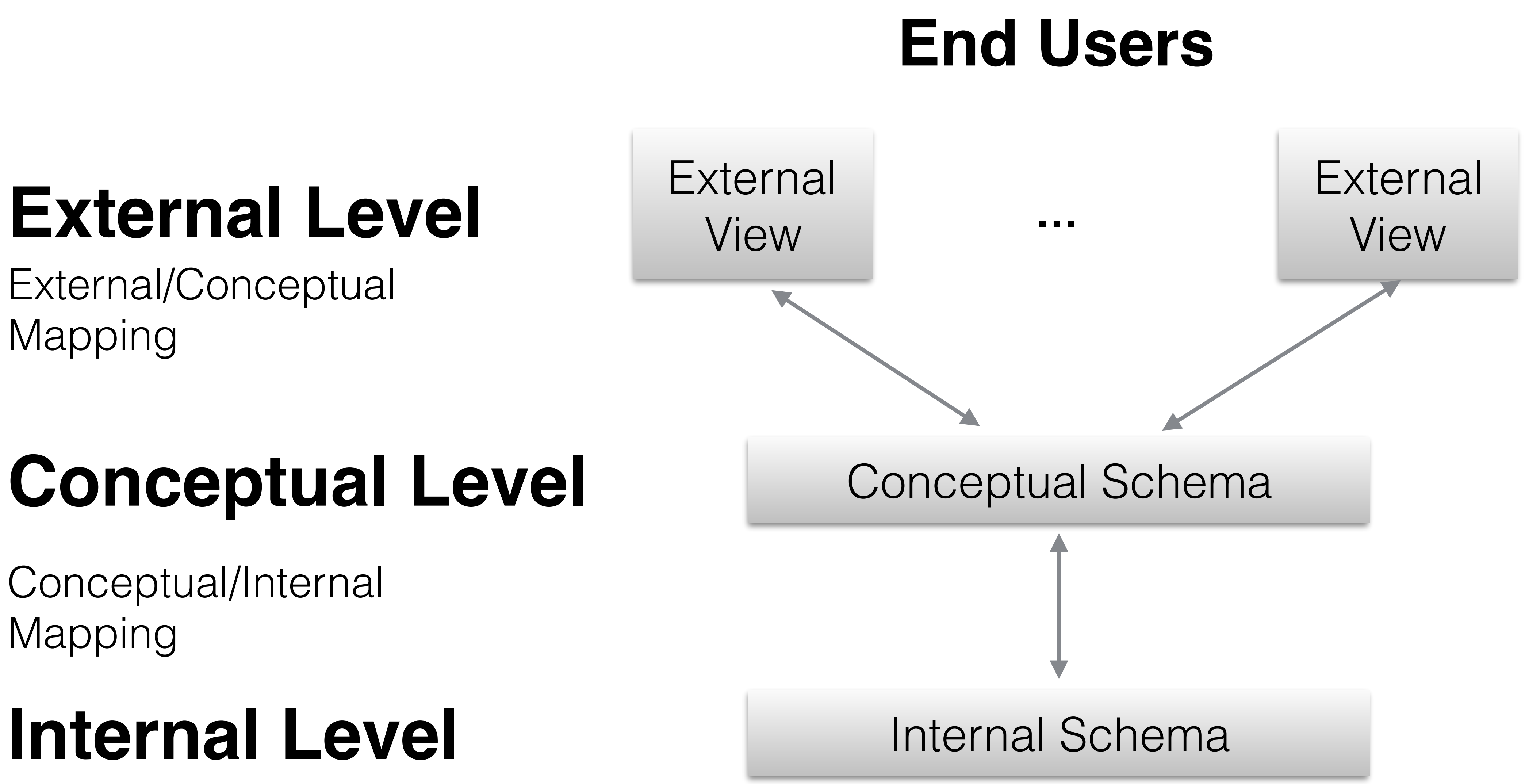
```
WITH RECURSIVE included_parts(sub_part, part, quantity) AS (
    SELECT sub_part, part, quantity
     FROM parts
     WHERE part = 'our_product'
  UNION ALL
    SELECT p.sub_part, p.part, p.quantity
     FROM included_parts pr, parts p
     WHERE p.part = pr.sub_part
)

SELECT sub_part, SUM(quantity) as total_quantity
FROM included_parts
GROUP BY sub_part
```

VIEWS

VIEWS (VIRTUAL TABLES) IN SQL

Any relation that is not of the conceptual model but is made visible to a user

- Single table derived **from other tables**
- Considered to be a **virtual** relation
- Always up-to-date
  - “Physically" or “logically”
  - The DBMS takes care of synchronisation



WHY VIEWS?

