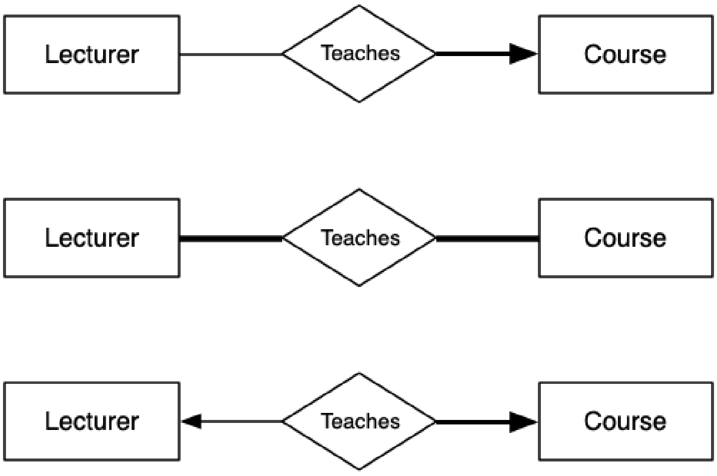
Week1: Data modelling, ER, Relational (tuple == row) relationship != relation

Data modelling works from a description of the *requirements* and aims to build a comprehensive description of the entities involved in the application and the relationships among these entities. The model constructed should ensure that all of the requirements can be met i.e. that all of the relevant data is represented and is structured in such a way that all of the operations mentioned in the requirements can be carried out.

Duplicate tuples are not allowed in relations since it's a set

Semantics of the following relationships ...



A lecturer may teach one course
Every course is taught by 1 or more lecturers

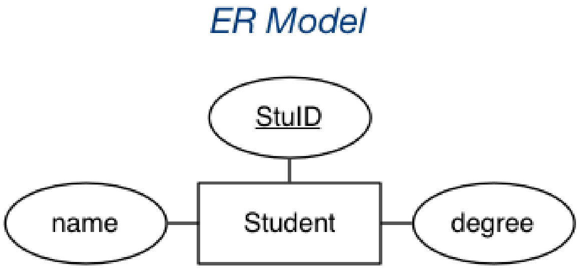
Every lecturer teaches one or more courses
Every course is taught by 1 or more lecturers

Every lecturer may teach one course
Every course is taught by one lecturer

Entity relationship model: entities, relationships, attributes

Relationship constraints: total/partial, n:m/1:n/1:1

Mapping strong entities



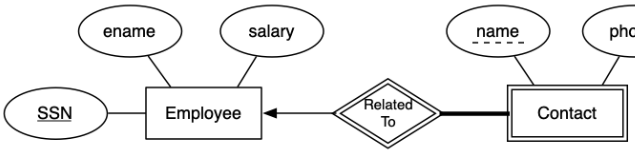
Relational Version

Student

StuID	name	address

Mapping weak entities

ER Model



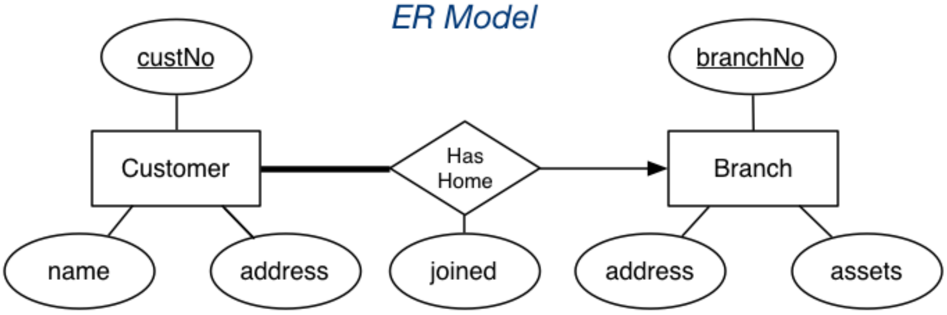
Relational Version

Employee	**SSN**	ename	salary
Contact	**SSN**	**name**	phone

Mapping M:N: customers own accounts-> Owns(custNO, acctNo, lastAccessed)

Mapping 1:N:

ER Model

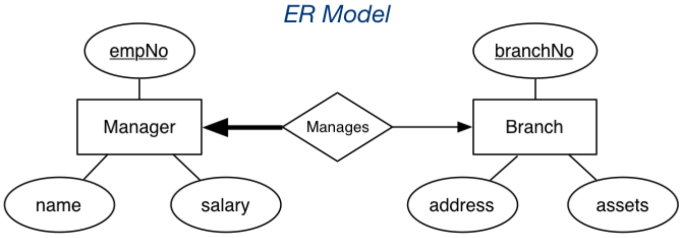


Relational Version

Customer	**custNo**	name	address	branchNo	joined
Branch	**branchNo**	address	assets		

Mapping 1:1: put the fk(pk of one entity) into the one has total participation

ER Model



Relational Version

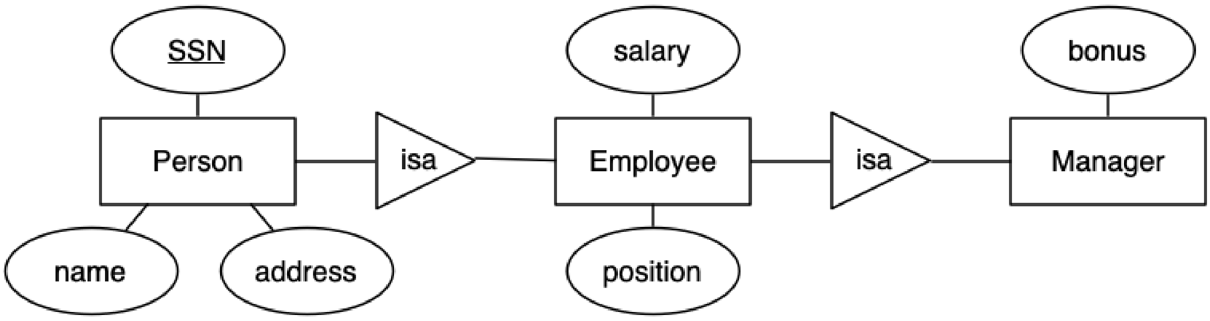
Manager	**empNo**	name	salary	branchNo
Branch	**branchNo**	address	assets	

Mapping Multi-values attribute: FavColour(PersonID, colour)

