Databases

Lecture 9 - Views and Transactions

Emily Lucia Antosch

HAW Hamburg

18.03.2025

Contents

1. Introduction

1.1 Where are we right now?

- Last time, we looked at the basics of subqueries and views
- Today, we'll be discussing
  - how we can expand our knowledge of views,
  - how we can use transactions to increase the safety of our data manipulation statements
  - how transactions are executed.

1.1 Where are we right now?

1. Introduction
2. Basics
3. SQL
4. Entity-Relationship-Model
5. Relationships
6. Constraints
8. **Subqueries & Views**
9. **Transactions**
10. Database Applications
11. Integrity, Trigger & Security

1.2 What is the goal of this chapter?

- At the end of this lesson, you should be able to
  - create views in PostgresQL and use them effectively and
  - use transactions to make safe changes, that can be undone if necessary.

2. Repetition

Updating Views

- Views are Relations, just like tables!
- Should make no difference to users

? Question

Can we modify the view's data?

- Depends on type of view!

2.1 Views

Basics

- Classify views based on the select:
  - **Projection View**
    - `SELECT a, b, c ...`
  - **Selection View**
    - `... WHERE <condition> ...`
  - **Join View**
    - `FROM tab_a JOIN tab_b ...`
    - **Aggregation View**
      - `SELECT MAX(x) ...`
- Other types and combinations exist

2.1 Views

Basics

- A view with a single defining table is updatable if
  - the view attributes contain the primary key of the base relation,
  - as well as all attributes with the NOT NULL constraint that have a default value specified
- Views defined on multiple tables using joins are only updatable in special cases
  - E.g., INSERT and UPDATE for Join Views, if join condition is based on PK-FK
- Views defined using grouping and aggregate functions are not updatable

2.1 Views

Generated Tables

```sql
CREATE TABLE <name> AS SELECT
```

SQL

- Can create new table based on query
- New table is independent from old table
- Use cases:
  - Copy table
  - Copy parts of table

Memorize

New table does not have all constraints of the parent table!

2.1 Views

Generated Tables

Example

```sql
INSERT INTO Underpaid ( lname , fname )
  SELECT lname , fname **FROM** Employee WHERE salary < 1000 ;
```

- `WHERE` clause belongs to `SELECT`

2.2 Transactions

Operations

- A transaction bundles several operations into one logical unit
  - Unit of Work
- Includes one or more database access operations E.g., `INSERT`, `DELETE`, `UPDATE`, `SELECT`
- Operations must be executed all or none
- Example: Order a hotel room over the internet
  - Choose and reserve room
  - Payment
  - Final booking of the hotel room

2.2 Transactions

ACID

- Key features of transactions
  - **Atomicity**: Transaction is executed in whole or not at all
  - **Consistency**: State of the DB is consistent before and after a transaction
  - **Isolation**: Transactions do not interfere with other concurrent transactions
  - **Durability**: Changes are stored permanently in the database and will not get lost

2.2 Transactions

ACID - Atomicity

- Begin of Transaction (**BoT**)
  - SQL99: START TRANSACTION
  - PostgresQL:

```sql
BEGIN;
```

- Commit a transaction: `COMMIT;`
  - All operations are made persistent
  - All changes are visible to other users
- Rollback transaction: `ROLLBACK;`
  - DB is in state at **BoT** again

2.2 Transactions

ACID - Consistency

- DB: in consistent state before transaction Also, in consistent state after transaction
- Integrity constraints assure that
- Constraints can be defined as
  - `IMMEDIATE` (default in mySQL)
    - are checked immediately after operation
  - `DEFERRED`
    - Check at time of commit

2.2 Transactions

ACID - Isolation

- Transactions are isolated from other concurrent transactions
- Concurrent transactions shall behave well

2.2 Transactions

ACID - Isolation: Concurrency Control

- Concurrent operations can lead to problems
  - Lost Update
  - Dirty Read
    - Unrepeatable read
    - Phantom tuples