- Users **should not be allowed** to access the **actual** relations / data stored in the database
  - Limits on row values
  - Aliases for column or table names
- Favour the reuse of queries that are frequently used just to ***denormalize*** the logical schema
- Help simplifying complex queries, or enable the execution of queries that could not be expressed otherwise
  - **Decompose** the problem and produce a **more readable** solution
  - Combine and nest several **aggregate operators**

VIEWS DEFINITION

```
CREATE [TEMPORARY] VIEW ViewName [(AttributeList)]
 AS SelectSQL
  [WITH [CASCADED|LOCAL] CHECK OPTION]
```

- ViewName
  - Used by other queries / views as table name
- [(AttributeList)]
  - Used for projection / renaming / functions / operations
- SelectSQL
  - Any SELECT query (with some limitations)
- [WITH [CASCADED|LOCAL] CHECK OPTION]
  - Constrains inserts or updates to rows in tables referenced by the view
- [TEMPORARY]
  - Defines if the view should be dropped at the end of the current user session

VIEW DEFINITION EXAMPLES

Administration Employees

```sql
CREATE VIEW AdminEmployee (FirstName, Surname, Salary) AS
  SELECT FirstName,Surname,Salary
   FROM Employee
   WHERE Dept = ‘Administration’ AND Salary > 10
```

VIEW

Junior Administration Employees

```sql
CREATE VIEW JuniorAdminEmployee AS
  SELECT *
   FROM AdminEmployee
   WHERE Salary < 50
   WHICH CHECK OPTION
```

VIEW of VIEWs

USING VIEWS FOR QUERYING /1

- Find the departments with the highest salary expenditure

Without VIEWs

```
SELECT Dept
 FROM Employee
 GROUP BY Dept
 HAVING sum(Salary) >= ALL(SELECT Dept
                            FROM Employee
                            GROUP BY Dept)
```

With VIEWs

```
CREATE VIEW SalaryBudget(Dept,SalaryTotal) AS
  SELECT Dept, sum(Salary)
  FROM Employee
  GROUP BY Dept
```

```
SELECT Dept
 FROM SalaryBudget
 WHERE SalaryTotal =(SELECT max(SalaryTotal)
                            FROM SalaryBudget)
```

USING VIEWS FOR QUERYING /2

- Find the average number of offices per department

Incorrect Solution

```
SELECT avg(count(DISTINCT Office))
 FROM Employee
 GROUP BY Dept
```