-fav colour is mv attribute

Mapping subclass ER style

ER Model



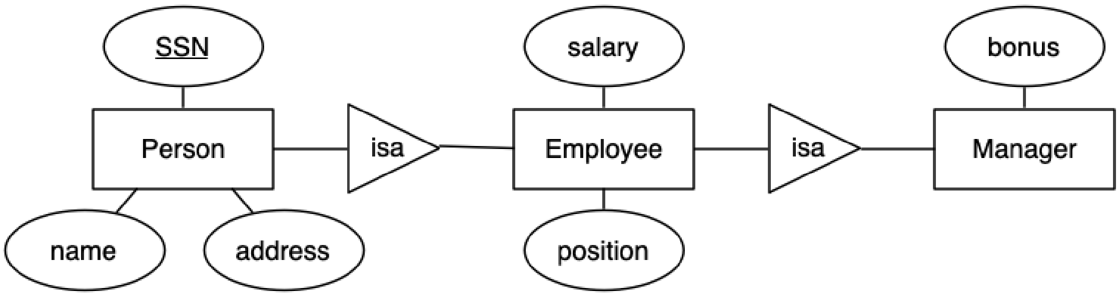
Relational Version

Person	**SSN**	name	address
Employee	**SSN**	salary	position
Manager	**SSN**	bonus	

For composite attribute, better to put several attribute in the table

Mapping subclass O-O style

ER Model

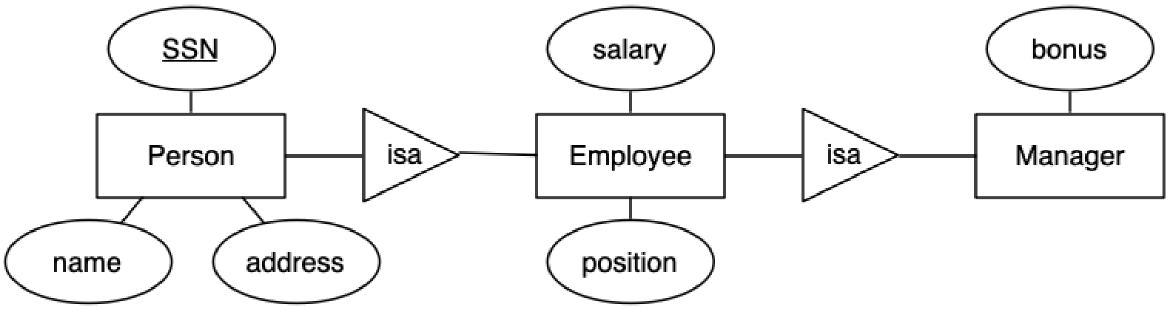


Relational Version

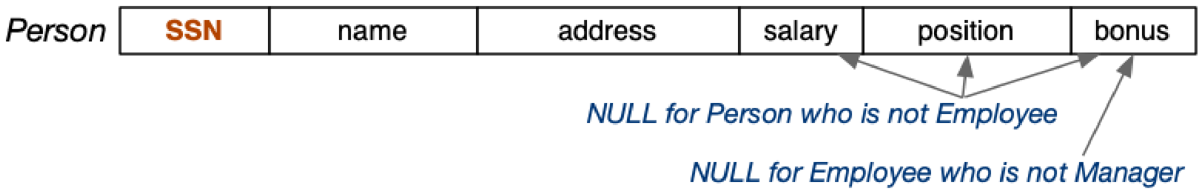
Person	**SSN**	name	address			
Employee	**SSN**	name	address	salary	position	
Manager	**SSN**	name	address	salary	position	bonus

Mapping subclasses single-table-with-nulls

ER Model



Relational Version



Person	**SSN**	name	address	salary	position	bonus

NULL for Person who is not Employee

NULL for Employee who is not Manager

Week02 SQL intro, expression

- meta-data definition language (e.g. **`create table`**, etc.)
- meta-data update language (e.g. **`alter table, drop table`**)
- data update language (e.g. **`insert, update, delete`**)
- query language (e.g. **`select ... from ... where`**, etc.)

Comment: --

'\n' → e'n'

Types: **`integer`**, **`float`**, **`char`**(*n*), **`varchar`**(*n*), **`date, text, currency`**

Operators: **=**, **<>**, **<**, **<=**, **>**, **>=**, **AND**, **OR**, **NOT**, ...

```
'John'    'some text'    '!%#%!$'
'O''Brien'
'"'    '[A-Z]{4}\d{4}'    'a VeRy! LoNg
String'
```

```
E'\n'   E'O\'Brien'   E'[A-
Z]{4}\\d{4}'   E'John'
```

Date, time, timestamp, interval

```
'2008-04-13'  '13:30:15'  '2004-10-19
10:23:54'
'Wed Dec 17 07:37:16 1997 PST'
'10 minutes'  '5 days, 6 hours, 15
seconds'
```

Type-casting: ‘10’ :: integer, now()::TIMESTAMP

```
create domain TaxFileNum as char(11)
           check (value ~ '^[0-9]{3}-[0-9]{3}-[0-9]{3}$');
create domain ISBNumber as char(15)
           check (value ~ '^[A-Z][0-9]{3}-[0-9]{4}-[0-9]{5}$');
```

```
CREATE DOMAIN PosInt AS integer CHECK (value > 0);
-- a UNSW course code
CREATE DOMAIN CourseCode AS char(8)
    CHECK (value ~ '[A-Z]{4}[0-9]{4}');
-- a UNSW student/staff ID
CREATE DOMAIN ZID AS integer
    CHECK (value betweem 1000000 and 9999999);
-- standard UNSW grades (FL,PS,CR,DN,HD)
CREATE DOMAIN Grade AS char(2)
    CHECK (value in ('FL','PS','CR','DN','HD'));
-- or
CREATE TYPE Grade AS ENUM ('FL','PS','CR','DN','HD')
```

% → .* in reg expression

_ → . In reg expression

name LIKE 'Ja%' == name ~ '^Ja'	**name** begins with 'Ja'
name LIKE '_i%' == name ~ '^.i'	**name** has 'i' as 2nd letter
name LIKE '%o%o%' == ~'.*o.*o.*'	**name** contains two 'o's
name LIKE '%ith' == name ~'ith$'	**name** ends with 'ith'
name LIKE 'John' == name ~'John'	**name** equals 'John'/ cont

Case- insensitive matching: ~* / !~*

SQL Operators

- str_1 **||** str_2 ... return concatenation of str_1 and str_2
- **`lower(`***str***`)`** ... return lower-case version of *str*
- **`substring(`***str*,*start*,*count***`)`** ... extract substring from *str*

a	*b*	*a* **AND** *b*	*a* **OR** *b*
TRUE	TRUE	TRUE	TRUE
TRUE	FALSE	FALSE	TRUE
TRUE	NULL	NULL	TRUE
FALSE	FALSE	FALSE	FALSE
FALSE	NULL	FALSE	NULL
NULL	NULL	NULL	NULL