2.2 Transactions

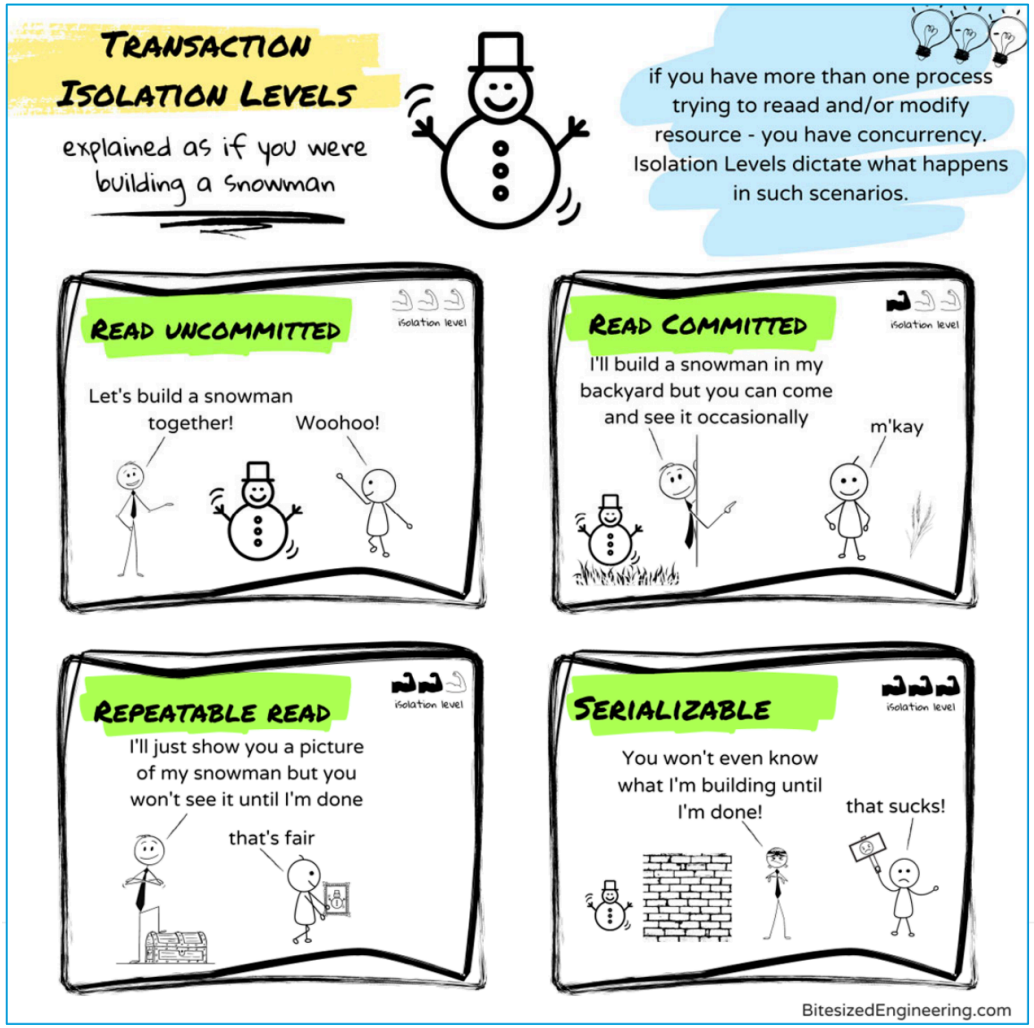
ACID - Isolation: Concurrency Control

- Lost Update is prevented by SQL
- Transactions: may choose **Isolation Level**
  - `SERIALIZABLE`
    - no problems
    - `REPEATABLE READ` (default in mySQL)
      - Open for phantom tuples
    - `READ COMMITTED` (default in Oracle, SQL Server)
      - Open for phantom tuples and unrepeatable read
    - `READ UNCOMMITTED`
      - Open for all problems

