

disjoint

syntax

Res = Rename[Res(b,c,d)](Project[b,c](Sel[a>5](R)) Join S)  
Tmp1 = Select[a>5](R)  
Tmp2 = Project[b,c](Tmp1)  
Tmp3 = Rename[Tmp3(cc,d)](S)  
Tmp4 = Tmp2 Join[c=cc] Tmp3  
Res = Rename[Res(b,c,d)](Tmp4)  
  
select c.given, c.family, t.source, t.amount  
from Transactions t join Customers c on t.actor = c.id  
 join Accounts a on t.source = a.id  
 join Branches b on a.held\_at = b.id  
where t.tdate < '2022-05-01' and t.ttype = 'withdrawal';  
  
Tmp1 = Sel[t.tdate < '2022-05-01' and t.ttype = 'withdrawal'] Transactions  
Tmp2 = Customers Join[Customers.id = Tmp1.actor] Tmp1  
Tmp3 = Accounts Join[Accounts.id = Tmp2.source] Tmp2  
Tmp4 = Branches Join[Branches.id = Tmp3.held\_at] Tmp3  
Result(given, family, source, amount) = Proj[given, family, source, amount] Tmp4

Selection: sel[expr](rel)  
-> selects the rows in some relation that satisfies expression  
  
Projection: proj[a,b,c](rel)  
-> filters relation by its columns a, b, c  
  
Rename: rename[schema](rel)  
-> renames the columns of rel with the columns specified in schema

View Serializability:  
1) Reads same version of shared object  
2) Writes same final version of shared object  
  
T1: R(A) W(A) R(B) W(B)  
T2: R(A) W(A) R(B) W(B)  
  
T1: R(A) W(A) R(B) W(B)  
T2: R(A) W(A) R(B) W(B)  
  
A: T1 reads initial, T2 reads T1's write, T2 writes final  
B: T1 reads initial, T2 reads T1's write, T2 writes final

Conflict Serializability: Make concurrent into serial schedule with swapping orders

Transaction: ‘unit of work’, may change multiple databases at once.  
  
READ, WRITE, BEGIN, COMMIT, ABORT  
Syntax: R(X), W(Y), etc  
  
No concurrency  
T1: R(X) W(X) R(Y) W(Y)  
T2: R(X) W(X) R(Y) W(Y)  
With concurrency  
T1: R(X) W(X) R(Y) W(Y)  
T2: R(X) W(X) R(Y) W(Y)

For all FDs X->Y…  
1) X->Y is trivial (is Y a subset of X?)  
2) X is a superkey  
3) Y is single attribute & partof candidate key  
1,2: BCNF, 1,2,3: 3NF  
  
F = {AB->C, AB->D, C->A, D->B}  
AB+ = ABCD, C+ = AC, CD+ = ABCD

|  |  |  |
| --- | --- | --- |
|  | 3NF? | BCNF? |
| AB->C | 2 | 2 |
| AB->D | 2 | 2 |
| C->A | 3 | X |
| D->B | 3 | X |

Minimal Cover: Smallest set of FDs which covers entire FD set  
  
F = {A->BC, B->C, A->B, AB->C}  
1) Convert FDs into canonical form  
{A->B, A->C, B->C, AB->C}  
2) For multi-attribute LHS, remove redundant attributes  
{A->B, A->C, B->C, A~~B~~->C}  
3) Eliminate redundant FDs  
{A->B, ~~A->C~~, B->C}

Single Table  
create table R (  
 id integer primary key,  
 is\_a char(1) not null check (is\_a in ('A','B','C')),  
 x text,  
 y text  
);

ER-style  
create table People (  
 id integer primary key,  
 name text not null,  
 lives\_in text not null  
);   
  
create table Customers (  
 id integer primary key references People(id)  
);  
  
create table Employees (  
 id integer primary key references People(id),  
 salary integer not null check (salary > 0),  
 works\_in integer not null references Branches(id)  
);  
  