A IS NULL, A IS NOT NULL eg. nullif(mark, '??')

```
CREATE DOMAIN GenderType AS
       char(1) CHECK (value in ('M','F'));

CREATE TABLE Students (
    zid     serial,
    family  text,
    given   varchar(40) NOT NULL,
    code    char(8) NOT NULL CHECK (code ~ '[A-
Z][4][0-9][4]')
    d_o_b   date NOT NULL,
    gender  char(1) CHECK (gender in ('M','F')),
    degree  integer,
    PRIMARY KEY (zid),
    FOREIGN KEY (degree) REFERENCES Degrees(did)
    timestamp default now()
);
CASE
   WHEN test1 THEN result1
   WHEN test2 THEN result2
   ...
   ELSE result_n
END
```

```
INSERT INTO Accounts(acctNo, owner, branch, balance)
           VALUES ('A-123', 765432, 'Nowhere', 5000);
```

SQL statements:

- **CREATE TABLE** *table* **(** *Attributes+Constraints* **)**
- **ALTER TABLE** *table TableSchemaChanges*
- **DROP TABLE** *table(s)* [**CASCADE**]
- **TRUNCATE TABLE** *table(s)* [**CASCADE**]
- **INSERT INTO** *table* **(** *Attrs* **) VALUES** *Tuple(s)*
- **DELETE FROM** *table* **WHERE** *condition*
- **UPDATE** *table* **SET** *AttrValueChanges* **WHERE** *condition*

Mapping ER to SQL

- stop using upper-case for SQL keywords (use **table** vs **TABLE**)
- all tables based on entities are given plural names
- attributes in entities are given the same name in ER and SQL
- attributes in relationships are given the same name in ER and SQL
- ER key attributes are defined using **primary key**
- text-based attributes are defined with type **text**,
  unless there is a size which is obvious from the context
- attribute domains can be PostgreSQL-specific types where useful
- foreign keys within entity tables are named after the relationship
- foreign keys in relationship tables are named ***table*_id**

Subclass

```
create table People (
    ssn      integer primary key,
    name     text not null,
    address text
);
create table Employees (
    person_id integer primary key,
    salary    currency not null,
    position  text not null,
    foreign key (person_id) references
People(ssn)
);
create table Managers (
    employee_id integer primary key,
    bonus       currency,
    foreign key (employee_id)
                references Employees(person_id)
);
```

Week03: SQL

SQL sample database updating: create table, drop table, alter table

Insert, delete, update new tuples

Insertion

```
INSERT INTO RelationName
VALUES (val1, val2, val3, ...)

INSERT INTO RelationName(Attr1, Attr2, ...)
VALUES (valForAttr1, valForAttr2, ...)

INSERT INTO RelationName
VALUES Tuple1, Tuple2, Tuple3, ...

INSERT INTO Likes VALUES ('Justin','Old');
-- or --
INSERT INTO Likes(drinker,beer)
VALUES('Justin','Old');
-- or --
INSERT INTO Likes(beer,drinker)
VALUES('Old','Justin');
```

ALTER TABLE XXX

ALTER COLUMN XXX SET DEFAULT ‘xxx’/SET NOT NULL;

```
ALTER TABLE Likes
   ALTER COLUMN drinker SET DEFAULT 'Joe';
```

Deletion/ Updates

DELETE FROM table_Relation where xxx = ‘xxx’;

Eg. Delete from Sells where price > 100;

UPDATE table_relation Set xxx where xxx;

Eg. UPDATE Drinkers SET phone = ‘0470397745’, age = ‘20’

where name = ‘Jay’;

UPDATE Sells set price = price * 0.9;

Important: SQL QUERY LANGUAGE !!!!!!!!

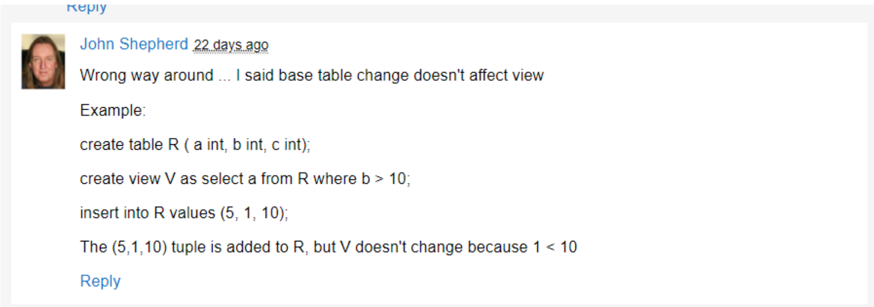
```
SELECT    projectionList
FROM      relations/joins
WHERE     condition
GROUP BY  groupingAttributes
HAVING    groupCondition
```

```
SELECT   s.id, s.name, avg(e.mark) as avgMark
FROM     Students s
         JOIN Enrolments e on (s.id = e.student)
GROUP BY s.id, s.name
```

```
CREATE VIEW ViewName AS Query

CREATE VIEW ViewName (AttributeNames) AS Query

DROP VIEW ViewName
```



Reply

John Shepherd 22 days ago

Wrong way around ... I said base table change doesn't affect view

Example:

create table R (a int, b int, c int);

create view V as select a from R where b > 10;

insert into R values (5, 1, 10);

The (5,1,10) tuple is added to R, but V doesn't change because 1 < 10

Reply

```
JOIN: (left outer, right outer)
SELECT Attributes
FROM   R1
       JOIN R2 ON (JoinCondition1)
       JOIN R3 ON (JoinCondition2)
       ...
WHERE  Condition
```

subqueries:

SELECT * FROM R where R.a = (SELECT XX From xx where xxx);

SELECT * FROM R where R.a IN (SELECT XXX FROM XXX WHERE XXX);

Week04 SQL/PLPGSQL

SQL query results are actually bags, allowing duplicates

Select DISTINCT age from Students;

```
(1,2,3) UNION (2,3,4)  yields  (1,2,3,4)
(1,2,3) UNION ALL (2,3,4)  yields
(1,2,3,2,3,4)
```

In operator

Select name, brewer from Beers where name IN

(select beer from likes where drinker = ‘JAY’);