2.2 Transactions

ACID - Isolation: Concurrency Control

2.2 Transactions

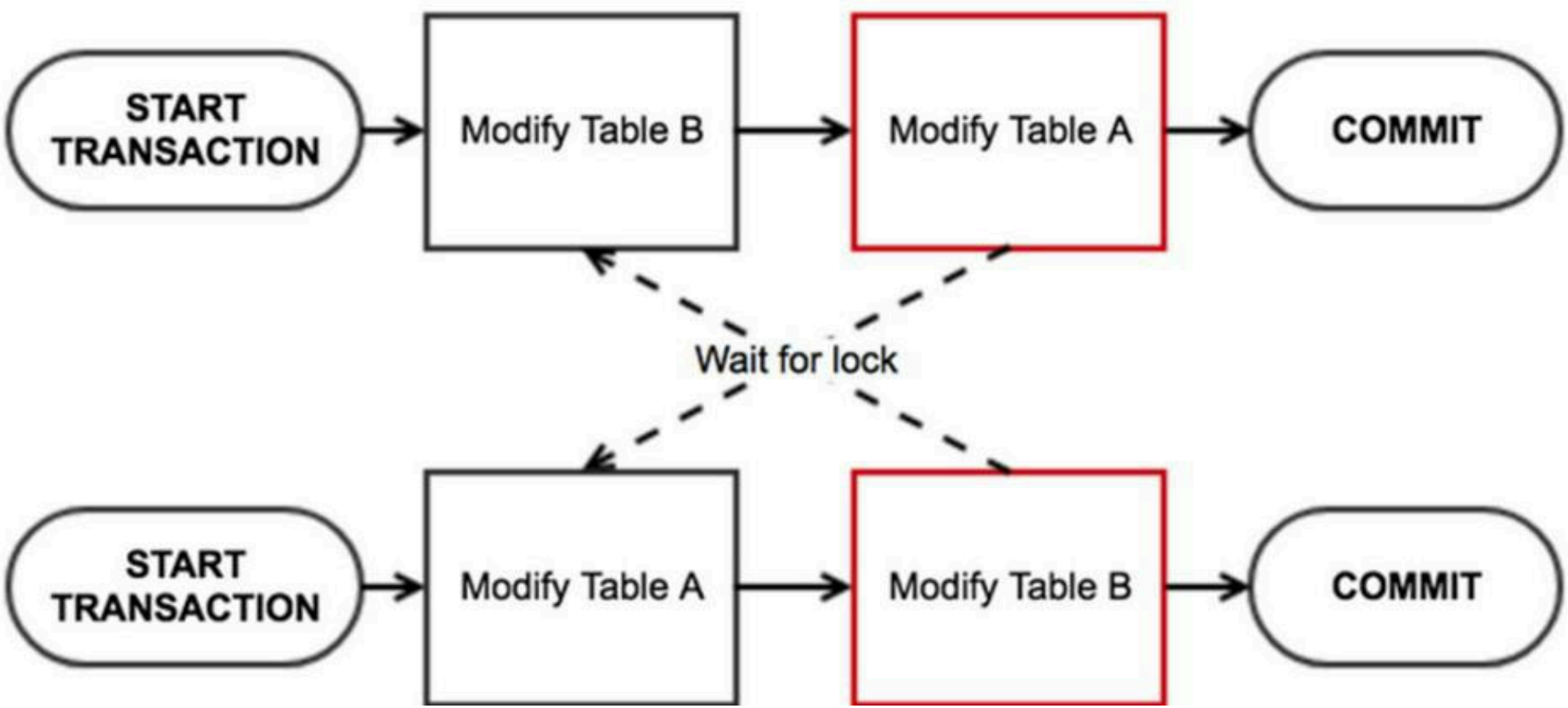


2.2 Transactions

ACID - Isolation: Concurrency Control

- Deadlocks may occur!
  - Usually are resolved automatically by aborting one transaction

2.2 Transactions



2.2 Transactions

ACID - Durability

- Once committed, changed data is safe
- Error types
  1. Computer failure
  2. Transaction or system error (constraint violation, $\frac{x}{0}$, blackout, system crash)
  3. Local Errors
  4. Concurrency control enforcement
  5. Disk error (harddisk broken)
  6. Physical problems and catastrophes (fire, earthquake, robbery, …)