Single Table  
create table People (  
 id integer primary key,  
 name text not null,  
 lives\_in text not null,  
  
 is\_customer boolean not null,  
 is\_employee boolean not null,  
 salary integer,  
 works\_in integer references branches(id),  
  
 constraint emp\_data check (  
 (is\_employee = false and salary is null and works\_in is null)  
 -- non-employees can not have salary or work in a branch  
 or  
 (is\_employee = true and salary > 0 and works\_in is not null)  
 -- employees must have a positive salary and work in a branch  
 ),  
 constraint total\_participation check (  
 is\_customer = true or is\_employee =  
true  
 )  
);

confusing

ER-style  
create table A (  
 id integer,  
 x text,  
 primary key (id)  
);  
create table B (  
 a integer references A(id),  
 y text,  
 primary key (a)  
);  
create table C (  
 a integer references A(id),  
 primary key (a)  
);  
  
Single Table   
create table A (  
 id integer,  
 x text,  
 y text,  
 b boolean, -- true if A is a B  
 c boolean, -- true if A is a C  
 primary key (x)  
);

ER-style  
Person: ID, name  
Employee: ID, salary  
Manager: ID, bonus  
  
O-O style  
Person: ID, name  
Employee: ID, name, salary  
Manager: ID, name, salary, bonus  
  
Single Table  
Person: ID, name, salary, bonus  
(class depends on null values)

create table Uses (  
 recipe integer,  
 ingredient integer,  
 amount integer not null,  
 unit text not null,  
 primary key (recipe, ingredient),  
 foreign key (recipe) references Recipes(id),  
 foreign key (ingredient) references Ingredients(id),  
 constraint positive\_amount check (amount > 0)  
);

create table S (  
 id serial,  
 t integer,  
 primary key (id),  
 foreign key (t) references T(id)  
);  
  
create table T (  
 id serial,  
 c text not null,  
 primary key (id)  
);

Superkey -> Candidate key -> Primary key  
  
Functional Dependency example  
-> (Position -> Salary)

|  |  |  |
| --- | --- | --- |
| A | B | C |
| a | 1 | X |
| a | 1 | y |

hold O: A->B  
hold X: A->C, B->C, AB->C

Functional Dependencies

There is a conflict between 2 transactions if 1 & 2 apply  
1) Performs operations on same data item  
2) At least one is write (e.g. R&W, W&W, W&R)  
  
Conflict Serializable:  
If we can swap non-conflicting operations so that result is serial schedule: ‘Conflict Serializable’  
  
View Serializable: If some schedule S and some serial schedule S’  
1. Same transaction reads same initial value of X  
2. Same transaction reads same changed version of X  
3. Same transaction writes to final value of X  
  
e.g.  
T1: W(Y) W(X)  
T2: R(Y) W(X)  
T3: W(X)  
  
Conflict:  
X: T1->T2, T2->T3, T1->T3  
Y: T2->T1  
Cycle In T1 and T2, not conflict serializable  
  
View:  
Considering T2;T1;T3  
T1: W(Y), W(X)  
T2: R(Y) W(X)  
T3: W(X)  
  
X) Final write: T3  
Y) First read: T2, Final write: T1  
Both satisfy above, therefore is ‘view serializable’

Divison  
-> Here, get all A values where B has x and y.

Closure: Largest collection of dependencies that can be derived from set F (aka F+)  
  
F = {A->B, BC->F, BD->EG, AD->C, BEG->FA}  
ACEG+ = {A, B, C, E, F, G}

Union: R∪S (Query1) UNION (Query2)  
Intersection: R∩S (Query1) INTERSECT (Query2)  
Difference: R - S (Query1) EXCEPT (Query2)

overlapping

#!/usr/bin/python3  
import sys  
import psycopg2  
conn = None  
usage = f"Usage: {sys.argv[0]} lmao"  
name = sys.argv[1]  
conn = None  
  
if len(sys.argv) != 2:  
 print(usage)  
 sys.exit(1)  
  
main\_query = """  
select blah  
from blah b   
join what w on something  
order by b.lol  
"""  
  
try:  
 conn = psycopg2.connect("dbname=bank")  
 cur = conn.cursor()  
 cur.execute(main\_query, [name])  
 main\_result = cur.fetchall()  
 except Exception as err:  
 print("DB error: ", err)  
finally:  
 if conn is not None:  
 conn.close()

