Lecture 10

# SQL

SQL is an industry standard

* Query language -select from where
* Data manipulation language -create/drop/…

Insert/update/delete tuples

* Case insensitive
* Strings are delimited by ‘’
* Each statement ends with ;
* Comments are in --

## Local psql setttings

psqlpassword

psql -h localhost -U postgres

# psql administrative commands

## General:

\l

List all the databases

\c <database name>

Enter the given database

\dc

List all database

\dt

List all tables in database

# PSQL general grammar:

## SELECT

This is as (project) in relational algebra

### e.g.

SELECT first\_name from CUSTOMER;

SELECT first\_name, last\_name from CUSTOMER;

### Concatenation, the || operator

e.g.:

SELECT first\_name || ‘ ‘ || last\_name from CUSTOMER;

### Select can use algebraic expressions

## FROM & WHERE

This is as (select) in relational algebra

FROM specifies where the record shall be selected

### Where

Assert certain attributes:

e.g.:

SELECT \* FROM actors WHERE first\_name = ‘Joe’;

SELECT \* FROM actors WHERE age >= 12;

### General expressions

E.g.:

'abc' LIKE 'abc' *true*

'abc' LIKE 'a%' *true*

'abc' LIKE '\_b\_' *true*

'abc' LIKE 'c' *false*

The ‘%’ character matches 0 or any wildcard characters;

The ‘\_’ character matches exactly one character;

SELECT \* FROM showstoppers WHERE make LIKE ‘%’’%’;

## Null Values

There is no value for an attribute because

* No Value exists
* I don’t know the value
* I don’t know whether there is a value or not

SELECT \* FROM cuteanimals WHERE name **is** null;

### Truth tables

3 = 4 false

3 = 3 true

3 = NULL unknown

unknown and false = false

unknown and true = unknown

unknown or false = unknown

unknown or true = true

not unknown = unknown

## date, time, interval

SELECT

Id

, title

FROM

Episodes

WHERE

firstaired < date ‘2018-09-21’;