ACID - Durability: Error Handling

- Recovery from transaction failures usually means that the database is **restored** to the most recent consistent state just before the time of failure
- Minor damages due to error types 1-4 from slide “ACID – Durability”
  - DBMS provides handling
    - Recovery strategy is to identify any changes that may cause an inconsistency in the database
      - Changes are first written to redo logs (files on disk)
      - Written to database files after commit

ACID - Durability: Error Handling

- Extensive damage due to error types 5-6 from slide “ACID – Durability”
  - Recovery handling restores a past copy of the database from archival storage
  - Reconstructs a more current state by redoing the operations
  - Last transactions are lost!
- Solution: Redundancy
  - RAID (**r** edundant **a** rray of **i** ndependent **d** isks)
    - Data Replication by DBMS

2.2 Transactions

ACID - Durability: Error Handling

- Changes are performed on (replicated to) several database instances
- Master/Slave
  - Updates only on one instance (master)
  - Slave: Read only vs. Standby
- Multi-Master
  - Updates on different instances
  - Needs conflict resolution strategy

2.2 Transactions

ACID - Durability: Error Handling

- **Synchronous**
  - Transaction valid only when committed on all DBs
  - Safest, but performance impact
  - May reduce availability of the system
- **Asynchronous**
  - Transaction valid when committed locally

2.2 Transactions

ACID - Durability: Error Handling

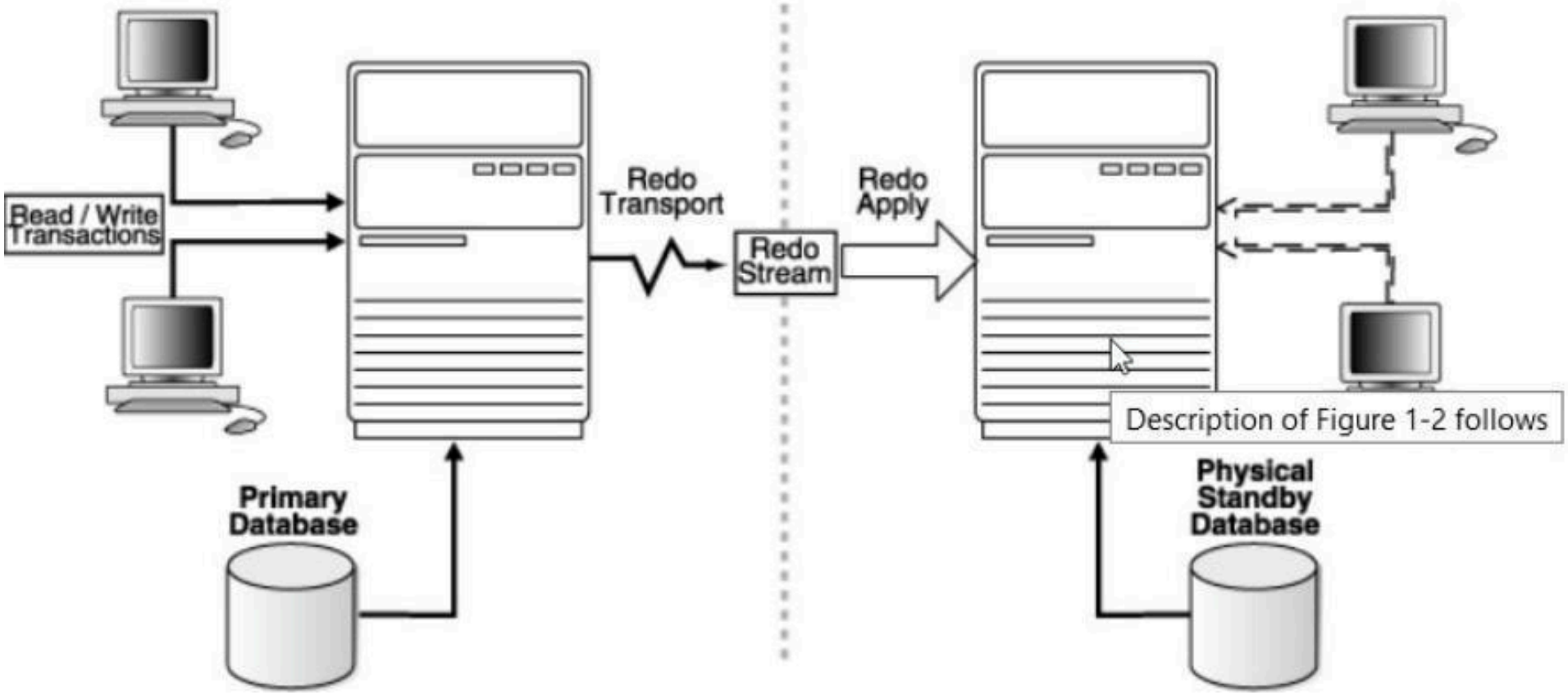
- Low level (disk device)
- Trigger based
  - Update triggers the replication (SQL level)
- Logfile shipping
  - Changes are stored in redo logs (as usual)
  - redo logs are copied to standby DB