Aggregate

SQL functions

create or replace function  
 Q5(pattern text)  
returns table(  
 rug text,  
 size\_and\_stoper text,  
 total\_knots numeric(8,0)  
) as $$  
select  
 name as rug,  
 size || 'sf ' || coalesce(rug\_stop::text, '') as size\_and\_stoper,  
 (coalesce(knot\_per\_foot, 50) \* coalesce(knot\_per\_foot, 50) \* size)::numeric(8,0) as total\_knots  
from  
 rugs  
where  
 name ~ pattern;  
$$ language sql;

- First parameter: value so far  
- Second parameter: new value  
 - that's why we don't need to use 2nd parameter in count aggregate

create aggregate concat(text) (  
 stype = text,  
 initcond = '',  
 sfunc = join  
);  
  
create function  
 join(s1 text, s2 text) returns text  
as $$  
begin  
 if (s1 = '') then  
 return s2;  
 else  
 return s1||','||s2;  
 end if;  
end;  
$$ language plpgsql;

create aggregate prod(numeric) (  
 stype = numeric,  
 initcond = 1,  
 sfunc = mult  
);  
  
create function  
 mult(soFar numeric, next numeric) returns numeric  
as $$  
begin   
 return soFar \* next;   
end;  
$$ language plpgsql;

create aggregate myCount(anyelement) (  
 stype = int, -- the accumulator type  
 initcond = 0, -- initial accumulator value  
 sfunc = oneMore -- increment function  
);  
  
create function  
 oneMore(sum int, x anyelement) returns int  
as $$  
begin   
 return sum + 1;  
end;  
$$ language plpgsql;

create or replace view farthest(person,event,location,ordering)  
as (helper view)  
select person, event, location, ordering  
from trail t  
where ordering = (select max(ordering) from trail where person=t.person and event=t.event);  
  
create or replace function q3(\_eventID integer)  
 returns setof text  
as $$  
declare  
 \_last integer;  
 \_tuple record;  
 \_nquitters integer := 0;  
begin  
 perform id from Events where id = \_eventID;  
 if not found then  
 return next 'No such event';  
 return;  
 end if;  
 select max(c.ordering) into \_last  
 from CheckPoints c  
 join Events e on c.route\_id = e.route\_id  
 where e.id = \_eventID;   
  
 for \_tuple in  
 select person,location  
 from farthest  
 where event = \_eventID and ordering < \_last  
 loop  
 return next \_tuple.person || ' gave up at ' || \_tuple.location;  
 \_nquitters := \_nquitters + 1;  
 end loop;  
  
 if \_nquitters = 0 then  
 return next 'Nobody gave up';  
 end if;  
end;  
$$ language plpgsql;

for horse\_id, horse\_name in  
 select id, name  
 from horses  
 where name ~\* part\_name  
 order by name  
loop  
 result.horse := horse\_name;  
 prize\_sum := 0;  
 races\_ran := 0;  
 select race\_count  
 into races\_ran  
 from (  
 select h.name, count(\*) as race\_count   
 from horses h   
 join blah  
 where h.id = horse\_id  
 group by h.name  
 );  
 for prize in (for loop in for loop)  
 select r.prize  
 from horses h   
 join blah   
 where h.id = horse\_id and ru.finished = 1  
 loop  
 prize\_sum := prize\_sum + prize;  
 end loop;  
 average := prize\_sum / races\_ran;  
 result.average = average;  
 return next result;  
end loop;

create or replace function q4()   
returns setof SongCounts  
as $$  
declare  
result SongCounts;  
begin  
for group\_id, group\_name in  
~  
loop  
result."group" := group\_name;  
for album\_id in  
~  
loop  
~  
end loop;  
result.nshort := nshort;  
result.nlong := nlong;  
return next result;

create or replace function q4()   
returns setof SongCounts  
as $$  
declare  
result SongCounts;  
begin  
for group\_id, group\_name in  
~  
loop  
result."group" := group\_name;  
for album\_id in  
~  
loop  
~  
end loop;  
result.nshort := nshort;  
result.nlong := nlong;  
return next result;