Find the beers that are unique beer by their manufacturer

SELECT name, brewer From Beers b1

WHERE **not exists**

(select * from Beers b2

where b2.name <> b1.name AND b2.brewer = b1.brewer);

Find the beers sold for the highest price

Select beer from Sells WHERE price >= ALL(SELECT price from sells);

Select beer from Sells where price = (select max(price) from sells);

Find bars sell all of beers Justin like

SELECT distinct a.bar from Sells a

Where NOT EXIST ((SELECT BEER FROM LIKES WHERE DRINKER = 'JUSTIN') EXCEPT (SELECT BEER FROM Sells b where bar = a.bar));

```
(SELECT drinker, beer FROM Likes)
INTERSECT
(SELECT drinker,beer
 FROM   Sells natural join Frequents);

 drinker |      beer
---------+-----------------
 Adam    | New
 John    | Three Sheets
 Justin  | Victoria Bitte
SELECT DISTINCT a.bar
FROM   Sells a
WHERE  NOT EXISTS (
           (SELECT beer FROM Likes
            WHERE drinker = 'Justin')
           EXCEPT
           (SELECT beer FROM Sells b
            WHERE bar = a.bar)
       );
```

SUM/AVG/MIN/MAX

Group by:

select brewer, count(name) as nbeers from Beers Group by brewer;

- Attribute have to be in aggregation operator or in the Group-by clause

Number of styles from brewers who make at least 5 beers

```
SELECT   brewer, count(name) as nbeers,
         count(distinct style) as nstyles
FROM     Beers
GROUP BY brewer
HAVING   count(name) > 4
```

```
 ORDER BY brewer;
```

Partitions

```
SELECT city, date, temperature
       min(temperature) OVER (PARTITION BY city) as
lowest,
       max(temperature) OVER (PARTITION BY city) as
highest
FROM  Weather;
```

--- less tuple

Select city, min(temp), max(temp) From weather group by city;

Using view for abstraction (selectinto)

```
SELECT course, student, mark
FROM   (SELECT course, student, mark,
               avg(mark) OVER (PARTITION BY course)
        FROM   Enrolments) AS CourseMarksWithAvg
WHERE  mark < avg;
```

```
WITH CourseMarksWithAvg AS
     (SELECT course, student, mark,
             avg(mark) OVER (PARTITION BY course)
      FROM   Enrolments)
SELECT course, student, mark, avg
FROM   CourseMarksWithAvg
WHERE  mark < avg;
```

SQL FUNCTION:

```
CREATE OR REPLACE
   funcName(arg1type, arg2type, ....)
   RETURNS rettype
AS $$
   SQL statements
$$ LANGUAGE sql;
```

```
create view Cheapest(bar, price) as
select bar, min(price) from Sells group by bar;
select s.*
from   Sells s
where  s.price =
         (select price from Cheapest where bar =
s.bar);
```

```
create or replace
    function LowestPriceAt(text) returns float
as $$
select min(price) from Sells where bar = $1;
$$ language sql;

select * from Sells where price = LowestPriceAt(bar);
```

```
create or replace view EmpList
as
select given||' '||family as name,
       street||', '||town as addr
from   Employees;
```

PLpgSQL Functions

```
CREATE OR REPLACE
   funcName(param1, param2, ....)
   RETURNS rettype
AS $$
DECLARE
   variable declarations
BEGIN
   code for function
END;
$$ LANGUAGE plpgsql;
```

```
Data Types:
quantity    INTEGER;
start_qty   quantity%TYPE;

employee    Employees%ROWTYPE;
-- or
employee    Employees;

name        Employees.name%TYPE;
```

```
create or replace function
   factorial(n integer) returns integer
as $$
declare
   i integer;
   fac integer := 1;
begin
   for i in 1..n loop
      fac := fac * i;
   end loop;
   return fac;
end;
$$ language plpgsql;
```

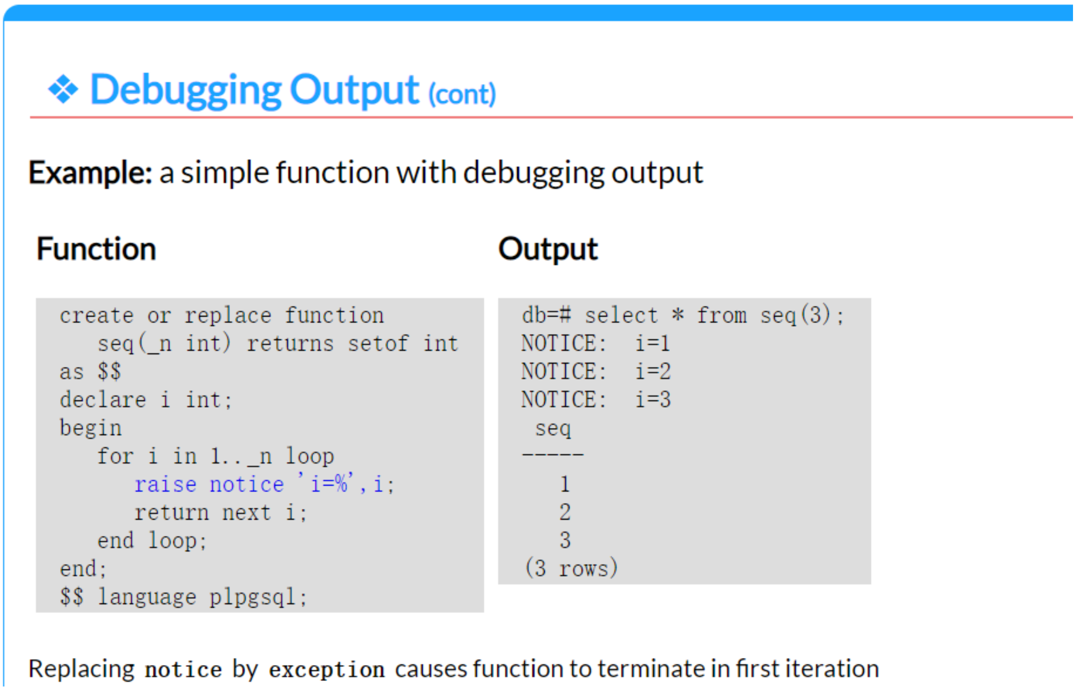
SELEC…INTO (select a into b from R where…; if (not found) then….)

```
   select * into emp
   from Employees where id = 966543;
   eName := emp.name;
```

Week05 plpgsql, constraints, triggers, aggregates

Setting retype to void means "no return value"

Debugging output: raise notice



❖ Debugging Output (cont)

