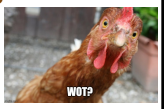
Transactions in SQL

Queries and Transactions

- Queries: requests to the DBMS to retrieve data from the database
- Updates: requests to the DMBS to insert, delete or modify existing data
- Transactions: logical grouping of query and update requests to perform a task
  - Logical unit of work (like a function/subroutine)

The chicken and the egg problem…

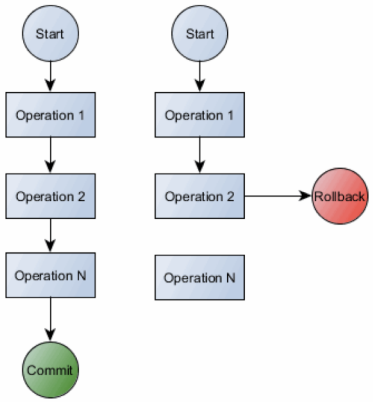
```
CREATE TABLE Chicken (ID INT PRIMARY KEY,
   eID INT NOT NULL REFERENCES Egg(ID));
CREATE TABLE Egg(ID INT PRIMARY KEY,
   cID INT NOT NULL REFERENCES Chicken(ID));
```



- Do we know commands that could create these tables?
- But how can we insert values into either table??
  - Need to treat two inserts into both tables as one logical unit of work…

SQL TRANSACTIONS

- Start:
  - Each SQL statement *implicitly* start a transaction, unless one is active.
  - Multi-SQL statement transaction is within
    BEGIN;
    …
    END;
- COMMIT ;
- ROLLBACK default action



Standard SQL TRANSACTIONS

- SET TRANSACTION [transaction characteristics];
  - It does not start a transaction
  - It can be invoked between transactions to set the next transaction to be activated
- Transaction characteristics: READ WRITE | READ ONLY
  - SQL1: DECLARE TRANSACTION [READ WRITE | READ ONLY];
  - SQL2/SQL3: SET TRANSACTION [READ WRITE | READ ONLY];
- SQL3 introduced START TRANSACTION [transaction characteristics];
  - It starts a transaction explicitly if one is not active
- COMMIT ;
- ROLLBACK; -- ROLLBACK default action

SQL transactions in PostgreSQL

- Basic transaction statements (SQL3 plus):
  - START TRANSACTION [READ WRITE | READ ONLY];
  - BEGIN [TRANSACTION] [READ WRITE | READ ONLY];
    - … should be unnecessary according to the SQL standard. Each SQL statement should implicitly start a transaction
  - COMMIT: Unless START TRANSACTION is issued, PostgreSQL implicitly issues a COMMIT after each SQL statement
    - This functionality is sometimes referred to as *autocommit*
- You cannot effectively have a multi-statement transaction without issuing a START TRANSACTION

Transaction Atomicity (Atomic Block)

- "All or nothing" can be achieved with *begin-end block*
- Consider a transaction:

```
begin;
   insert into Student values (23, 'John', 'CS');
   insert into Dept values ('CS', 501);
end;
```

- What happens if the first insert fails, e.g., due to a referential constraint violation?
  - Is the new tuple inserted into Department? No/Yes?
- The transaction fails and both inserts are aborted!!!

Modes of Constraints Enforcement

- **NOT DEFERRABLE** or **IMMEDIATE**
  - Evaluation is performed at input time
  - By default constraints are created as NON DEFERRABLE
  - It *cannot* be changed during execution
- **DEFERRED**
  - Constraints are not evaluated until commit time
- **DEFERRABLE**
  - It can be changed within a transaction to be DEFERRED using SET CONSTRAINTS
- Modes can be specified when a table is created.
  - INITIALLY IMMEDIATE: constraint validation to happen immediate
  - INITIALLY DEFERRED: constraint validation to defer until commit

Specifying Initial Eval. Mode inTables

```
CREATE TABLE dept(
    dno VARCHAR(50),
    did INTEGER,
    CONSTRAINT dept_PK PRIMARY KEY (dno));

CREATE TABLE student(
    sid  INTEGER,
    name VARCHAR(50),
    dno  VARCHAR(50),
    CONSTRAINT student_PK PRIMARY KEY (sid)
             INITIALLY DEFERRED DEFERRABLE,
    CONSTRAINT student_FK FOREIGN KEY (dno)
    REFERENCES dept(dno) INITIALLY IMMEDIATE DEFERRABLE);
```

- Primary key could be DEFERRED/DEFERRABLE **only** if it's not referenced by another table in PostgreSQL

Changing Constraint Evaluation Mode

- It is permitted only for deferrable constraints
- Setting the constraint validation mode within a transaction
  - set mode of all deferrable constraints

```
SET CONSTRAINTS ALL IMMEDIATE;
SET CONSTRAINTS ALL DEFERRED;
```

  - set mode of specific deferrable constraints (list)

```
SET CONSTRAINTS student_FK IMMEDIATE;
SET CONSTRAINTS student_FK DEFERRED;
```