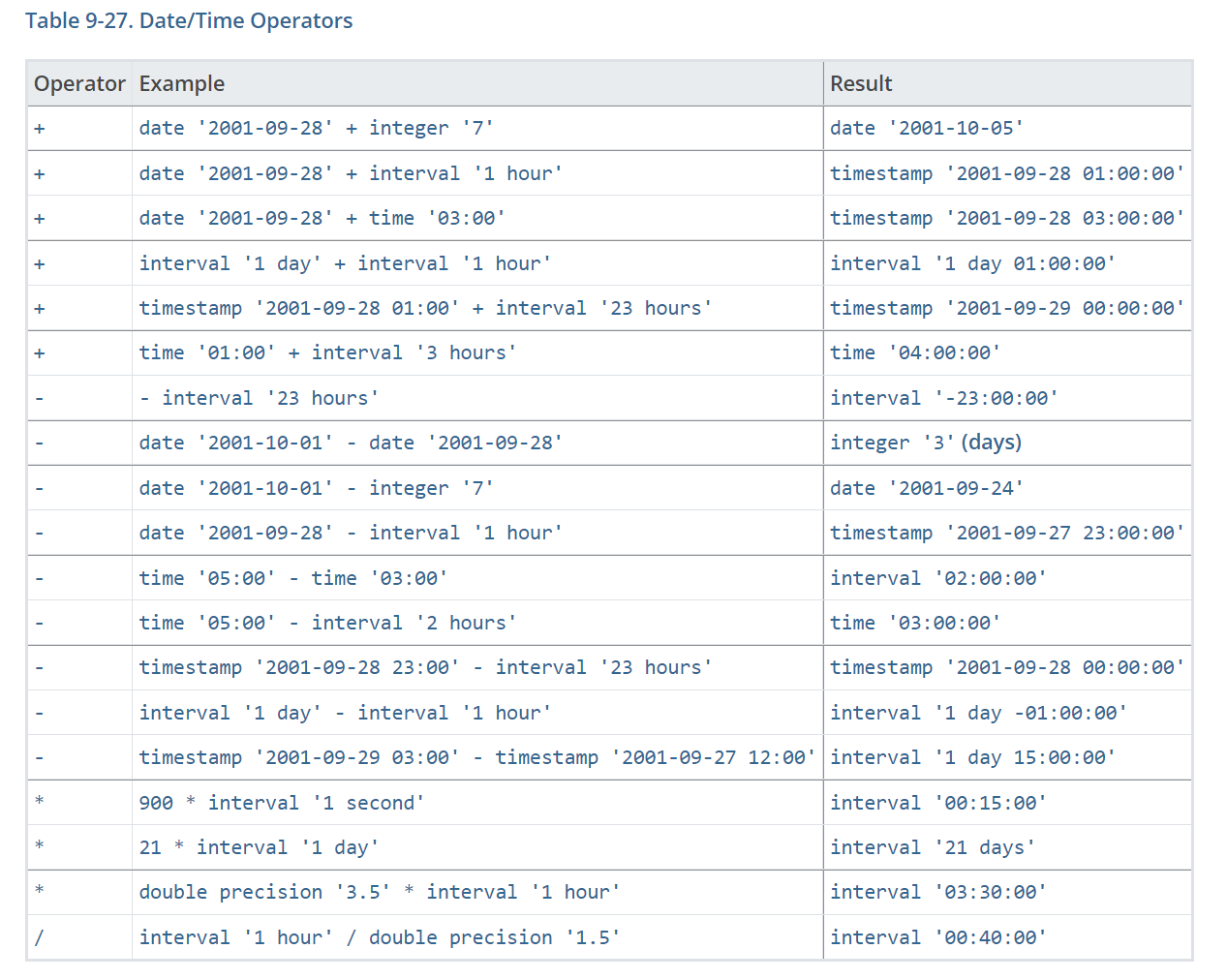
date – date -> interval

date + interval -> date

date + time -> timestamp

timestamp + interval -> timestamp

time + interval -> time



Lecture 11

SQL

-----

SELECT

FROM

WHERE

SELECT baker

FROM results

WHERE result = 'eliminated';

SELECT baker

FROM results

WHERE result = ‘star baker’;

SELECT episodeid, baker

FROM results

WHERE result = ‘star baker’

ORDER BY episodeid desc;

## “PROJECT” and “SELECT” in postgresql

SELECT

b.fullname

, s.episodeid

FROM

bakers b

, signatures s

WHERE

b.baker = s.baker

and lower(s.make) like '%cardamom%'

ORDER BY

b.fullname ;

### e.g.

-- Find the fullname of bakers who used 'lemon' in signature

-- challenges for two back to back episodes

SELECT

b.fullname

FROM

bakers b,

signatures s1,

signatures s2,

WHERE

s1.baker = b.baker

and s2.baker = s1.baker

and s2.episodeid = s1.episodeid + 1

and lower(s2.make) like ‘%lemon%’

and lower(s1.make) like ‘%lemon%’

### e.g.

-- Find the full name of bakers who were eliminated

-- in an episode with 9.5 or more viewers7day, order by name asc

SELECT

b.fullname

FROM

bakers b,

results r,

episodes e

WHERE

b.baker = r.baker

and r.episodeid = e.id

and e.viewers7day >= 9.5

and r.result = ‘eliminated’

ORDER BY

b.fullname ASC;

# Set Operation in SQL

UNION/EXCEPT/INTERSECT

## Grammar

SELECT

FROM

WHERE

UNION/EXCEPT/INTERSECT

SELECT

FROM

WHERE

UNION/EXCEPT/INTERSECT

SELECT

FROM

WHERE

**e.g.**

Select all bakers who have used lemon in signature or showstopper

challenges. Return baker name.

SELECT

baker

FROM

showstoppers

WHERE

lower(make) like '%lemon%'

UNION

SELECT

baker

FROM

signatures

WHERE

lower(make) like '%lemon%';

## e.g.

Select all bakers who have used lemon in signature AND showstopper

challenges. Return baker name.

SELECT

baker

FROM

showstoppers

WHERE

lower(make) like '%lemon%'

INTERSECT

SELECT

baker

FROM

signatures

WHERE

lower(make) like '%lemon%';

#### Alternate solution

SELECT DISTINCT

si.baker

FROM

signatures si

, showstoppers sh

WHERE

si.baker = sh.baker

and lower(si.make) like '%lemon%'

and lower(sh.make) like '%lemon%';

### e.g.

-- Select all bakers who have used chocolate in showstopper make, but

-- never in signature challenge. Return baker name.

SELECT

b.baker,

b,fullname

FROM

bakers b,

showstoppers s

WHERE

b.baker = s.baker

and lower(s.make) like “%chocolate%”

EXCEPT

SELECT

b.baker,

b.fullname

FROM

bakers b,

signatures s

WHERE

b.baker = s.baker

and lower(s.make) like “%chocolate%”

;

# BAG OPERATIONS

R UNION ALL S = {t occurs (m+n) times | t occurs: m times in R and n times in S}

R INTERSECT ALL S = {t occurs min(m,n) | t occurs: m times in R and n times in S}

R EXCEPT ALL S = {t occurs max(0,(m-n)) times | t occurs: m times in R and n times in S}

create table tmp1 (id int) ;

insert into tmp1 values(1) ;

insert into tmp1 values(2) ;

insert into tmp1 values(2) ;

insert into tmp1 values(3) ;

insert into tmp1 values(3) ;

insert into tmp1 values(3) ;

create table tmp2 (id int) ;

insert into tmp2 values(2) ;

insert into tmp2 values(3) ;

insert into tmp2 values(3) ;

insert into tmp2 values(3) ;

insert into tmp2 values(3) ;

insert into tmp2 values(4) ;

insert into tmp2 values(4) ;