2.2 Transactions

ACID - Durability: Error Handling

- Oracle
  - Data Guard
    - Replication on second server, can be used to answer Read-Only queries

2.2 Transactions

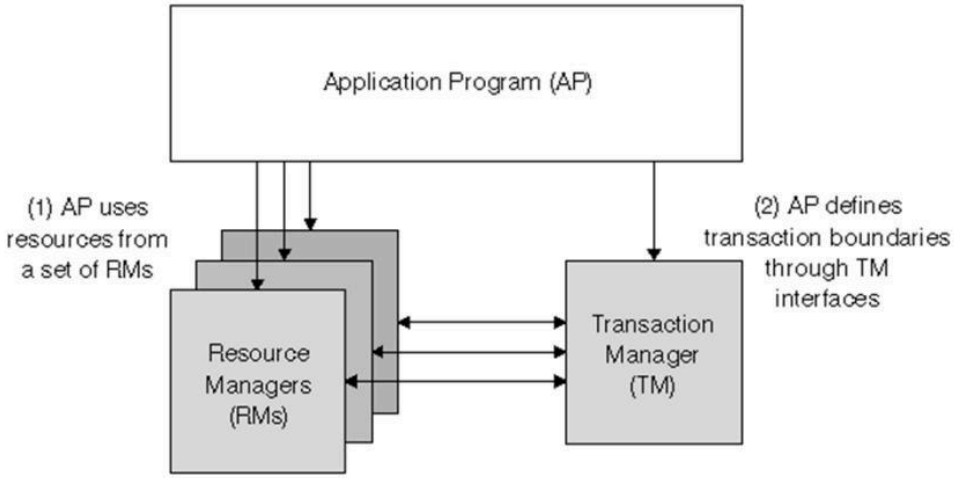


- Real Application Cluster (RAC)
  - Several servers share the same DB

2.2 Transactions

Distributed Transactions

- Transactions not only in a single DBS
- Standardized by X/Open
  - Transaction Manager: A software component that guarantees transaction properties
  - Resource Manager: Every resource (e.g., DBS, GUI) that is able to work in a



transactional mode without providing a transaction control structure itself

- The Transaction manager coordinates the Resource Manager that take part in the transaction. E.g., different DBS (distributed transactions) that appear as one DBS from outside (transparency!)