Specifying Transaction Atomicity

- Errors at commit time: only when **deferred constraints** are violated
  - Constraints can be deferred if specified as **deferrable** in the table schema, and
  - deferred in the scope of the transaction
- E.g., *assume the constraints are deferrable*

```
start transaction read write;
   set constraints all deferred;
   insert into Student  values (23, 'John', 'CS');
   insert into Dept values  ('CS', 501);
Commit;
```

- No constraint violation of the first insert is detected at *commit time* → the whole transaction is committed

Specifying Transaction Atomicity (2)

- E.g. 2, *assume the constraints are deferrable and assume SID 23 exists in that Database*

```
insert into Student values (23, 'John', 'CS');

start transaction read write;
   set constraints all deferred;
   insert into Student values (23,'John', 'CS');
   insert into Dept values  ('CS', 501);
Commit;
```

- The constraint violation of the first insert is detected at *commit time* → the whole transaction is rollback

The chicken and the egg problem…

```
CREATE TABLE Chicken(ID INT PRIMARY KEY, eID INT NOT NULL);
CREATE TABLE Egg(ID INT PRIMARY KEY, cID INT NOT NULL);

ALTER TABLE Chicken ADD CONSTRAINT Chicken_FK
            FOREIGN KEY (eID) REFERENCES Egg(ID)
            DEFERRABLE INITIALLY IMMEDIATE;

ALTER TABLE Egg ADD CONSTRAINT Egg_FK
            FOREIGN KEY (cID) REFERENCES Chicken(ID)
            DEFERRABLE INITIALLY IMMEDIATE;
```


The chicken and the egg problem…

```
START TRANSACTION READ WRITE;
SET CONSTRAINTS ALL DEFERRED;
  INSERT INTO Chicken VALUES (1, 2);
  INSERT INTO Egg VALUES (2, 1);
COMMIT;
```


SQL Savepoints

- In long transactions, isolates set of "safe" operations from set of "Risky" Operations
- Operations
  - SAVEPOINT NAME < name > ;
  - RELEASE SAVEPOINT < name > ;
  - ROLLBACK [WORK] [TO SAVEPOINT < name >];

ANSI SQL2 Isolation Levels

- SET TRANSACTION [READ ONLY | READ WRITE]
  [, ISOLATION LEVEL READ UNCOMMITTED |
  READ COMMIT |
  REPEATABLE READ |
  SERIALIZABLE]
- Isolation (alias concurrency atomicity / serializability)
  - transactions are independent,
  - the result of the execution of *concurrent transactions* is the same as if transactions were executed serially, one after the other

Example Transaction

- **CLASS** (classid, max_num_students, cur_num_students)
- Consider the following transaction

```
START TRANSACTION READ WRITE;
SELECT max_num_students, cur_num_students          Read(classid =1)
FROM    CLASS
WHERE classID = 1;

If  (cur_num_students <max_num_students)
    update CLASS                                  Write(classid =1)
    set cur_num_students = cur_num_students +1
    where classID = 1;
else
    print 'The class is full';
COMMIT;
```


Transactions t1 and t2 – Serial Execution

- Assume *max_num_students* = 40, *cur_num_students = 39*
- *Execution of transaction t1 and t2:*

r$_1$(*max_num_students*)
r$_1$(*cur_num_students*) -- *cur_num_students* = ***39***
If (cur_num_students <max_num_students)
  w$_1$(*cur_num_students++*) -- *cur_num_students* = ***40***

r$_2$(*max_num_students*)
r$_2$(*cur_num_students*). -- *cur_num_students* = ***40***
If (cur_num_students <max_num_students) -- 40<40 - false
"The class is full"

Concurrent Transactions

```
START TRANSACTION READ WRITE;
SELECT max_num_students, cur_num_students
FROM CLASS
WHERE classID = 1;

sleep…

If  (cur_num_students <max_num_students)
    update CLASS
    set cur_num_students = cur_num_students +1
    where classID = 1;
else
   print 'The class is full';
COMMIT;
```


Transactions t1 and t2 – Concurrent Execution

- Assume *max_num_students* = 40, *cur_num_students = 39*
- *Execution:*

r$_1$(*max_num_students*)
r$_1$(*cur_num_students*) -- *cur_num_students* = ***39***
… sleep$_1$
r$_2$(*max_num_students*)
r$_2$(*cur_num_students*). -- *cur_num_students* = ***39***
… sleep$_2$
If (cur_num_students <max_num_students)
  w$_1$(*cur_num_students++*) -- *cur_num_students* = ***40***
If (cur_num_students <max_num_students)
  w$_1$(*cur_num_students++*) -- *cur_num_students* = ***41***

Write/Exclusive Lock

- Example:

```
SELECT max_num_students, cur_num_students
FROM CLASS
WHERE classID = 1555
FOR UPDATE OF cur_num_students;
```

- Alternative just specify FOR UPDATE;
- Error Messages:
  - No lock: "Cannot serialize access for this transaction"
  - With lock: "The class is full"