SQL does not allow cascades of aggregate operators

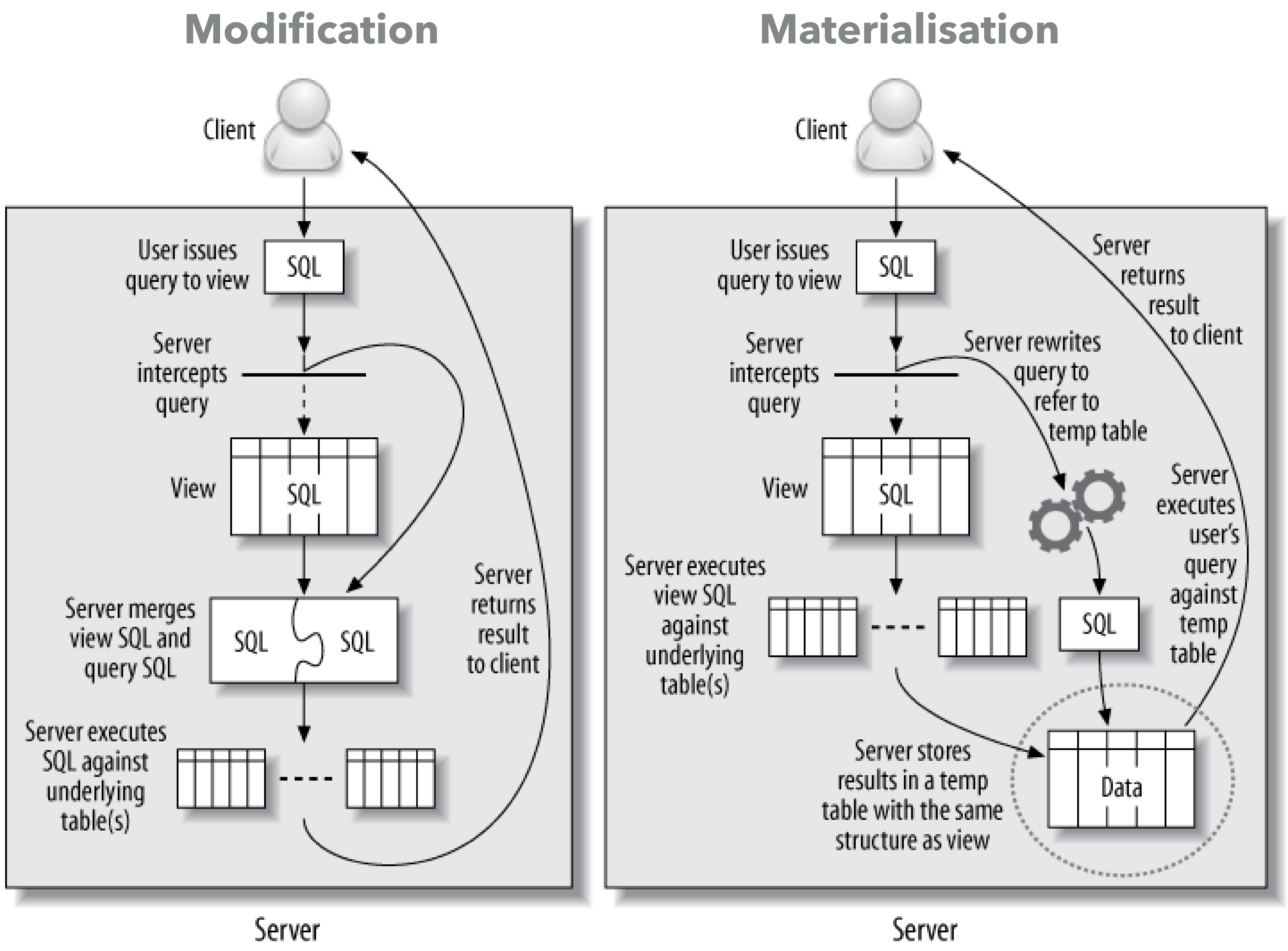
Correct Solution

```
CREATE VIEW DeptOffice(Dept,NoOfOffice) AS
  SELECT Dept, count(DISTINCT Office)
  FROM Employee
  GROUP BY Dept
```

```
SELECT avg(NoOfOffices)
 FROM DeptOff
```

VIEW EXECUTION APPROACHES

- **Query Modification**
  - **Modify** view query into a query on underlying base tables
  - Disadvantage: inefficient for views defined via complex queries that are time-consuming to execute
- **View Materialisation**
  - **Physically create** a temporary view table when the view is first queried
  - **Keep that table** on the assumption that other queries on the view will follow
  - Requires efficient strategy for **automatically** updating the view table when the base tables are updated


Modification
Materialisation
Client
User issues query to view
SQL
Server intercepts query
View
SQL
Server merges view SQL and query SQL
SQL
SQL
Server executes SQL against underlying table(s)
Server returns result to client
Server
Client
User issues query to view
SQL
Server intercepts query
Server rewrites query to refer to temp table
View
SQL
Server executes view SQL against underlying table(s)
SQL
Server executes user's query against temp table
Server returns result to client
Server stores results in a temp table with the same structure as view
Data
Server

Source: High Performance MySql, 3rd Edition. TEMPTABLE mechanism in MySQL works similarly to POSTGRESQL Materialised View extension, In the latter, materialised views are updated on demand

VIEW EXECUTION IN POSTGRESQL

- **Query Modification**
  - The query is run every time the view is referenced in a query
  - Standard mode
  - `TEMPORARY` option specify lifetime of view (current session)
- **Query Materialisation**: `CREATE MATERIALIZED VIEW`
  - Defines a physical table
  - Manual refresh of data
    - `REFRESH MATERIALIZED VIEW`
    - Can slow down/lock the DB
  - POSTGRESQL Extension

```
CREATE MATERIALIZED VIEW [CONCURRENTLY] ViewName
 AS SelectSQL
  [WITH [NO] DATA]
```

AUTOMATICALLY UPDATABLE VIEWS

Idea: Update the underlying base tables through CRUD operations on a VIEW

- INSERT, UPDATE or DELETE statement on the view are turned into corresponding statements on the base relation
- Possible when
  - VIEW defined on single table / VIEW
  - No WITH, DISTINCT, GROUP BY, HAVING, LIMIT, or OFFSET clauses at the top level
  - No set operations (UNION, INTERSECT or EXCEPT) at the top level
  - No aggregates or set-returning functions

Some updates can be ambiguous

```
CREATE VIEW Employee_info AS
  SELECT FirstName,Surname, City
  FROM Employee JOIN Department ON dept=DeptName
```

```
INSERT INTO Employee_info VALUES (‘Joe’,’Smith’,’Berlin’);
```

- Which department, if multiple departments in Berlin?
- What if no department is in Berlin?

