SD202: Databases

Advanced SQL and PostgreSQL features

Antoine Amarilli

Télécom Paris

Views

Table inheritance

Transactions and Concurrency

Views

Views: definition

- You can define a view to represent the result of a complex query:

```
CREATE VIEW Movie_with_actor AS
  SELECT DISTINCT Movie.id, title FROM Movie, Actor_in_movie
    WHERE Movie.id = Actor_in_movie.movie;
```

- View definition: simply a SELECT query as usual.
- You can then use the view as if it were a regular table

View: example

- Logical schema: Employee entity, every employee is either Secretary or Professor
- This is specialization (complete and disjoint)
- Physical schema: one Secretary table and one Professor table
- The Employee table is their union, projected on the common attributes
- Instead of storing it, we can define it with a view

Advantages of views

What are views good for?

- Logical independence: you can change the definition of the view in an application without changing the rest of the code
- Can be used to restrict access rights (only allow users to see a specific view)
- Can be switched easily to a materialized view for performance

Views can be a “fix” to address problems with the schema, or to redefine the logical schema from the physical one

Materialized views

```
CREATE MATERIALIZED VIEW Movie_with_actor ...
```

Must then be manually updated with:

```
REFRESH MATERIALIZED VIEW Movie_with_actor ...
```

How to make the view refresh **automatically**? Workaround:

- Make the materialized view a **regular table**
- Define **triggers** to update it in the right way whenever the underlying tables are changed

Example of maintaining a materialized view

- Logical schema: Employee entity, each employee is Secretary or Professor
- Physical schema: one Secretary table and one Professor table
- The Employee table is their union, projected on the common attributes
- How to reflect updates from Professor and Secretary?
    - When a tuple in inserted in either table, insert its projection in Professor
    - When a tuple is modified, also modify the projection
    - When a tuple is deleted, also delete it

  → We assume that no tuple in Employee corresponds to two tuples in Professor and Secretary (common key)

  → Question: can we accept updates to Employee? how to reflect them?
- Other common use case: maintaining an aggregate, e.g., a sum

Stored procedures

You can write custom procedures in PostgreSQL

```
CREATE OR REPLACE PROCEDURE transfer
  (origin INT, destination INT, amount DECIMAL)
LANGUAGE plpgsql
AS $$
BEGIN
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
UPDATE Account SET balance = balance - amount WHERE id = origin;
UPDATE Account SET balance = balance + amount WHERE id = destination;
COMMIT;
END; $$
```

Also possible to write custom functions, custom aggregation operators...

Why use stored procedures

- For **triggers** (see later)
- To factor some application logic **in the database** for consistency across applications
- For **performance** (execute code closer to the data)
  - Stored procedures can be written **in C**

Triggers

- Procedures can be created as **triggers** to be automatically run whenever data is changed
  - Whenever a **table** is modified
  - For **every modified tuple** in a table
  - Can be run **before** or **after** the operation or **instead of** the operation
- Possible uses:
  - Complex **consistency check**, or **normalization**/**reformatting**
  - Recomputing **auxiliary tables**, automatically creating **dependent data**
  - Manually updating an **aggregate** (e.g., a sum)
  - Manually **log** database operations

Views

Table inheritance

Transactions and Concurrency

Table inheritance

Table inheritance

You can define tables that refine another table (inherit from it)

```
CREATE TABLE Employee (id SERIAL PRIMARY KEY, name VARCHAR, salary INT);
CREATE TABLE Professor (field VARCHAR) INHERITS (Employee);
CREATE TABLE Secretary (building VARCHAR) INHERITS (Employee);
INSERT INTO Employee(name, salary) VALUES ('John', 424);
INSERT INTO Professor(name, salary, field) VALUES ('Patricia', 343, 'CS');
INSERT INTO Secretary(name, salary, building) VALUES ('Simon', 252, 'A');
SELECT * FROM Professor;
SELECT * FROM Secretary;
SELECT * FROM Employee;
SELECT * FROM ONLY Employee;
```

Table inheritance subtleties

- Tables can inherit from multiple tables
- Deleting a parent table **cascades** to the tables that inherit from it
- **Warning:** uniqueness constraints and keys do not take inheritance into account!

```
INSERT INTO Professor(id, name, salary, field) VALUES
    (3, 'Paula', '454', 'CS');
SELECT * FROM Employee;
-- id's are no longer unique!
```

- **Warning:** inserting in a “parent” table does not work

```
-- This does not work
INSERT INTO Employee(name, salary, field) VALUES
    ('Priscilla', '4242', 'CS');
```

Views

Table inheritance

Transactions and Concurrency

Transactions and Concurrency

Reminder on ACID

SQL guarantees the ACID properties:

- Atomicity: a transaction block is either completely executed or not executed at all
- Consistency: the database always satisfies the integrity constraints
- Isolation: if there are multiple transactions, they happen as if one had taken place before the other
- Durability: one executed, transactions will not be lost

Transactions

- Default: every query (`SELECT`, `INSERT`, etc.) is a transaction
- We can manually define a transaction block with `BEGIN ... COMMIT`
- Start a transaction with `BEGIN`, and issue queries
- To perform the transaction, use `COMMIT`
- To abort the transaction, use `ROLLBACK`
- To define a savepoint, use `SAVEPOINT label`
- To roll back to a savepoint, use `ROLLBACK TO SAVEPOINT label`

Exercise: Can you think of a use case for transactions?

Challenges with single transactions

To correctly support transactions (one at a time) we must:

- Prepare the effects of the transaction, and atomically commit them
- Make sure the commits are durable, even if the hardware fails
- Be able to revert the effects of the transaction
- With save points, be able to revert its partial effects

Challenges with concurrent transactions

- **Transactions:** a sequence of read/write database operations
- These transactions are not **ordered** a priori (e.g., one may arrive while another is running)
- We want to execute them in parallel for **performance**
- Problems:
  - Two transactions can access the **same data item at the same time**
  - Even if individual operations do not conflict, the **sequence of operations** of a transaction may be affected by other transactions
- Strongest **ACID** guarantees: **serializability**

  → What will happen is **consistent** with a **serial ordering** of the transactions

  → **Challenge:** Parallelize as much as possible while respecting this

Concurrency

- Satisfying serializability is complicated and may cause transactions to:
    - wait for another transaction to complete, possibly deadlock
    - fail if we have started to execute it, but another transaction affected its data
- PostgreSQL supports several transaction isolation levels relaxing serializability
- Each level describes which kinds of anomalies may take place
- More restrictive isolation means:
    - worse performance
    - more failures, but
    - less inconsistency problems
- Also supports explicit locking in transactions (in addition to these mechanisms)

Replication and clustering

- Having **more than one server** has several uses:
  - **partition** the data if it is large
  - do **load balancing** to use multiple servers
  - evaluate a query on multiple servers **in parallel**
  - have **failover** servers for high availability
- PostgreSQL has some support to propagate changes from a main database to read-only **failovers**
- PostgreSQL **did not focus** initially on replication and clustering
- Many **third party** solutions