Functions and Procedures

- SQL language designed for specific types of data access and modification operations
  - Other operations much easier to express in procedural code
- Most database systems enable user defined functions and procedures to be written in a DB-specific procedural language
- In many cases, application-specific logic can be encoded in stored procedures

Example Postgres Procedural Code

```
CREATE or REPLACE FUNCTION how_many_consec(cust varchar(10))
RETURNS integer AS $$
DECLARE
    rc RECORD;
    count int;
    prev date;
BEGIN
    prev = null;
    count = 0;
    --Iterate through every row from flewon for this customer
    FOR rc IN SELECT * FROM flewon
              WHERE customerid = cust ORDER BY flightdate LOOP
      IF prev is not null THEN
        IF prev + interval '1 day' = rc.flightdate THEN
          count = count + 1;
        END IF;
      END IF;
      prev = rc.flightdate;
    END LOOP;
    RETURN count;
END;
$$ LANGUAGE plpgsql;
```

You can use the procedure we just made from SQL

```
select how_many_consec('cust100’);

select customerid, how_many_consec(customerid)
from flewon group by customerid;

update customers set fly_consec = how_many_consec(customerid);
```

Triggers

- A ***trigger*** is a statement that is executed automatically by the system as a side effect of a modification to the database.
  - Can choose to run trigger code before or after modification
- Useful for
  - Updating derived tables / columns
  - Performing external world actions as a result of database state (but usually indirectly)
  - Automatic data cleaning
- Most systems have their own syntax
- Be careful
  - Cascading triggers, Infinite Sequences…
  - Specifying cascading functionality in schema definition is usually better
    - E.g. foreign key (att1) references table **on delete cascade**

Example Trigger Code

```
CREATE TRIGGER update_consec_days
       AFTER INSERT OR UPDATE OR DELETE
       ON flewon
  FOR EACH ROW
    EXECUTE PROCEDURE update_consec_days();
```

```
CREATE OR REPLACE FUNCTION update_consec_days() RETURNS trigger AS $$
    BEGIN
     IF TG_OP = 'INSERT' OR TG_OP = 'UPDATE' THEN
      IF EXISTS
        (SELECT * FROM flewon WHERE customerid = NEW.customerid
                                AND flightdate = NEW.flightdate – interval '1 day') THEN
        UPDATE customers set consec_days = consec_days + 1;
      END IF;
      IF EXISTS
        (SELECT * FROM flewon WHERE customerid = NEW.customerid
                                AND flightdate = NEW.flightdate + interval '1 day') THEN
        UPDATE customers set consec_days = consec_days + 1;
      END IF;
     END IF;
     IF TG_OP = 'DELETE' OR TG_OP = 'UPDATE' THEN
      IF EXISTS
        (SELECT * FROM flewon WHERE customerid = OLD.customerid
                                AND flightdate = OLD.flightdate – interval '1 day') THEN
        UPDATE customers set consec_days = consec_days - 1;
      END IF;
      IF EXISTS
        (SELECT * FROM flewon WHERE customerid = OLD.customerid
                                AND flightdate = OLD.flightdate + interval '1 day') THEN
        UPDATE customers set consec_days = consec_days - 1;
      END IF;
     END IF;
     RETURN NULL;
    END;
$$ LANGUAGE plpgsql;
```

Note that there are at least two corner cases in the code above that will cause incorrect functionality. Can you find them?

Transactions

- Unit of work
- Atomic transaction
  - either fully executed or rolled back as if it never occurred
- Isolation from concurrent transactions
- Transactions begin implicitly
  - Ended by **commit work** or **rollback work**
- But default on most databases: each SQL statement commits automatically
  - Can turn off auto commit for a session (e.g. using API)
  - In SQL:1999, can use: **begin atomic** …. **end**
    - Not supported on many database systems