AUTOMATICALLY UPDATABLE VIEWS

Idea: Update the underlying base tables through CRUD operations on a `VIEW`

- There might be a mix of updatable and non-updatable column
  - Updatable: simple reference to updatable column of the base relation
  - Non-updatable: error when `INSERT` or `UPDATE`
- If `WHERE` statement in the query
  - Only rows that are in the view can be deleted or updated
  - `INSERT` can be executed on the view, but rows might not be in it
- `CHECK OPTION`
  - Ensures that any rows changed through the view continue to match the view's `WHERE` clause after the changes
  - `LOCAL`: check only the conditions defined directly in the view
  - `CASCADE`: check also conditions of underlying base `VIEW`s

LIMITATIONS AND PERFORMANCE IMPLICATIONS

- User, system, or local variables are **not allowed** in the SQL `SELECT` statement
  - It is not possible to have parametric `VIEW`s
- If base table schema changes, `VIEW`s won't be valid anymore
  - There is no `VIEW` integrity constraint
- `VIEW`s can have bad performance
  - But sometimes better than an equivalent query that doesn't use a view (caching)
- `MATERIALIZED` views add overhead, and it's hard to predict how a view will impact performance
  - `VIEW`s of `VIEW`s can easily generate VERY complex execution plans

VIEWS VS. BASE TABLES

BASE TABLE	VIEW
Tuples always physically stored in database	Tuples do not necessarily exist in physical form
CRUD operations are always allowed	CRUD operations are not always allowed on views
Always possible to **define tables** of views	Can be used to create other views, but not in a mutually dependent way - Recursion is possible on same `VIEW`

VIEWS IN MONGODB AND NEO4J

- MongoDB has a `createView` command
  - Same function as SQL `VIEW`s
  - Read only
  - Also `VIEW`s of `VIEW`s
- Neo4J has no `VIEW`s

ASSERTIONS AND TRIGGERS

SEMANTIC / BUSINESS CONSTRAINTS

Constraints that cannot be directly expressed in the schemas of the data model

- Examples
  - *The salary of an employee should not exceed the salary of the employee's supervisor*
  - *The maximum number of hours an employee can work on all projects per week is 56*
- SQL provides two constructs (not supported by all systems)
  - `CREATE ASSERTION`
  - `CREATE TRIGGER`

ASSERTIONS

- Assertions permit the specification of constraints outside of table definitions
  - Useful in many situations (e.g., to express generic inter- relational constraints)
- An assertion associates a name to a check clause

```
CREATE ASSERTION assertion_name CHECK (condition)
```

The constraint is satisfied if *no combination of tuples* in that database violates it

- The condition **must hold TRUE** for **every database state** for the assertion to be satisfied
- The DBMS is responsible for the condition not to be violated

ASSERTION EXAMPLE

- The condition clause can contain any condition that can be specified in the `WHERE` clause of a `SELECT` query

- There must always be at least one tuple in table `EMPLOYEE`

```
CREATE ASSERTION AlwaysOneEmployee
    CHECK (1 <= (SELECT count(*)
                 FROM Employee)
         )
```

Not available in PostgreSQL

TRIGGERS

Triggers (Active Rules): rules that are automatically triggered by database **events**, and that initiate certain **action** if certain **conditions** are met

- A trigger has three logical components
- The triggering **event**
  - data operations, temporal events, external events
- The **condition** that determines wether the rule action should be executed
  - Optional
- The **action** to be taken
  - SQL statements, external programs, etc.

WHAT ARE TRIGGERS FOR?

- **Notify** users about **violations** of some constraints
  - E.g. a manager should be notified if an employee is having too much travel expenses
- Manage **advanced referential integrity** constraints not based on keys
  - E.g. avoid updates of tuple values based on values of other tuples
- Automatic **maintenance** of **derived** data
  - E.g. materialised views, data replication, etc.

