# Trigger: the condition

- The condition appears within a function
  - Also known as a **stored procedure**
- This condition is normally boolean valued
  - However, as the stored procedure can have arbitrary code, it might not check any condition.
  - For our purposes, however, we do want to check it.
- Condition is evaluated on the database as it would exist:
  - Before the triggering event (if **before** is specified in the **create trigger** statement)
  - After the triggering event (if **after** is specified in the **create trigger** statement)
  - Regardless, applied before the changes take effect!
  - Access tuple via NEW and OLD variables (if using **for each row**)

# Trigger: the action

- An action can consist of more than one SQL statement
  - These are normally within the true branch of the procedure's if statement
- Note:
  - Queries make no sense in an action...
  - ... so we are really concentrating on modifications.

# Trigger: second example

- Consider Sells(pub, beer, price)...
- ... and NastyPubs(pub)
- We want to maintain a list of pubs that raise the price of any beer by more than $1
- Idea:
  - We examine when a row in Sells is updated...
  - ... such that its old price differs from its new price ...
  - ... where the new price is > old price + 1.00.
  - If that happens, we insert the pub into NastyPubs

# Trigger: condition + action; create trigger

```
create function price_trig_func() returns trigger as $price_trig$
    begin
        if NEW.price > (OLD.price + 1.00) then
            insert into NastyPubs(pub) values (NEW.pub);
        end if;
        return NEW;
    end;
$price_trig$ language plpgsql;
```

**Assuming here that the table NastyPubs has already been created.**

```
create trigger PriceTrig after update on Sells
    for each row
    execute procedure price_trig_func();
```

# Trigger: third example

- Some triggers are not applied to specific rows
  - Rather, they apply to a given SQL modification statement that might affect one or many rows
  - We use **for each statement** to indicate that the trigger is performed not for each modified row but once a whole statement is completed
- Example: Keeping track of stats
  - If there is a change to the beers sold in pubs (i.e., in Sells), then update PubStats to reflect the changed average price of all beers sold in all pubs

# We will need this...

```
create table PubStats (
    scope          varchar(20),
    avgbeerprice real,
    numberpubs    int,
    newestpub     varchar(20)
);
insert into PubStats(scope) values ('all');
```

# Trigger: condition + action; create trigger

```
create function average_trig_func() returns trigger as
$average_trig$
    begin
        update PubStats
            set avgbeerprice = (select avg(price) from sells)
            where scope = 'all';
        return null;
    end;
$average_trig$ language plpgsql;
```

```
create trigger UpdateAverageTrigger
    after update or delete or insert on Sells
    for each statement
    execute procedure average_trig_func();
```

# A bit of system-design philosophy...

- We have seen some constraints which are attribute- or tuple-based
- We have seen others which link a stored procedure to some events (ECA rules)
- Skipping ahead:
  - We also know some of these constraints could, in some way, be enforced by our own code
  - That is, the logic of our implemented algorithms would ensure data consistency
- Where do we state/implement a constraint?
  - In the database? I
  - In our code?
  - A mixture of both?

# Transactions

- Up to this point:
  - We have concentrated on a single database
  - We have assumed a single user performs all queries and modifications
  - Also have assumed that user does these sequentially (i.e., one after the other)
- However, these assumptions can be too strict
  - Database systems are normally accessed by many users or processors (or both) at the same time
  - These accesses mix both queries and modifications
- While operating systems support process interaction...
- ... DBMSes need some help to control potentially trouble accesses as they interact.

# Troublesome example

- Your name is Pat
- Your partner's name is Chris
- You each transfer $500 from your joint account to your own individual accounts...
  - ... and do so from different ATMs...
  - ... at the same time ...
  - ... and not because you are arguing with each other.
- What we expect:
  - DBMS supporting the bank's operations does not lose one of the transactions.
- What we don't expect:
  - Classic race condition a la operating systems
  - Lost / propogating money
  - Need for couple's counselling

# Potential order of operations

```
select balance from SavingsAccounts
where customer = 'Pat and Chris';
```

```
-- Bank's computer checks if balance is > 500; if so, then...
```

```
update SavingsAccount set balance = balance - 500
where customer = 'Pat and Chris';

update ChequingsAccount set balance = balance + 500
where customer = 'Pat';
```

```
select balance from SavingsAccounts
where customer = 'Pat and Chris';
```

```
-- Bank's computer checks if balance is > 500; if so, then...
```

```
update SavingsAccount set balance = balance - 500
where customer = 'Pat and Chris';

update ChequingsAccount set balance = balance + 500
where customer = 'Chris';
```