Example: a simple function with debugging output

Function

```
create or replace function
   seq(_n int) returns setof int
as $$
declare i int;
begin
   for i in 1.._n loop
      raise notice 'i=%',i;
      return next i;
   end loop;
end;
$$ language plpgsql;
```

Output

```
db=# select * from seq(3);
NOTICE:  i=1
NOTICE:  i=2
NOTICE:  i=3
 seq
-----
   1
   2
   3
(3 rows)
```

Replacing `notice` by `exception` causes function to terminate in first iteration

Functions can return a set of values (setof Type)

Eg. integer, float, numeric, date, text. varchar(n)

```
create type MyPoint as (x integer, y integer);

create or replace function
   points(n integer, m integer) returns setof MyPoint
as $$
declare
   i integer;  j integer;
   p MyPoint;  -- tuple variable
begin
   for i in 1 .. n loop
      for j in 1 .. m loop
         p.x := i;  p.y := j;
         return next p;
      end loop;
   end loop;
end;
$$ language plpgsql;
```

Insert..returning ➔ useful for recoding id values generated for serial Pks;

```
declare newid integer; colour text;
...
insert into T(id,a,b,c) values (default,2,3,'red')
returning id,c into newid,colour;
-- id contains the primary key value
-- for the new tuple T(?,2,3,'red')
```

Debug: raise debug1, log, info, notice, warning, exception

```
declare
   x integer := 3;
   y integer;
begin
   update T set firstname = 'Joe'
   where lastname = 'Jones';
   -- table T now contains ('Joe','Jones')
   x := x + 1;
   y := x / 0;
exception
   when division_by_zero then
      -- update on T is rolled back to ('Tom','Jones')
      raise notice 'caught division_by_zero';
      raise debug1 'hellooooo';
```

```
     return x; -- value returned is 4
end;
```

For loop of the query(table)

```
create or replace function
   well_paid(_minsal integer) returns integer
as $$
declare
   nemps integer := 0;
   tuple record;  -- could also be tuple Employees;
begin
   for tuple in
      select * from Employees where salary > _minsal
   loop
      nemps := nemps + 1;
   end loop;
   return nemps;
end;
$$ language plpgsql;
```

EXECUTE takes a string and excutes it as an sql query

```
create or replace function
   set(_table text, _attr text, _val text) returns
void
as $$
declare
   query text;
begin
   query := 'update ' || quote_ident(_table);
   query := query || ' SET ' || quote_ident(_attr);
   query := query || ' = ' || quote_literal(_val);
   execute query;

   for xxx in execute query
   loop
   end loop;
end;
$$ language plpgsql;
```

Constraints/Assertions

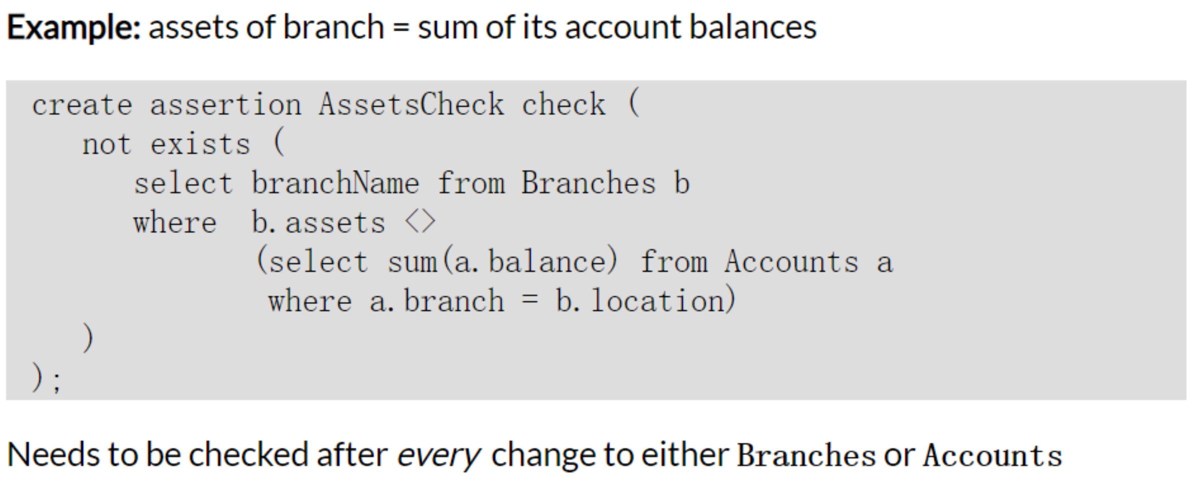
Attribute constrains: integer, varchar(30), integer check (age > 15) etc.

Relation contrainst: primary key, constrinat xxxx check (salary > age * 100)

Referential integrity constraints (xxxx integer references yyyy(id))

Example: #students in any UNSW course must be < 10000

```
create assertion ClassSizeConstraint check (
   not exists (
      select c.id
      from   Courses c
             join Enrolments e on (c.id = e.course)
      group  by c.id
      having count(e.student) > 9999
   )
);
```



Example: assets of branch = sum of its account balances

```
create assertion AssetsCheck check (
   not exists (
      select branchName from Branches b
      where  b.assets <>
             (select sum(a.balance) from Accounts a
              where a.branch = b.location)
   )
);
```

Needs to be checked after *every* change to either `Branches` or `Accounts`

Triggers

```
CREATE TRIGGER TriggerName
{AFTER|BEFORE}  Event1 [ OR Event2 ... ]
[ FOR EACH ROW ]
ON TableName
[ WHEN ( Condition ) ]
Block of Procedural/SQL Code ;
```

```
create trigger checkState before insert or update
on Person for each row execute procedure
checkState();
```

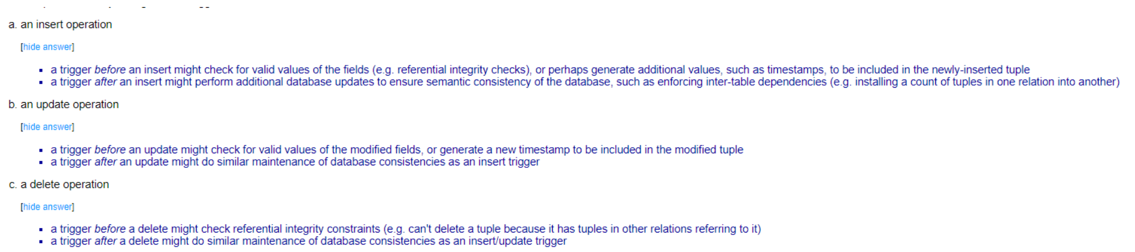
```
create function checkState() returns trigger as $$
begin
   -- normalise the user-supplied value
   new.state = upper(trim(new.state));
   if (new.state !~ '^[A-Z][A-Z]$') then
      raise exception 'Code must be two alpha chars';
   end if;
   -- implement referential integrity check
   select * from States where code=new.state;
   if (not found) then
      raise exception 'Invalid code %',new.state;
   end if;
   return new;
end;
$$ language plpgsql;
```

Before triggers, modify any new tuple, After triggers, update other tables related

Before triggers must contain ‘return old ’ or ‘return new’

Return old, no change occurs, exception raised, no change occurs.