# AGGREGATES

SELECT

count(\*) as numtuples

, count(distinct baker) as numbakers

FROM

signatures

WHERE

lower(make) like '%lemon%';

SELECT

count(distinct s.baker) as numbakers

, max(e.viewers7day) as maxviewers

, count(distinct e.id) as numepisodes

, avg(e.viewers7day)::numeric(4,2) as avgviewers

FROM

signatures s

, episodes e

WHERE

s.episodeid = e.id

and lower(make) like '%lemon%';

-- Find for each baker, how many times they used

-- lemon in their signature make

SELECT

baker

, count(\*) as numbakes

FROM

signatures

WHERE

lower(make) like '%lemon%'

GROUP BY

baker ;

-- Find for each episode, how many bakers used

-- lemon in their signature make

SELECT

episodeid

, count(\*) as numbakers

FROM

signatures

WHERE

lower(make) like '%lemon%'

GROUP BY

episodeid ;

SELECT

b.baker

, b.fullname

, count(\*) as numepisodes

FROM

signatures s

, bakers b

WHERE

b.baker = s.baker

and lower(s.make) like '%lemon%'

GROUP BY

b.baker

, b.fullname;

SELECT

b.baker

, b.fullname

, count(\*) as numepisodes

FROM

signatures s

, bakers b

WHERE

b.baker = s.baker

and lower(s.make) like '%lemon%'

GROUP BY

b.baker

, b.fullname

HAVING

count(\*) > 1

;

SELECT

b.baker

, b.fullname

FROM

showstoppers s

, bakers b

WHERE

b.baker = s.baker

and lower(s.make) like '%chocolate%'

EXCEPT

SELECT

b2.baker as baker1

, b2.fullname

FROM

signatures s

, bakers b2

WHERE

s.baker = b2.baker

and lower(s.make) like '%chocolate%'

ORDER BY

fullname

;

For each baker and technical positions 1-4, how many times each baker hade that position.

SELECT

baker,

count(\*) as numepisodes

FROM

technicals

WHERE

rank = 1

GROUP BY

baker

HAVING

count(\*) > 1

ORDER BY

numepisodes DESC;

For each baker and each technical position 1-4, return how many times each baker had that position.

SELECT

baker,

rank,

count(\*) as numepisodes,

min(episodeid)

max(episodeid)

FROM

technicals

WHERE

rank <=4

GROUP BY

baker,

rank

ORDER BY

baker ASC,

rank ASC;

For each baker and each technical position 1-4, return how many times each baker had that position

SELECT

baker,

rank,

count(\*) as numepisodes,

min(episodeid)

max(episodeid)

FROM

technicals

WHERE

rank <=4

GROUP BY

baker,

rank

HAVING

count(\*) >1

ORDER BY

baker ASC,

rank ASC;

‘

For each baker, find how many times they were a judge favorite.

SELECT

b.baker,

b.fullname,

count(\*) as numtimes

FROM

favorites f,

bakers b

WHERE

f.baker = b.baker

GROUP BY

b.baker

UNION

(

SELECT

baker,

fullname,

0

FROM

Bakers

EXCEPT select

bakers b,

favorites f,

0

WHERE

b.baker = f.baker

);

# Inner Join vs Outer Join

SELECT

b.baker,

b.fullname,

FROM

baker b

LEFT OUTER join favorites f

ON f.baker = b.baker

WHERE

f.baker is null;

## sample

a: id1

1

2

b: id2

2

3

## Inner Join

ab: id1 id2

2 2

Return the intersection of two

## Left Join

ab: id1 id2

1

2 2

Inner Join + leftover tuples from left

## Right Join

ab: id1 id2

2 2

3

Inner Join + leftover tuples from right

## FULL Join

ab: id1 id2

1

2 2

3

Left Join + Right Join

## e.g.

For each baker, find total number of times they won star baker.

SELECT DISTINCT

b.baker,

b.fullname,

count(r.baker)

FROM

baker b,

left join results r,

on b.baker = r.baker and r.result = ‘star baker’

GROUP BY

b.baker

;

For each baker, find total number of time they were a favorite of judges or they won stat baker.

SELECT DISTINCT

b.baker,

b.fullname,

count(DISTINCT r.episodeid) + count(DISTINCT f.episodeid) as correctcount

count(r.episodeid) + count(f.episodeid) as incorrectcount

FROM

baker b,

left join results r,

on b.baker = r.baker and r.result = ‘star baker’

left join favorites f

on b.baker = f.baker

GROUP BY

b.baker

;

# Limit Statement

SELECT

FROM

WHERE

GROUP BY

HAVING

UNION

SELECT

FROM

WHERE

GROUP BY

HAVING

ORDER BY

**LIMIT K;**

This will limit the maximum number of return

# Anonymous queries

--Not necessary to use

For each baker, find the number of times they were #1 in technicals.