2.2 Transactions

Distributed Transactions

2.2 Transactions

Savepoints

- There are operations that may be expensive to execute time consuming
- If certain constraints fail within transaction execution, then maybe these constraints may not fail in a second attempt (e.g., time dependent)
- So “fall back” points can be defined, which are called **savepoints**
- It is possible to rollback up to a savepoint and restart transaction execution from this point on

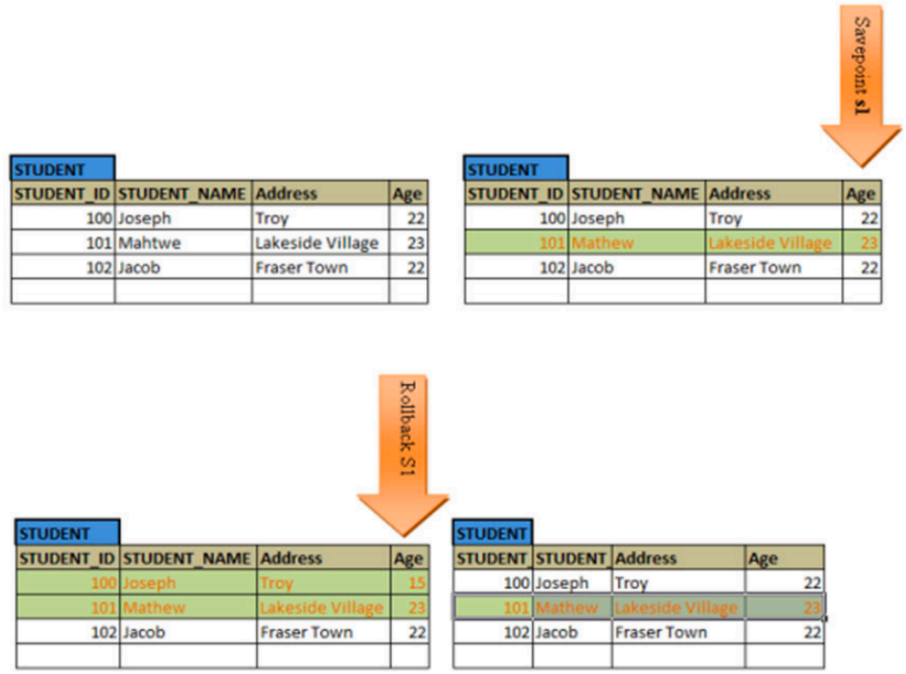
2.2 Transactions

Savepoints

2.2 Transactions

Code

```
UPDATE STUDENT SET STUDENT_NAME = 'Mathew' WHERE STUDENT_NAME = 'Mahtwe';
SAVEPOINT S1;
UPDATE STUDENT SET AGE = 15 WHERE STUDENT_ID = 100;
ROLLBACK to S1;
```



3. Integrity, Trigger and Security

Integrity Constraints

- Static Constraints
  - Conditions on states
  - Conditions must be fulfilled before and after operations
  - Used until now
    - Primary Key
    - Foreign Key
    - `UNIQUE`, `NOT NULL`, `CHECK`
- Dynamic Constraints (**Assertions**)
  - Integrity conditions that affect multiple tables
  - Conditions on state transitions

Example

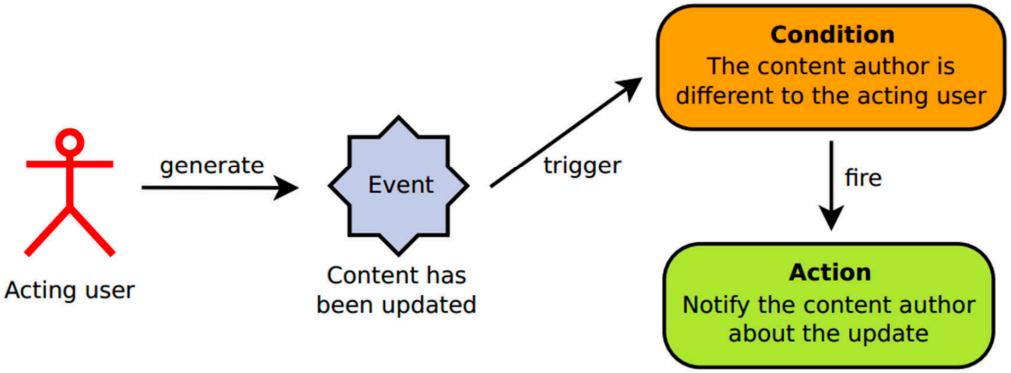
status of order ⟶ new ⟶ payed ⟶ processing ⟶ shipped