Insert no old, delete no new



a. an insert operation

[hide answer]

- a trigger *before* an insert might check for valid values of the fields (e.g. referential integrity checks), or perhaps generate additional values, such as timestamps, to be included in the newly-inserted tuple
- a trigger *after* an insert might perform additional database updates to ensure semantic consistency of the database, such as enforcing inter-table dependencies (e.g. installing a count of tuples in one relation into another)

b. an update operation

[hide answer]

- a trigger *before* an update might check for valid values of the modified fields, or generate a new timestamp to be included in the modified tuple
- a trigger *after* an update might do similar maintenance of database consistencies as an insert trigger

c. a delete operation

[hide answer]

- a trigger *before* a delete might check referential integrity constraints (e.g. can't delete a tuple because it has tuples in other relations referring to it)
- a trigger *after* a delete might do similar maintenance of database consistencies as an insert/update trigger

Case 1: new employees arrive

```
create trigger TotalSalary1
after insert on Employees
for each row execute procedure totalSalary1();

create function totalSalary1() returns trigger
as $$
begin
    if (new.dept is not null) then
        update Department
        set    totSal = totSal + new.salary
        where  Department.id = new.dept;
    end if;
    return new;
end;
$$ language plpgsql;
```

Case 2: employees change departments/salaries

```
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;
```

Case 3: employees leave

```
create trigger TotalSalary3
after delete on Employee
for each row execute procedure totalSalary3();

create function totalSalary3() returns trigger
as $$
begin
    if (old.dept is not null) then
        update Department
        set    totSal = totSal - old.salary
        where  Department.id = old.dept;
    end if;
    return old;
end;
$$ language plpgsql;
```

Note: in the solution below, `TG_OP` is a special variable that tells the trigger function which operation caused it to be invoked. This is useful when a trigger is defined to act on more than one type of operation (as in the triggers below).

```
create trigger R_pk_check before insert or update on R
for each row execute procedure R_pk_check();

create function R_pk_check() returns trigger
as $$
```

```
begin
     if (new.a is null or new.b is null) then
            raise exception 'Partially specified primary key for
R';
     end if;
     if (TG_OP = 'UPDATE' and old.a=new.a and old.b=new.b) then
            return;
     end if;
    -- not UPDATE, so must be INSERT
     select * from R where a=new.a and b=new.b;
     if (found) then
            raise exception 'Duplicate primary key for R';
     end if;
end;
$$ language plpgsql;
```

Create or define new Aggregates

```
CREATE AGGREGATE AggName(BaseType) (
    sfunc     = UpdateStateFunction,
    stype     = StateType,
    initcond  = InitialValue,
    finalfunc = MakeFinalFunction,
    sortop    = OrderingOperator
);
```

- **`initcond`** (type *StateType*) is optional; defaults to **`NULL`**
- **`finalfunc`** is optional; defaults to identity function
- **`sortop`** is optional; needed for min/max-type aggregates

```
aggregate Agg : setof BaseType → ResultType
sfunc : StateType,BaseType → StateType
finalfunc : StateType → ResultType
```

How they work together:

```
S : StateType
S = initcond
for each value V in column A of relation R {
   S = sfunc(S, V)
}
```

```
return finalfunc(S)
create type StateType as ( sum numeric, count numeric );create
function include(s StateType, v numeric) returns StateType
as $$
begin
   if (v is not NULL) then
      s.sum := s.sum + v;
      s.count := s.count + 1;
   end if;
   return s;
end;
$$ language plpgsql;

create or replace function compute(s StateType) returns numeric
as $$
begin
   if (s.count = 0) then
      return null;
   else
      return s.sum::numeric / s.count;
   end if;
end;
$$ language plpgsql;

create aggregate mean(numeric) (
    stype     = StateType,
    initcond  = '(0,0)',
    sfunc     = include,
    finalfunc = compute
);
```

```
select sum(a)::numeric/count(a) from R;
```

Example: defining the **`count`** aggregate (roughly)

```
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;
```

Example: **`sum2`** sums two columns of integers

```
create type IntPair as (x int, y int);

create function
   addPair(sum int, p IntPair) returns int
as $$
begin return sum + p.x + p.y; end;
$$ language plpgsql;

create aggregate sum2(IntPair) (
   stype     = int,
   initcond  = 0,
   sfunc     = addPair
);
```

Product aggregate of a eg. select prod(*) from (1,2,3,4,5) = 120

```
create aggregate prod(numeric) (
    stype    = numeric,    -- the accumulator type
    initcond = 1,      -- initial accumulator value
    sfunc    = mult -- increment function
);

create function
    mult(sofar numeric, next numeric) returns numeric
as $$
begin
     return sofar * next;
end;
$$ language plpgsql;

select count(*), concat(name) from Employee;
-- returns e.g.
  count |         concat
 -------+-----------------------
      4 | John,Jane,David,Phil
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 concat(text) (
   stype    = text,
   initcond = '',
   sfunc    = join
);
```

Week07 DB Programming, Python, Pyscopg2

dbAcess = 500ms, dbQuery = 200ms, dbNext = 10ms

Example: find info about all marks for all students (1000 students each with 8 marks)

```
query1 = "select id,name from Student";
res1 = dbQuery(db, query1);
while (tuple1 = dbNext(res1)) {
    query2 = "select course,mark from Marks"
             + " where student = " + tuple1['id'];
    res2 = dbQuery(db,query2);
    while (tuple2 = dbNext(res2)) {
        --  process student/course/mark info
    }
}
```

Run 10000+1 queries, total cost = 10001 * 200 + 80000 * 10

```
query = "select id,name,course,mark"
        + " from Student s join Marks m "
        + " on (s.id=m.student)"
results = dbQuery(db, query);
while (tuple = dbNext(results)) {
    --  process student/course/mark info
}
```