SELECT

baker,

count(\*) as numepisodes

FROM

technicals

WHERE

rank = 1

GROUP BY

baker

HAVING

count(\*) > 1

ORDER BY

numepisodes DESC;

return maximum number of times a baker has been number 1 in technicals.

SELECT

max(numepisode)

FROM (

SELECT

baker,

count(\*) as numepisodes

FROM

technicals

WHERE

rank = 1

GROUP BY

baker

HAVING

count(\*) > 1

) as rank1

For each baker, return total number of time they were stark baker and number one in technicals.

SELECT

r.baker,

count(\*) as numstarbakerwinsandrank1

FROM (

SELECT

baker,

count(\*) as numepisodes

FROM

technicals

WHERE

rank = 1

GROUP BY

baker

HAVING

count(\*) > 1

) as rank1,

results r

WHERE

rank1.baker = r.baker

and r.result = ‘star baker’

GROUP BY

r.baker,

rank1.numepisodes;

for each baker, find the average viewership of episodes in which they won star baker and the average viewership of episodes in which they placed first in a technical challenge. Only return bakers who won starbakers.

SELECT

r. baker,

AVG(e.viewers7day)::numeric(5,2) as avgviewership

AVG(e2.viewers7day)::numeric(5,2) as avgviewership2

count(DISTINCT e.id) as s1

count(DISTINCT e2,id) as s2

FROM

episodes e,

join results r

on e.id = r.episodeid and r.result = ‘star baker’

left join technicals t

on r.baker = t.baker and t.rank = 1

left join episodes e2

on t.episodeid = e2.id

WHERE

e.id = r.episodeid

and r.result = ‘star baker’

GROUP BY

r.baker

# Scalar Queries

Queries that returns one tuple with 1 attributes

SELECT

count(\*)

FROM

results

WHERE

result = ‘star baker

and baker = ‘Briony’;

Find people who won star baker more time than briony

SELECT

baker

FROM

results

WHERE

result = ‘star baker

GROUP BY

baker

HAVING

count(\*) > (

SELECT

count(\*)

FROM

results

WHERE

result = ‘star baker

and baker = ‘Briony’;

)

Find the airdate of episodes that aired after the first episode to reach 9 million in viewers7day.

SELECT

id,

airdate

FROM

episodes

WHERE

firstairdate > (

SELECT

MIN(airdate)

FROM

Episodes

WHERE

Viewers7day >= 9;

)

# Set Operations

Value in (SET)

Value not in (SET)

Value >= ALL(SET)

Value <= ALL(SET)

Value <= ANY (SET)

Value >= ANY(SET)

e.g.

SELECT

id

FROM

episodes

WHERE

Viewers7day in (SELECT viewers7day FROM episodes where viewers7day >= 9)

SELECT

id,

airdate

FROM

episodes

WHERE

firstairdate <= ALL(

SELECT

firstairdate

FROM

Episodes

WHERE

Viewers7day >= 9;

)

Lecture 13

# Using Query inside a query:

## Option 1 Anonymous query:

Query 🡪 use in from statemen like a relation (anonymous relations)

FROM (--- Query ---) as x

航指出，可以不用as

Where x.

## Option 2 Scalar query:

Query that returns a single value 🡪 use anywhere instead of a number (Scalar queries)

…

where 5 = (--- Scalar query ---)

## Option 3 Set operations:

Query that returns a set/bag of tuples

Check values against the set

Value in (--- query ---)

Value not in (--- query ---)

Value >= ALL (--- query ---) (>=, >, =, <, <=)

Value >= ANY (--- query ---) (>=, >, =, <, <=)

Exists (--- query ---) – true if the result of the query is a non-empty set/bag

NOT Exists (--- query ---) – true if the result of the query is an empty set

## e.g.

**Find episodes in which no one has been eliminated**

SELECT id FROM episodes

EXCEPT

SELECT episodes FROM Results WHERE Result = ‘eliminated’;

SELECT episodeid

FROM

episode e

LEFT OUTER JOIN result r

ON e.id = r.episodeid and r.result = ‘eliminated’

WHERE

r.episodeid is null;

SELECT

id

FROM

episodes

WHERE

Id NOT IN (SELECT episodeid FROM results WHERE result=’eliminated’);

**Find episode with the highest number of elimination**

SELECT

episodeid

FROM

results

WHERE

result = ‘eliminated’

GROUP BY

episodeid

HAVING

COUNT(\*) >= ALL (

SELECT

episodeid

FROM

results

WHERE

result = ‘eliminated’

GROUP BY

episodeid

)