Integrity Constraints

- Assertions have been part of the SQL since SQL-92 (DDL)
- Not supported by most DBMS (e.g., MySQL, Postgres and Oracle)
- If the concept of assertions is to be simulated `TRIGGER`
- Concept:
  - Whenever anything is modified in the database, the assertion checks its condition
  - If the `SELECT`-statement gives a non-empty result, the operation that has triggered the assertion is denied

Integrity Constraints - ECA

- ECA rules
  - on event (E)
  - under certain conditions (C)
  - perform actions (A)



Trigger Syntax

SQL

```sql
CREATE
    [DEFINER = user]
    TRIGGER trigger_name
    trigger_time trigger_event
    ON tbl_name FOR EACH ROW
    [trigger_order]
    trigger_body
trigger_time: { BEFORE | AFTER }
trigger_event: { INSERT | UPDATE | DELETE }
trigger_order: { FOLLOWS | PRECEDES } other_trigger_name
```

Excursion Delimiter

- A PostgresQL client program such as pgadmin or psql program uses the delimiter (“;”) to separate statements and executes each statement separately
- However, a stored procedure consists of multiple statements separated by a semicolon (“;”)
- If you use a PostgresQL client program to define a stored procedure that contains semicolon characters, the PostgresQL client program will not treat the whole stored procedure as a single statement, but many statements.

- Therefore, you must redefine the delimiter temporarily so that you can pass the whole stored procedure to the server as a single statement.
- To redefine the default delimiter, you use the delimiter command

Excursion Delimiter

- In short: A delimiter is a separator between commands

Example

```sql
delimiter |
…
|
delimiter ;
```

In the code block between “delimiter” and “delimiter;” the delimiter is changed to “|” (instead of “;”)

Excursion Delimiter: Example

SQL

```
delimiter |
CREATE TRIGGER SALARY_VIOLATION
BEFORE INSERT ON EMPLOYEE
FOR EACH ROW
    BEGIN
        IF NEW.SALARY > (SELECT SALARY
                         FROM EMPLOYEE
                         WHERE SSN = NEW.SUPER_SSN )
        THEN SET NEW.Salary = (SELECT SALARY
                               FROM EMPLOYEE
```

```
11                                          WHERE SSN =
   NEW.SUPER_SSN )-1;
12          END IF;
13  END;
14  |
15  delimiter;
```

Events

- Triggers can react on events
  - DML: `INSERT`, `UPDATE`, `DELETE`
    - Most common trigger types
  - DDL: `CREATE`, `ALTER`, `DROP`
  - DB: startup, shutdown, logon of a user
- No `COMMIT` triggers

Types

- Time of execution, relative to event
  - `BEFORE`
  - `AFTER`
- `INSTEAD OF`
- Statement trigger
  - Once per statement
  - Even if no row is affected!
  - Default trigger type
- Row trigger

3.1 Basics

- For every affected row
- Syntax: `FOR EACH ROW`

Order of Trigger

- Before Statement Trigger (once!)
- For every row affected:
  - Before row trigger
  - DML operation
  - Immediate integrity checks
  - After row trigger
- After Statement Trigger (once!)

Transition Variables

- Row triggers can access old and new tuples
  - PostgresQL
    - `:old` or `old` ⟶ `NULL` for `INSERT`
    - `:new` or `new` ⟶ `NULL` for `DELETE`
- Oracle
  - `NEW` and `OLD`
  - Before row triggers:
    - Can even modify new!