Run 1 queries, total cost = 1 * 200 + 80000 * 10

// basic python database sample

```
import sys
import psycopg2

if len(sys.argv) < 2:
   print("Usage: opendb DBname")
   exit(1)
db = sys.argv[1]
try:
   conn = psycopg2.connect("dbname="+db)
   print(conn)
   cur = conn.curosr()
   cur.execute("query", [arg1, arg2])
   conn.close()
   conn.commit()
   print(conn)
except Exception as e:
   print(f"Unable to connect to database {db}")
```

Very useful!!!!

print(cur.mogrify(query, [xxx]))

```
cur.execute("select * from R")
result = cur.fetchall()
if (len(result) == 0):
   exit(1)
for tup in result:
   x,y = tup
   print(x,y)
list = cur.fetchall() put all result for a query in a list of
tuples

cur.fetchone()
cur.fetchmany(10)
```

```
cur = conn.cursor()
qry = """
select b.name, r.name
from   Brewers r join Beers b on (b.brewer=r.id)
"""
cur.execute(qry)
for tuple in cur.fetchall():
   print(tuple[1] + " " + tuple[0])
```

Week 8 relational algebra, Functional dependencies

Reflexivity x->x, augmentation x->y => xz->yz, Transitivity x->y, y->z ➔ x->z

Cloursers based on set of attributes rather than set of fds.

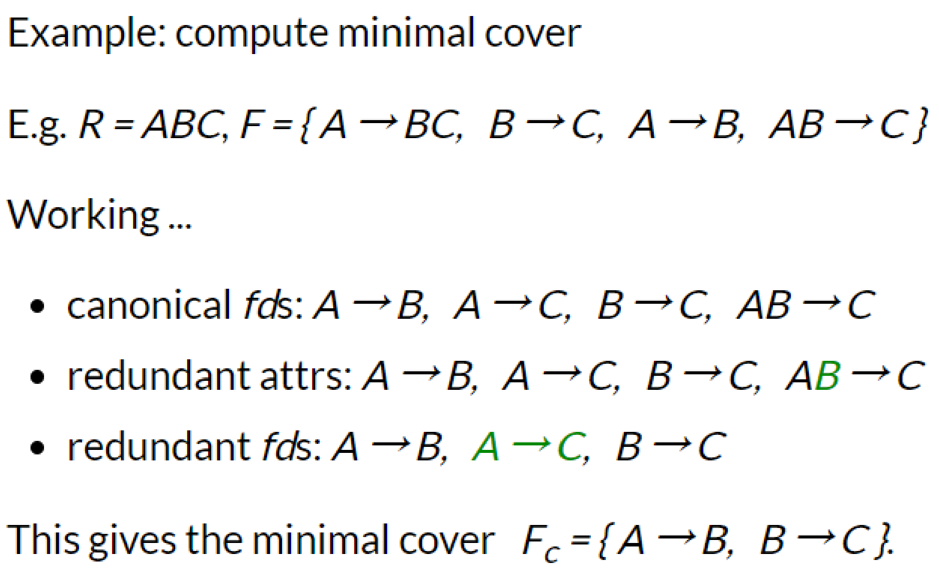
Determine keys

Minimal covers:

Step1: put fds in to canonical form

Step2: eliminate redundant attributes

Step3: eliminate redundant fds



Example: compute minimal cover

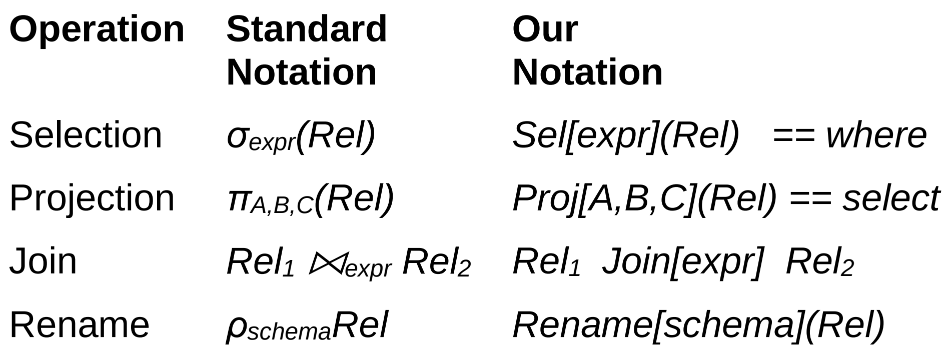
E.g. $R = ABC$, $F = \{ A \rightarrow BC,\ B \rightarrow C,\ A \rightarrow B,\ AB \rightarrow C \}$

Working ...

- canonical *fds*: $A \rightarrow B,\ A \rightarrow C,\ B \rightarrow C,\ AB \rightarrow C$
- redundant attrs: $A \rightarrow B,\ A \rightarrow C,\ B \rightarrow C,\ AB \rightarrow C$
- redundant *fds*: $A \rightarrow B,\ A \rightarrow C,\ B \rightarrow C$

This gives the minimal cover $F_c = \{ A \rightarrow B,\ B \rightarrow C \}$.

Week09 Relation algebra



Operation	Standard Notation	Our Notation
Selection	$\sigma_{expr}(Rel)$	*Sel[expr](Rel)* == *where*
Projection	$\pi_{A,B,C}(Rel)$	*Proj[A,B,C](Rel)* == *select*
Join	$Rel_1 \bowtie_{expr} Rel_2$	*Rel₁ Join[expr] Rel₂*
Rename	$\rho_{schema} Rel$	*Rename[schema](Rel)*

Beers made by Sierra Nevada

```
SNBeers = Sel[manf=Sierra Nevada](Beers)
Result  = Rename[beer](Proj[name](SNBeers))
```

Rename/Selection/projection

Intersection

```
JohnBars = Proj[bar](Sel[drinker=John](Frequents))
GernotBars =
Proj[bar](Sel[drinker=Gernot](Frequents))

Result = JohnBars union GernotBars
```