**Find bakers who got a rank 1 in technical in an episode and then never won star baker after the episode.**

SELECT

DISTINCT baker

FROM

technical t

WHERE

t.rank = 1

and NOT EXIST (

SELECT

episodeid

FROM

result r

WHERE

r.result = ‘star baker’

and t.baker = r.baker

and t.episodeid < r.episodeid

)

**Episode in which no one is eliminated**

SELECT

e.id

FROM

episode e

WHERE

NOT EXISTS (SELECT r.episodeid

FROM results r

WHERE r.result=’eliminated’

AND r.episodeid = e.id);

**Return for each constant the last episode in which they won rank 1 in technical**

SELECT

FROM

technicals t

WHERE

t.rank = 1

AND NOT EXISTS (SELECT

FROM technicals t2

WHERE t2.rank = 1

AND t2.baker = t.baker

AND t2.episodeid >t.episodeid)

# TRANSACTIONS

A sequence of operations that change data

## Sample usage:

BEGIN

-- sql operations that query or change data

END;

## Atomicity:

A transactions either executes fully or not at all

If xact succeeds: commit 🡪 make changes permanent!

If xact fails: rollback 🡪 all changes made by the transaction are erased!

DROP TABLE IF EXIST tmp;

CREATE TABLE tmp (

id INT PRIMARY KEY,

name VARCHAR(10) NOT NULL,

val VARCHAR(10)

);

INSERT INTO tmp VALUES(2,’ABC’,’val1’)

INSERT INTO tmp(id, name) VALUES(3,’ABC’)

INSERT INTO tmp(id, val) VALUES(3,’ABC’)

(^^^^^^^^^^^^^^^^^^^^^^^^^^^ this will fail because name cannot be NULL)

## INSERT

### Inert known value into a table:

INSERT INTO table\_name (column\_list)

VALUES

(value\_list\_1),

(value\_list\_2),

...

(value\_list\_n);

### Insert the result of a query into another relation

INSERT INTO tmp SELECT … FROM … WHERE;

create table stats (baker varchar(20), num int);

INSERT INTO stats

SELECT

r.baker, count(DISTINCT t.episodeid) + count(DISTINCT r.episodeid) as numwins

FROM

results r

FULL JOIN technicals t on r.baker = t.baker

AND t.rank = 1

AND r.result = ‘star baker’

AND r.baker is NOT NULL

GROUP BY

r.baker

;

## DELETE

delete from relation

where condition;

Delete all tuples that satisfy the condition in the WHERE clause.

DELETE FROM stats WHERE numwins = 1;

DELETE FROM stats;

Delete all tuples from stats for bakers who never won stat baker.

DELETE FROM stats

WHERE baker NOT IN (SELECT baker FROM results where result = ‘stat baker’)

## UPDATE

Update relation

SET attribute = X, attribute 2 = Y;

WHERE condition

For those tuples satisfy the WHERE condition

Change the value of attribute in SET

UPDATE stats

SET numwins = numwins + 1

WHERE baker like ‘R%’;

Update stats, add 1 to numwins for people who got rank 1 in technicals more than once.

UPDATE stats

SET numwins = numwins + 1

WHERE baker in (SELECT

Baker

FROM

Technicals

WHERE

Rank = 1

GROUP BY

Baker

HAVING

COUNT(\*) > 1)

DELETE FROM stats;

INSERT INTO stats SELECT baker, 0 FROM bakers;

UPDATE stats

SET numwins = (SELECT count(\*) FROM results r

WHERE r.result = ‘star baker’ and r.baker = stats.baker)

Lecture 14

DROP TABLE IF EXISTS t4;

DROP TABLE IF EXISTS t3;

DROP TABLE IF EXISTS t2;

DROP TABLE IF EXISTS t1;

CREATE TABLE t1(

id1 int primary key

, name varchar(10)

);

CREATE TABLE t2(

id2 int primary key

,id1 int

,name varchar(10)

,foreign key(id1) references t1(id1)

on delete cascade

on update set null

);

CREATE TABLE t3(

id3 int primary key

,id1 int

,id2 int not null

,foreign key(id1) references t1(id1)

on delete set null

on update cascade

,foreign key(id2) references t2(id2)

on delete cascade

on update cascade

);

CREATE TABLE t4(

id4 int primary key

,id1 int

, a int

, b int

, c int

, d int

, foreign key (id1) references t1(id1)

on delete cascade

on update set null

,unique(a,b)

,check(c is not null or d is not null)

);

INSERT INTO t1 values(1, 'a');

INSERT INTO t1 values(2, 'b');

INSERT INTO t1 values(3, 'c');

INSERT INTO t1 values(4, 'd');

INSERT INTO t2 values(1,1, 'r');

INSERT INTO t2 values(2,1, 's');

INSERT INTO t2 values(3,4, 't');