Use Cases

- Constraints on state transitions
- Audit
  - When was a record last modified?
- Integrity checks with error correction
  - Change `:new`
- Maintain redundant data
- Updateable views
  - `INSTEAD OF`

Trigger: Example

- Audit insertion of new persons

```sql
DROP TRIGGER IF EXISTS emp_insert;
CREATE TRIGGER emp_insert AFTER INSERT ON employee
FOR EACH ROW
INSERT INTO EMPLOYEE_LOG (ESSN, INSERT_DATE) VALUES
( NEW.ssn , NOW() ) ;
```

Trigger: Example

- Salary of new persons

```sql
delimiter |
CREATE PROCEDURE output
   (in ssn char(9), in old_sal DECIMAL(10,2),
    in new_sal DECIMAL(10,2), in diff_sal DECIMAL(10,2))
BEGIN
   INSERT INTO EMPLOYEE_SALDIFF VALUES ( ssn , old_sal , new_sal, diff_sal);
END
|
```

```
9  delimiter;
```

Problems

- Cascading triggers
  - Trigger actions cause other triggers to fire
- Execution order
  - Result of high-level operation must be independent hereof!
- “Mutating Tables”

Problems

- Hard to implement
  - Transaction save!
  - Multi-session save
- Hard to debug
  - Update may lead to insert in another table
  - … can cause for example constraint violation
  - Which statement failed?

Assignment: Webshop

3.1 Basics

- Suppose the following relations in your database
- In the table `Price_History` we want to track on how the prices of the products of table Product develop over time. Table `Price_History` has four attributes:
  - The record ID `PHID`
  - The reference to table Product with the foreign key `PID`
  - The current price `Price`

Table `Product`

PID	Price	Description
1	0.50	red apple
2	0.60	green apple
3	1.20	red pepper
4	1.10	green pepper
…	…	…

Table `Product_History`

PHID	PID (FK)	Price	Change_Date
1	1	0.50	02.06.2021
2	3	1.20	02.06.2021
3	2	0.60	03.06.2021
4	4	1.10	04.06.2021
…	…	…	…

- The date `Change_Date`, where we store the date of the change

Assignment: Webshop

3.1 Basics

1. `INSERT` trigger: We want to get an `INSERT` with the current (start) price in table `Price_History` when we do an `INSERT` in the table `Product`. This is triggered when an `INSERT` on our table product is done (`AFTER`).
2. `DELETE` trigger: Furthermore, in case of a `DELETE`, all records of the deleted product in the table `Price_History` should be deleted as well.

Table `Product`

PID	Price	Description
1	0.50	red apple
2	0.60	green apple
3	1.20	red pepper
4	1.10	green pepper
…	…	…

Table `Product_History`

PHID	PID (FK)	Price	Change_Date
1	1	0.50	02.06.2021
2	3	1.20	02.06.2021
3	2	0.60	03.06.2021
4	4	1.10	04.06.2021
…	…	…	…

3. `UPDATE` trigger: If a price of a product is changed, this change should also result in an entry in the table `Price_History`.

Permissions

- DBMS are multi-user systems
- You need permissions to do anything with the DB:
  - login
  - `CREATE` table, `DROP` table, etc.
  - `SELECT`
  - `INSERT`, `UPDATE`, `DELETE`
- Permissions can be `GRANT`ed and `REVOKE`d

GRANT and REVOKE

- Permissions can be GRANTed and REVOKEd

```SQL
GRANT <privilege_name> ON <object_name>
TO { <user_name> | PUBLIC | <role_name>} [ WITH GRANT OPTION ] ;
```

- GRANT

```SQL
GRANT SELECT ON tab_a TO user_a;
GRANT UPDATE ON tab_b TO user_a;
```

- REVOKE

3.1 Basics

```sql
REVOKE SELECT ON tab_a FROM user_a;
```

SQL

Least Privilege Principle

- A user should have exactly the permissions necessary to do the work
  - … and not more!
- Important for web applications
  - anonymous end users
  - not trustworthy
- Limit the possible damage of attacks

Assignment: Webshop

1. Create a user student which is allowed to query and insert the table Product.
2. Revoke the insert privilege from a user student.

4. License Notice

4.1 Attribution

- This work is shared under the CC BY-NC-SA 4.0 License and the respective Public License.
- https://creativecommons.org/licenses/by-nc-sa/4.0/
- This work is based off of the work by Prof. Dr. Ulrike Herster.
- Some of the images and texts, as well as the layout were changed.
- The base material was supplied in private, therefore the link to the source cannot be shared with the audience.