GRANULARITY AND EXECUTION MODE

- Granularity
  - **Row-level**: the trigger is activated once for every tuple on which the event occurred
  - **Statement-level**: the trigger is activated once for every SQL statement, referring to all the tuples on which the statement operated (set-oriented)
- Execution mode
  - **Immediate**: right after (or even before) the event
    - **Before**: to perform actions prior to changes in the table. New /modified record can be changed
    - **After**: to perform actions after changes in the table. Record written in the table cannot be changed
  - **Deferred**: at transaction commit

TRIGGERS IN POSTGRESQL

Examples in book use ORACLE syntax
PostgreSQL complete syntax has more options

```
CREATE TRIGGER name {BEFORE|AFTER}{event [OR …]}
ON tableName
[FOR [EACH] {ROW|STATEMENT}]
[WHEN (condition)]
 EXECUTE PROCEDURE function_name (arguments)
```

`name`

- Unique for the table

`time`

- BEFORE
- AFTER

`event`

- INSERT/UPDATE/DELETE
- Multiple events can be specified using OR (a PostgreSQL extension of the standard)

`function_name(arguments)`

- The function to be executed (and input parameters)

`FOR EACH ROW`

- function_name is executed once for each of the affected records

`FOR EACH STATEMENT`

- function_name is executed once for any given operations

STORED PROCEDURES IN POSTGRESQL

```
CREATE FUNCTION name (arguments)
RETUNS returnType
local_declarations
function_body
```

- Programs stored in the database
    - Can be written in different imperative/declarative languages
    - PL/pgSQL is PostgreSQL language
- A trigger function
    - Takes no parameters
    - Return type is `TRIGGER`
- `OLD` and `NEW` keywords: allow to refer to the data before and after the activating event takes place
- It is possible to use variables and flow controls.

TRIGGER EXAMPLE (FROM WERKCOLLEGE 5)

- Automatically increase/decrease vote counts when votes are created/deleted

```
CREATE FUNCTION f_inc_votes()
RETURNS TRIGGER
LANGUAGE PLPGSQL;
AS $$
BEGIN
    UPDATE suggestions
     SET votecount = votecount+1
     WHERE id = NEW.suggestionid;
     RETURN NEW;
END
$$

CREATE TRIGGER inc_votes
                     AFTER INSERT ON votes
   FOR EACH ROW EXECUTE PROCEDURE
                            f_inc_votes();
```

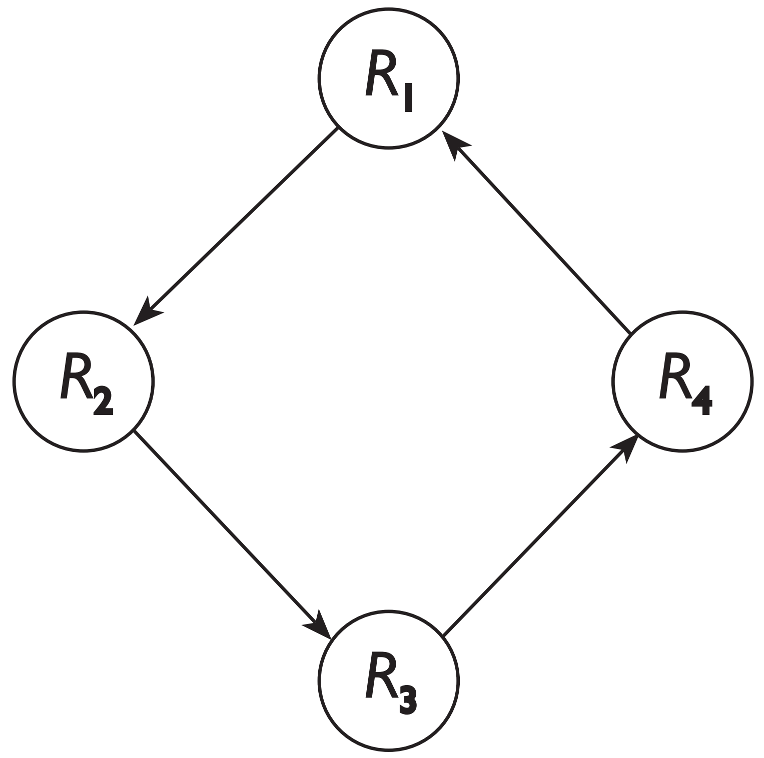
```
CREATE FUNCTION f_dec_votes()
RETURNS TRIGGER
LANGUAGE PLPGSQL;
AS $$
BEGIN
    UPDATE suggestions
     SET votecount = votecount-1
     WHERE id = NEW.suggestionid;
     RETURN OLD;
END
$$

CREATE TRIGGER dec_votes
                     AFTER DELETE ON votes
   FOR EACH ROW EXECUTE PROCEDURE
                            f_dec_votes();
```