INSERT INTO t2 values(4,2, 'v');

INSERT INTO t2 values(5,null, 'w');

INSERT INTO t3 values(1,2,3);

INSERT INTO t3 values(2,2,3);

INSERT INTO t3 values(3,null,4);

INSERT INTO t3 values(4,4,2);

INSERT INTO t3 values(5,4,1);

# Create -- Parameters

## Cascade

The operation of the referenced foreign key is passed through to the next level

e.g. delete, update

## Restrict

The linked keys will not allow any update/ delete operation when the behavior is not specified

## Set

Specific behavior

## () on delete

e.g. set null on delete

## () on update

e.g. cascade on update

## Tuple level constrains

### Unique()

### Check()

# Table or database level constraints

create assertion nozerousers (

check (select count(\*) from users) > 0

);

# Isolation in Transactions

Read(X)

X=X-W

Write(X)

Read(Y)

Y=Y+10

Write(Y)

文本, 信件

描述已自动生成

# Dirty Value

A value that is changed by an uncommitted transaction

BEGIN;

SET ISOLATION LEVEL xxx

## Isolation levels:

### READ UNCOMITTED;

Fastest,

Read even though someone is in the procedure of editing this value

--- xacts cannot change tuples

### READ COMMITTED;

Only read the committed levels

### REPEATABLE READ;

Read committed + repeatable

### SERIALIZABLE

Read committed + repeatable + no phantom updates

### Sample transaction

BEGIN TRANSACTION;

SELECT \* FROM T;

WAITFOR DELAY '00:01:00'

SELECT \* FROM T;

COMMIT;

* under READ COMMITTED, the second SELECT may return *any* data. A concurrent transaction may update the record, delete it, insert new records. The second select will always see the *new* data.
* under REPEATABLE READ the second SELECT is guaranteed to display at least the rows that were returned from the first SELECT *unchanged*. New rows may be added by a concurrent transaction in that one minute, but the existing rows cannot be deleted nor changed.
* under SERIALIZABLE reads the second select is guaranteed to see *exactly* the same rows as the first. No row can change, nor deleted, nor new rows could be inserted by a concurrent transaction.

TABLE (X, Y, Z)

# Procedural Programming

**For each X, return the highest 2 Y values,**

SELECT X, Y

FROM T1,

ORDER BY X, Y

**For each X, return 2 Y values with highest total Z**

Loop

SELECT X, Y, sum(Z) as val

FROM T1,

ORDER BY X, Y, val

Choose the top 2 Y for each X

# Basic constructs to add DB functionality:

Create a connection to DB server

Run a Query or update/insert/delete operation

* feed programming variables into the program

Loop through all the result of query

* read returned values into program variables

Close a connection/query results

Check errors

PL/pgSQL:

Documentation here:

https://www.postgresql.org/docs/13/plpgsql.html

Disclaimer: not all functions I write here are good ideas!

<don't use all of these, they are just examples.>

Basic structure:

-------------------------------------

CREATE FUNCTION functionname(variable type) RETURNS type AS $$

DECLARE

varname type ;

BEGIN

<body>

RETURN value ;

END ;

$$ LANGUAGE plpgsql ;

--------------------------------------

Data types:

user\_id integer;

quantity numeric(5);

url varchar;

myrow tablename%ROWTYPE;

myfield tablename.columnname%TYPE;

arow RECORD;

# Defining functions and use in query:

## Function definition:

DROP FUNCTION IF EXISTS epsum(int, date, varchar) ;

CREATE OR REPLACE FUNCTION

epsum(id int, epdate date, name varchar)

**RETURNS varchar AS**

**$$**

DECLARE

sumdata varchar ;

BEGIN

sumdata = 'Episode: ' || id::varchar || ' titled "' || name || '" took place on ' || cast(epdate as varchar) ;

RETURN sumdata ;

END ;

**$$ LANGUAGE plpgsql ;**

## Usage in query:

SELECT title, epsum(id, firstaired, title) from episodes ;

# Queries that returns one tuple;

## Select \*, \* INTO \*, \*

This is assigning variable to variable, 1 to 1 relation

Selecting the given parameters into new variables

(similar to SELECT \* AS \*)

## Version 1

DROP FUNCTION IF EXISTS epsum2 (int);

CREATE OR REPLACE FUNCTION

epsum2(inputepid int)

RETURNS varchar AS

$$

DECLARE

sumdata varchar ;

epdate date ;

eptitle varchar ;

BEGIN

-- Assigning parameters into variables

SELECT firstaired, title INTO epdate, eptitle

FROM episodes where id = inputepid ;

sumdata = 'Episode: ' || inputepid::varchar || ' titled "' || eptitle || '" took place on ' || cast(epdate as varchar) ;

RETURN sumdata ;

END ;

$$ LANGUAGE plpgsql ;