# Inappropriate order of operations

```
select balance from SavingsAccounts
where customer = 'Pat and Chris';
```

```
-- Bank's computer checks if balance is > 500; if so, then...
```

```
select balance from SavingsAccounts
where customer = 'Pat and Chris';
```

```
-- Bank's computer checks if balance is > 500; if so, then...
```

```
update SavingsAccount set balance = balance - 500
where customer = 'Pat and Chris';

update ChequingsAccount set balance = balance + 500
where customer = 'Pat';
```

```
update SavingsAccount set balance = balance - 500
where customer = 'Pat and Chris';

update ChequingsAccount set balance = balance + 500
where customer = 'Chris';
```

# Transactions

- There are many variants of this for different situations
  - State of DBMS indicates a possible change can be made
  - Change is performed
  - Want system in some "consistent" state (from the point of view of the users) after the change
- **Transaction**
  - A process involving database queries or modification or both
  - Normally with strong properties regarding concurrency
  - Formed in SQL from single statements or explicit programmer control.

# Properties: ACID

- Transactions are normally designed to be **ACID**
- **Atomic**: Either the whole transaction is performed, or not if it is.
- **Consistent**: Database constraints are preserved.
- **Isolated**: It appears to the users as if only one processes executes at any one time.
- **Durable**: Effects of a process will survive a crash of the system
- Optional:
  - Weaker forms of transaction are also supported (**serializable level**)
  - These permit a higher degree of concurrency but with some potential problems.

# Transaction: syntax

- Sequence of database actions are bracketed by **begin** and **commit**
- After the **commit** operation, database modifications are permanent in the database

```
begin;

update accounts set bal = bal - 90000 where name = 'Nigel Wright;

update accounts set bal = bal + 90000 where name = 'Michael Duffy';

update accounts set bal = bal - 90000 where name = 'Michael Duffy';

update accounts set bal = bal + 90000 where name = 'Senate Petty Cash';

commit;
```

# Rollback

- A transaction can be ended with **rollback** instead of **commit**...
- ... but this instead causes the transaction to be aborted rather than results written to the DBMS
- That is:
  - rollback ensure there are no changes to the database caused by the aborted transaction
- Note:
  - Some failures can require a rollback even without the programmer knowing it.
  - Example: division by 0
  - Example: constraint violation

# Interacting processes: example

- Consider our relation Sells(pub, beer, price)
- Suppose the Valhalla Inn pub sells **only** the following two beers: Blue for $4.00 and Coors Light for $4.25.
- Siegfried is querying Sells for the highest and lowest prices charged by Valhalla Inn
- The manager at Valhalla Inn decides to stop selling Blue and Coors Light...
- ... but she adds Corona to the menu for $4.50.

# Program: run by Siegfried

- Siegfried performs the following two SQL statements.
- The result from each query is stored some place by Siegfried

```
-- max operation
select max(price) from Sells
where pub = 'Valhalla Inn';
```

```
-- min operation
select min(price) from Sells
where pub = 'Valhalla Inn';
```

# Program: run by Brunnhilde from the pub

- Brunnhilde (manager at the Valhalla Inn) executes the following delete and insert steps.
- Why she has write-access to the table used by all pubs is a technical detail we will gloss over for now.

```
-- del operation
delete from Sells
where bar = 'Valhalla Inn';
```

```
-- ins operation
insert into Sells
values ('Valhalla Inn', 'Corona', 4.50);
```

# Interleaving of statements

- We know that:
  - the **max operation** occurs before the **min operation**
  - the **del operation** occurs before the **ins operation**
- There are at present no other restrictions on the order of the operations
- In order to add restrictions to Siegfried and Brunnhilde's actions...
  - ... We would need to use transactions.
  - (Restrictions here only apply to the actions related to the database.)

# Example: strange interleaving

- Assume we are not yet using transactions.
- Suppose the operations are executed in this order:
  - (max)(del)(ins)(min)
- (max): Set of prices is {4.00, 4.25}, the max is 4.25
- (del): Set of prices becomes {}
- (ins): Set of prices becomes {4.50}
- (min): As set of prices is {4.50}, the min is 4.50
- Siegfried sees that the max < min!
  - This is not good (i.e., this is an example of **inconsistency**)