EXTENSIONS (NOT USUALLY AVAILABLE)

- Boolean combinations of events
  - PostgreSQL has `OR`
- `instead of` clause
  - it is not executed the operation that triggered the event, but another one in its place
  - Available in PostgreSQL
- “Detached” execution mode: a separate transaction is started to manage the triggers
- Explicit user-defined priorities
- Rule sets, that can be made activated and deactivated with a single command

PROPERTIES OF TRIGGERS

- **Termination**
  - A rule set is guaranteed to terminate if, for any database state and initial modification, rule processing cannot continue forever
- **Confluence**
  - A rule set is confluent if, for any database state and initial modification, the final database state after rule processing is unique, i.e. it is independent of the order in which activated rules are executed
- Termination is assessed studying rules interaction
  - An important conceptual tool is the *triggering graph*



ADVANTAGES OF TRIGGERS

- Provide a **complementary**, and **more robust**, integrity checking mechanism to foreign keys
  - They can verify if foreign key **tuples** have certain characteristics
- Ability to **catch** business process **errors**
- Guarantee that for **every** change, the trigger is run
  - Control code in external application could not control other 3rd party changes
- Allow **scheduled tasks** directly in the DB
  - No cron or other system-level scheduler
- Simplify **propagation** of value **changes** from various tables
- Enable the execution of **calculation before** a row or value is inserted

DISADVANTAGES OF TRIGGERS

- They can’t replace ALL types of validation
    - No substitute for client-side validation
- **Hard to create**, debug and maintain
    - Few developer tools
    - It is VERY easy to create cyclical rules
        - **termination** and **confluence**
    - Bugs are difficult to locate in trigger chains
- Very heterogeneous functionalities and support across vendors

TRIGGERS IN MONGODB AND NEO4J

- MongoDB *Stitch* triggers allow for database events handling
  - Events: CRUD on documents, Authentication
  - Actions: Javascript functions

- Neo4J has no specific commands for triggers
  - a “TransactionEventHandler” API that could be used to implement custom Java functions that are executed when transactions are about to be committed

ACCESS CONTROL

ACCESS CONTROL

- Every component of the schema can be protected (tables, attributes, views, etc.)

The owner of a resource (the creator) assigns privileges to the other users

- A predefined **role** (`postgres`, or the name of the operating system user that initialised the cluster) represents the database administrator and has complete access to all the resources

- A privilege is characterised by:
    - The **resource**
    - The user who **grants** the privilege
    - The user who **receives** the privilege
    - The **action** that is allowed on the resource
    - Whether or not the privilege can be **passed on** to other users

MANAGING ROLES

- Role: a database user, or a group of database users
  - The `pg_roles` catalog contains the current list of roles
- The role determines the access privileges
  - Login
  - Database creation
  - Role creation

```
CREATE ROLE name
```

```
DROP ROLE name
```

```
CREATE ROLE role_name LOGIN
```

- Group roles can be used to grant and revoke privileges to multiple users
  - Group roles have no LOGIN access privilege

```
GRANT group_role TO role_name
```

POSTGRES ROLE AUTHENTICATION

- Trust
  - Any role name in the roles list can access
  - Not recommended for multi-user or networked machines
- Password-based
  - PostgreSQL database passwords are separate from operating system user passwords
- SSL Certificates
- Plus a number of external authentication protocols and systems
  - GSSAPI, SSPI, LDAP, RADIUS

VIEWS IN MONGODB AND NEO4J

- MongoDB
  - Role-based access control, similar to PostgreSQL
  - Privileges are granted at database and collection level
  - MongoDB starts with no authentication methods enabled
    - It supports internal, certificate-based, and protocol-based authentication

- Neo4J
  - Authentication and authorisation available only in the enterprise edition
  - Role-based access control
  - Privileges granted on the whole graph
  - Authentication with native, LDAP, and other authorisation provides

WRAPPING UP

TODAY WE COVERED

- The WITH clause
- Views
- Triggers
- Access Control Mechanism

END OF LECTURE