## Creating a “record”-like datatype

## Version 2

CREATE OR REPLACE FUNCTION epsum2(inputepid int) RETURNS varchar AS $$

DECLARE

sumdata varchar ;

myrow RECORD ;

BEGIN

**SELECT firstaired, title INTO myrow**

FROM episodes where id = inputepid ;

sumdata = 'Episode: ' || inputepid::varchar || ' titled "' || myrow.title || '" took place on ' || cast(myrow.firstaired as varchar) ;

RETURN sumdata ;

END ;

$$ LANGUAGE plpgsql ;

## Version 3

DROP FUNCTION IF EXISTS epsum3 (int);

CREATE OR REPLACE FUNCTION epsum3(inputepid int)

RETURNS varchar AS

$$

DECLARE

sumdata varchar ;

myrow RECORD ;

BEGIN

SELECT firstaired, title INTO myrow

FROM episodes where id = inputepid ;

sumdata = 'Episode: ' || inputepid::varchar || ' titled "' || myrow.title || '" took place on ' || cast(myrow.firstaired as varchar) ;

RETURN sumdata ;

END ;

$$ LANGUAGE plpgsql;

SELECT id, epsum3(id) from episodes;

----------------

# Queries that return more than one tuple:

DROP FUNCTION IF EXISTS bakerinfo(varchar);

CREATE OR REPLACE FUNCTION bakerinfo(inputbaker varchar) RETURNS varchar AS $$

DECLARE

myrow RECORD;

summary VARCHAR ;

BEGIN

summary = 'Baker ' || inputbaker || ' was star baker in: ';

FOR myrow IN SELECT episodeid FROM results

WHERE result = 'star baker' AND lower(baker) = lower(inputbaker)

LOOP

summary = summary || myrow.episodeid::varchar || ' ';

END LOOP;

summary = trim(summary) || ' and favorite in: ' ;

FOR myrow IN SELECT episodeid FROM favorites

WHERE lower(baker) = lower(inputbaker)

LOOP

summary = summary || myrow.episodeid::varchar || ' ';

END LOOP;

summary = trim(summary) || '.' || E'\n' || 'Won technical in: ' ;

FOR myrow IN SELECT episodeid FROM technicals

WHERE rank = 1 AND lower(baker) = lower(inputbaker)

LOOP

summary = summary || myrow.episodeid::varchar || ' ';

END LOOP;

summary = trim(summary) || '.' || E'\n' || 'Eliminated in: ' ;

FOR myrow IN SELECT episodeid FROM results

WHERE result = 'eliminated' AND lower(baker) = lower(inputbaker)

LOOP

summary = summary || myrow.episodeid::varchar ;

END LOOP;

RETURN summary || '.';

END ;

$$ LANGUAGE plpgsql ;

select baker, bakerinfo(baker) from bakers;

select bakerinfo('Dan');

# Queries that are run many times: cursors

DROP FUNCTION youngerbakers();

CREATE OR REPLACE FUNCTION youngerbakers() RETURNS VARCHAR AS $$

DECLARE

bakerval VARCHAR;

curs1 CURSOR FOR SELECT b.fullname

FROM bakers b, bakers b2

WHERE b.age < b2.age AND b2.baker = bakerval

ORDER BY b.fullname ASC;

myrow RECORD ;

myrow2 RECORD ;

summary VARCHAR ;

BEGIN

summary = '' ;

FOR myrow IN SELECT baker, fullname FROM bakers ORDER BY age DESC

LOOP

summary = summary || 'baker: ' || myrow.fullname ;

summary = summary || E'\n' || 'Younger bakers: ' ;

bakerval = myrow.baker;

FOR myrow2 IN curs1 LOOP

summary = summary || myrow2.fullname || E' ';

END LOOP ;

summary = summary || E'\n' ;

END LOOP ;

RETURN summary ;

END ;

$$ LANGUAGE plpgsql ;

# Queries that return more than one tuple, i.e. a relation!

DROP TABLE IF EXISTS younger;

CREATE TABLE younger (baker1 varchar(10), baker2 varchar(10));

CREATE OR REPLACE FUNCTION youngerbakers2() RETURNS **SETOF younger** AS $$

DECLARE

bakerval VARCHAR;

myrow RECORD ;

**returnrow younger%rowtype;**

BEGIN

FOR myrow IN

SELECT b1.baker as baker1, b2.baker as baker2

FROM bakers b1, bakers b2

WHERE b1.age < b2.age

LOOP

returnrow.baker1 = myrow.baker1;

returnrow.baker2 = myrow.baker2;

**RETURN NEXT returnrow;**

END LOOP ;

RETURN ;

END ;

$$ LANGUAGE plpgsql ;

select b1.\*,b.fullname from youngerbakers2() as b1, bakers b where b1.baker1=b.baker ;

select \* from youngerbakers2() ;

select youngerbakers2() ;