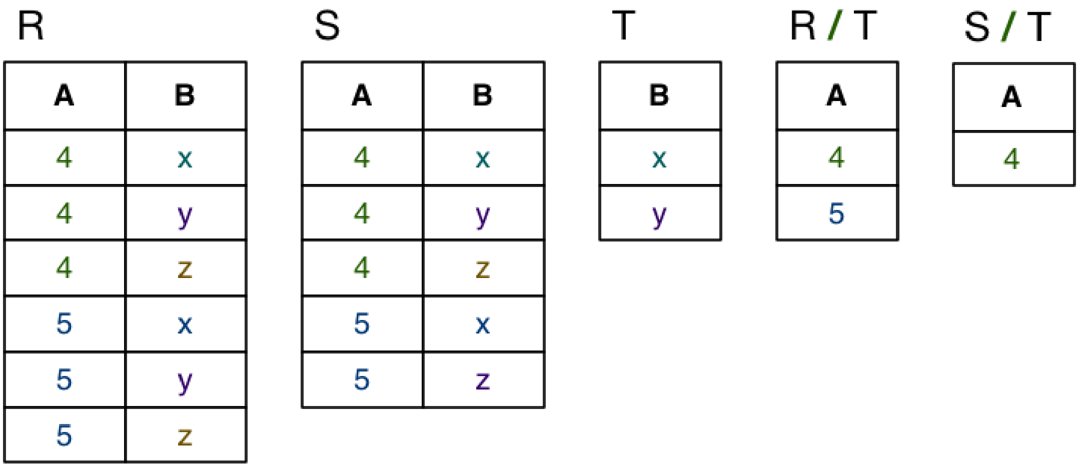
Bars where both John and Gernot drink

```
Result = JohnBars  intersect  GernotBars
```

Bars where Join drinks and Gernot not

```
Result = JohnBars - GernotBars
```

Division



R

A	B
4	x
4	y
4	z
5	x
5	y
5	z

S

A	B
4	x
4	y
4	z
5	x
5	z

T

B
x
y

R / T

A
4
5

S / T

A
4

Querying with relational algebra (division) ...

Division handles queries that include the notion "for all".

E.g. Which beers are sold in all bars?

We can answer this as follows:

- generate a relation of beers and bars where they are sold
  - *r1 = Proj[beer,bar](Sold)*
- generate a relation of all bars
  - *r2 = Rename[r2(bar)](Proj[name](Bars))*
- find which beers appear in tuples with every bar
  - *res = r1 Div r2*

Mapping SQL to relational algebra, e.g.

```
-- schema: R(a,b) S(c,d)
select a as x
from   R join S on (b=c)
where  d = 100
-- could be mapped to
Tmp1(a,b,c,d) = R Join[b=c] S
Tmp2(a,b,c,d) = Sel[d=100](Tmp1)
Tmp3(a)       = Proj[a](Tmp2)
Res(x)        = Rename[Res(x)](Tmp3)
```

In general:

- **`SELECT`** clause becomes *projection*
- **`WHERE`** condition becomes *selection* or *join*
- **`FROM`** clause becomes *join*
- **`Group by becomes GroupCount`**

Find the *sids* of suppliers who supply some red part or whose address is 221 Packer Street.

```
RedPartIds =
Rename[RedPartIds(part)](Proj[pid](Sel[colour='red'](Parts)))
RedPartSupplierIds =
Rename[RedPartSupplierIds(sid)](Proj[supplier](RedPartIds Join
Catalog))
PackerStSupplierIds = Proj[sid](Sel[address='221 Packer
Street'](Suppliers))
Answer = RedPartSupplierIds Union PackerStSupplierIds
```

Example: Select on indexed attribute

```
db=# explain analyze select * from Students where id=100250;
                            QUERY PLAN
------------------------------------------------------------
 Index Scan using student_pkey on student
                  (cost=0.00..5.94 rows=1 width=17)
                  (actual time=3.209..3.212 rows=1 loops=1)
   Index Cond: (id = 100250)
 Total runtime: 3.252 ms
```

Example: Select on non-indexed attribute

```
db=# explain analyze select * from Students where
stype='local';
                         QUERY PLAN
------------------------------------------------------------
-
 Seq Scan on student  (cost=0.00..70.33 rows=18 width=17)
             (actual time=0.061..7.784 rows=2512 loops=1)
   Filter: ((stype)::text = 'local'::text)
 Total runtime: 7.554 ms
```

```
db=# explain select * from Students where stype='local';
```

Week10 Transactions, Serializability, Schedules

- **`READ`** - transfer data item from database to memory EG. SELECT
- **`WRITE`** - transfer data item from memory to database EG. INSERT
- **`BEGIN`** - start a transaction
- **`COMMIT`** - successfully complete a transaction
- **`ABORT`** - fail a transaction and unwind effects
- **`Both READ and WRITE EG.`** UPDATE, DELETE but treat as WRITE

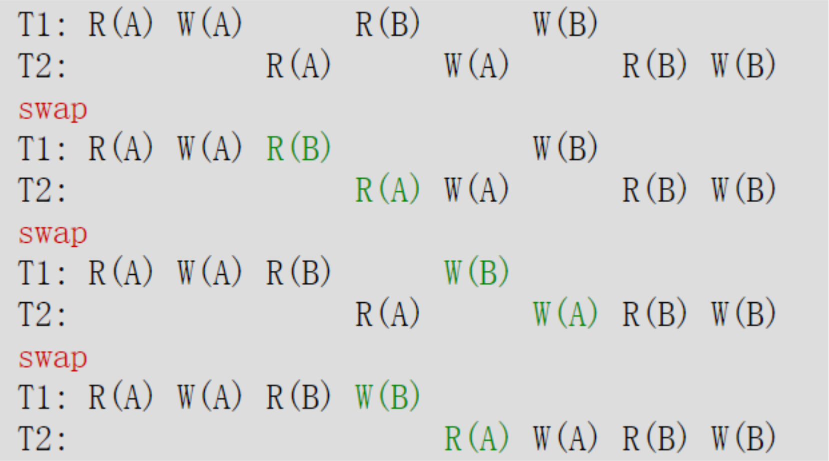
Serial execution: T1 then T2 or T2 then T1

```
T1: R(X) W(X) R(Y) W(Y)
T2:                     R(X) W(X)

T1:           R(X) W(X) R(Y) W(Y)
T2: R(X) W(X)
```

Serializability: Transform a concurrent schedule to serial schedule

Precedence graph



```
T1: R(A) W(A)      R(B)      W(B)
T2:           R(A)      W(A)      R(B) W(B)
swap
T1: R(A) W(A) R(B)           W(B)
T2:                R(A) W(A)      R(B) W(B)
swap
T1: R(A) W(A) R(B)      W(B)
T2:                R(A)      W(A) R(B) W(B)
swap
T1: R(A) W(A) R(B) W(B)
T2:                     R(A) W(A) R(B) W(B)
```

Whether the schedule is conflict-serializable(serializable), if there is a cycle, not conflict-serializablle

```
grieg$ rm -fr /srvr/z5215032/pgsql
grieg$ ~cs3311/bin/pginit
```