select distinct -- distinct select  
 b.location,  
 c.lives\_in  
from customers c

select distinct -- distinct select  
 b.location,  
 c.lives\_in  
from customers c

string\_agg(expression, delimiter [order by expression])

select  
 department,  
string\_agg(employee\_name, ', ' order by employee\_name) as employee\_list  
from  
 employees  
group by  
 department;

select   
 r.name, -- case example  
 count(case when h.gender = 'S' then 1 end) as s\_count,  
 count(case when h.gender = 'G' then 1 end) as g\_count,

select   
p.id, -- string concat  
(p.street\_no || ' ' || st.name || ' ' || st.stype) as street\_name,  
su.name  
from properties p

case -- case usage  
when instrument ilike '%guitar%' then 'guitar'  
when instrument in ('keyboard', 'piano') then 'piano'  
else instrument  
end

having ( -- advanced(?) having  
 sum(s.length) = (  
 select max(total\_length)  
 from (  
 select sum(s1.length) as total\_length  
 from groups g1   
 group by a1.id

Triggers

select unit\_no, street\_no, street, ptype  
into unit\_num, street\_num, street\_id, property\_type  
from properties  
where id = propID;

Triggers

return; exits function without returning further values

P.S. returns not needed for after triggers

Meaning of ‘new’ and ‘old’ in triggers  
create trigger q9\_2  
after update of name on groups  
-> if ‘of name’ exists, only triggered when ‘name’ column is updated.

nullif(val1,val2)  
-> returns null if val1 is equal to val2  
insert into employees (id, name, department, salary) values (101, 'Alice Johnson', 'HR', 60000);  
update employees set salary = 65000 where id = 101;  
delete from employees where id = 101;

|  |  |  |  |
| --- | --- | --- | --- |
|  | Insert | Update | Delete |
| NEW | NewRow | NewRow | X |
| OLD | X | OldRow | OldRow |

if TG\_OP = 'INSERT' then  
 fno := new.flight\_no  
 else  
 fno := old.flight\_no  
 end if;  
  
 if TG\_OP = 'DELETE' then  
 return old; -> for ‘before delete’  
 else  
 if flight.nbooked = flight.nseats then  
 raise exception 'Booking error';  
 end if; -> raise exception for befores  
 return new;-> for ‘before insert & update’

end if;

Flights(fid, from, to, distance, departs, arrives, price)  
Aircraft(aid, aname, range)  
Certified(employee, aircraft)  
Employees(eid, ename, salary)  
  
fid, aid, eid are primary keys.   
from,to -> distance violates BCNF, we need new table  
  
new schema:  
Flights(fid,from,to,departs,arrives,price)  
Routes(from,to,distance)  
Aircraft(aid,aname,range)  
Certified(employee,aircraft)  
Employees(eid,ename,salary)

Non-BCNF schema to BCNF schema

Triggers – what to do:  
1. Split between ‘before’ and ‘after’ triggers  
2. Split again by analysing the requirements  
  
Before triggers: Raise exceptions (usually)  
After triggers: Update values  
  
create trigger TotalSalary2  
after update on Employee  
for each row execute procedure totalSalary2();  
  
create function totalSalary2() returns trigger  
as $$  
begin  
 update Department  
 set totSal = totSal + new.salary  
 where Department.id = new.dept;  
  
 update Department  
 set totSal = totSal - old.salary  
 where Department.id = old.dept;  
 return new;  
end;  
$$ language plpgsql;

create or replace function  
 Q6(pattern text)  
returns table(  
 province text,  
 first integer,  
 nrugs integer,  
 rating numeric(3,1)  
) as $$  
select  
 l.province,  
 min(r.year\_crafted) as first,  
 count(r.id)::integer as nrugs,  
 avg(r.rating)::numeric(3,1) as rating  
from locations l  
join factories f on l.id = f.located\_in  
join crafted\_by cb on f.id = cb.factory  
join rugs r on cb.rug = r.id  
where l.province ~\* pattern  
group by l.province;

$$ language sql;