---------------------

# Procedures

(that do not return anything, a newer addition to pl/pgsql):

CREATE OR REPLACE PROCEDURE test1()

AS $$

BEGIN

CREATE TABLE summary AS SELECT baker, count(\*) as count

FROM technicals WHERE rank=1

GROUP BY baker ;

COMMIT;

END;

$$ LANGUAGE plpgsql ;

**CALL** test1();

# Triggers:

Constraint management:

- Trigger is triggered when a certain event occurs

insert/update/delete from specific tables

- There can be additional conditions (WHEN statement)

- Trigger body tells you what to do

- NEW is the new value of a tuple

OLD is the old value of the tuple

Updates have new and old both

Deletes only have old values

Inserts only have new values

CREATE FUNCTION fix\_favorites ()

RETURNS **trigger** AS

$$

BEGIN

IF NEW.result = 'star baker' THEN

DELETE FROM favorites

WHERE baker = NEW.baker AND episodeid = NEW.episodeid ;

END IF ;

RETURN NEW;

END;

$$ LANGUAGE plpgsql;

CREATE TRIGGER fix\_favorites AFTER INSERT OR UPDATE ON results

FOR EACH ROW EXECUTE FUNCTION fix\_favorites();

CREATE FUNCTION fix\_episodes () RETURNS trigger AS $$

BEGIN

IF NEW.viewers7day-OLD.viewers7day > 5 THEN

NEW.viewers7day = OLD.viewers7day;

END IF ;

RETURN NEW;

END;

$$ LANGUAGE plpgsql;

CREATE TRIGGER episode\_trigger **BEFORE** UPDATE ON episodes

FOR EACH ROW EXECUTE FUNCTION fix\_episodes();

---------

Frameworks:

Django

Flask

Ruby on Rails

.NET

WebObjects

Spring

Grails

Lecture 16

# Anonymous views

SELECT baker, COUNT(\*) AS numwins,

FROM results,

WHERE result = ‘star baker’

GROUP BY baker;

SELECT

FROM (

SELECT baker, COUNT(\*) AS numwins,

FROM results,

WHERE result = ‘star baker’

GROUP BY baker;

) as w,

bakers b

WHERE

w.baker = b.baker;

# WITH

Anonymous query that may be reused at multiple times;

WITH wincounts AS

(

SELECT baker, COUNT(\*) AS numwins,

FROM results,

WHERE result = ‘star baker’

GROUP BY baker;

)

SELECT b.fullname, w.numwins

FROM bakers b, wincounts w

WHERE b.baker = w.baker;

WITH wincounts AS

(

SELECT baker, COUNT(\*) AS numwins,

FROM results,

WHERE result = ‘star baker’

GROUP BY baker;

)

SELECT

f.baker

FROM favorites f

GROUP BY f.baker

HAVING COUNT(\*) > (SELECT w.numwins

FROM wincounts w

WHERE w.baker = f.baker);

WITH wincounts AS

(

SELECT baker, COUNT(\*) AS numwins,

FROM results,

WHERE result = ‘star baker’

GROUP BY baker;

) ,

favcounts AS

(

SELECT baker, COUNT(\*) AS numfavs,

FROM results,

GROUP BY baker;

) ,

SELECT

w.numwins + f.numfavs

FROM

Wincount w, favcount f

WHERE

w.baker = f.baker;

## WITH can use relations already created

# Views

Views are queries that is stored locally and may be referenced in further queries.

CREATE VIEW wincounts(baker, numwins) AS

SELECT baker, count(\*) as numwins

FROM results

WHERE result = 'star baker'

GROUP BY baker ;

SELECT b.fullname, w.numwins

FROM bakers b, wincounts w

WHERE b.baker = w.baker;

All views allow you to query tables. You can use them to define

access control.

## Updatable views

Only views with a single table (no joins) and no distinct/group by/aggregates are updatable. In view with no such constructs. There is a one to one correspondence between tuples in the view and the table.

create view olderthan40(baker, name, age) as

select baker, fullname, age

from bakers

where age > 40

with check option;

update olderthan40 set name = 'Jon Jenkins II'

where baker = 'Jon' ;

update olderthan40 set age = 30

where baker = 'Terry' ;

insert into olderthan40 values('Sibel', 'Sibel Adali', 60) ;

# indexing

Radio DB

CREATE INDEX sindex1 on songs(id, artistid);

# Users

DATABASE Server

Satabase

Schema

User:

Role: groups of users or roles

CREATE user “” password “”;

Inherit: inherit all privileges as soon as you login;

Create role activetester login inherit;

grant activetester to “”;

Grant: SELECT/INSERT/UPDATE/DELETE

Revoke

Grant connect on database xyz to user/role;

Grant select on table to user/role **with grant option**

--- user can now grant it to other users