Making and editing DBs

cs383, March 2

Attributes
The columns of a table

- Relational DB are composed of a set of tables
  - Each table has a set of columns
  - Those column are referred to as attributes
- The set of allowed values for each attribute is called the **domain** of the attribute
- Attribute values are (normally) required to be **atomic**; that is, indivisible
- The special value ***null*** is a member of every domain. Indicated that the value is "unknown"
- The null value causes complications in the definition of many operations

- Suppose making a Family DB and want to put entire "nuclear" family into one table
  - what attributes are needed??
    - "19 kids and counting"

```
create table nuclearfamily1 (
    parent1 varchar,
    parent2 varchar,
    child1 varchar,
    child2 varchar,
    child3 varchar,
    child4 varchar,
    child5 varchar
);

drop table nuclearfamily1;
```

Adding data to table

- SQL Insert
  - INSERT INTO table_name (column1, column2, column3, ...)
    VALUES (value1, value2, value3, ...);
  - INSERT INTO table_name
    VALUES (value1, value2, value3, ...);

```
insert into nuclearfamily1 (parent1, parent2, child1, child2, child3, child4,
child5) values('a', 'b', 'c', 'd', 'e', 'f', 'g');

insert into nuclearfamily1 values('ab', 'bb', 'cb', 'db', 'eb', 'fb', 'gb');

insert into nuclearfamily1 (parent1, parent2, child1, child2, child4, child5)
values('aa', 'ab', 'ca', 'da', 'fa', 'ga');

insert into nuclearfamily1 (parent1,  child1, child2, child4) values('aa', 'cd',
'dd', 'fd');
```

Modifying data in a table

- DELETE FROM table
  WHERE condition;
  - delete from nuclearfamily1 where parent1='a';
    - DANGER: if you do not include a WHERE clause

- UPDATE table_name
  SET column1 = value1, column2 = value2, ...
  WHERE condition;
  - update nuclearfamily1 set parent1='dd', child1='ff' where parent2='bb';
    - DANGER: if you do not include a WHERE clause

Keys

Cannot use the shortcut insert syntax.
Must list all of the columns
Cannot manually set family_id.

- Keys are important in RDBS
  - They define the unique elements in a single table.
  - They define how tables are related.
  - In DB with single table, keys are less important
- Keys are usually a single attribute, but may be a composite of several attributes
  - Is there a key in the nuclearfamily1 table??

```
create table nuclearfamily1 (
    parent1 varchar,
    parent2 varchar,
    child1 varchar,
    child2 varchar,
    child3 varchar,
    child4 varchar,
    child5 varchar
);
```

```
create table nuclearfamily2 (
    family_id INT GENERATED ALWAYS AS IDENTITY,
    parent1 varchar,
    parent2 varchar,
    child1 varchar,
    child2 varchar,
    child3 varchar,
    child4 varchar,
    child5 varchar
);
```

PostgreSQL only
Other can do same thing, different syntax

Keys

- *K* is a **key** of *R* if values for *K* are sufficient to identify a unique tuple of each possible relation *r(R)*
  - Example: {*family_id*} and {family_id, parent1} are both keys of nuclearfamily2*.*
- key *K* is a **candidate key** if *K* is minimal

  Example: {*family_id*} is a candidate key for nuclearfamily2
  - There may be more than one candidate key in a table. (there may be none)
- One of the candidate keys is selected to be the **primary key**.
  - If there is more than one candidate key need to pick one as a primary key

```
create table nuclearfamily3 (
    family_id INT GENERATED ALWAYS AS IDENTITY,
    parent1 varchar,
    parent2 varchar,
    child1 varchar,
    child2 varchar,
    child3 varchar,
    child4 varchar,
    child5 varchar,
    primary key (family_id)
);
```

Primary Key

- A primary key is a special relational database table column (or combination of columns) designated to uniquely identify each table record.
- A primary key is used as a unique identifier to quickly parse data within the table. A table cannot have more than one primary key.
- A primary key's main features are:
  - It must contain a unique value for each row of data.
  - It cannot contain null values.
  - Every row must have a primary key value.

- Desirable traits of primary keys
  - **Primary keys should be as small as necessary.** Prefer a numeric type because numeric types are stored in a much more compact format than character formats. This is because most primary keys will be foreign keys in another table as well as used in multiple indexes. The smaller your key, the smaller the index, the less pages in the cache you will use.
  - **Primary keys should never change.** Updating a primary key should always be out of the question. This is because it is most likely to be used in multiple indexes and used as a foreign key. Updating a single primary key could cause of ripple effect of changes.
  - **Do NOT use "your problem primary key" as your logic model primary key.** For example passport number, social security number, or employee contract number as these "natural keys" can change in real world situations. Make sure to add UNIQUE constraints for these where necessary to enforce consistency

Foreign Key

- A "constraint" in a database.
- The value the referencig relation must appear in the referenced relation.
  - **Referencing** relation
  - **Referenced** relation

- Problem in the nuclearfamily3 table. Each of the people are references by a name and the name may not unique.   Solution
  - Make a people table and reference that in most columns of the nuclearfamily table.

```
create table people4 (
    person_id INT GENERATED ALWAYS AS IDENTITY,
    name varchar,
    primary key (person_id)
);
```

```
create table nuclearfamily4 (
    family_id INT GENERATED ALWAYS AS IDENTITY,
    parent1 int,
    parent2 int,
    child1 int,
    child2 int,
    primary key (family_id),
    foreign key (parent1) references people4(person_id),
    foreign key (parent2) references people4(person_id),
    foreign key (child1) references people4(person_id),
    foreign key (child2) references people4(person_id)
);
```

Referential Integrity

- Ensures that a value that appears in one relation for a given set of attributes also appears for a certain set of attributes in another relation.
  - Example: If "Biology" is a department name appearing in one of the tuples in the *instructor* relation, then there exists a tuple in the *department* relation for "Biology".
- Let A be a set of attributes. Let R and S be two relations that contain attributes A and where A is the primary key of S. A is said to be a **foreign key** of R if for any values of A appearing in R these values also appear in S.

Referential Integrity (Cont.)

- Foreign *keys can be* specified as part of the SQL **create table** statement

  foreign key (*dept_name*) **references** *department*

- By default, a foreign key references the primary-key attributes of the referenced table.
- SQL allows a list of attributes of the referenced relation to be specified explicitly.

  foreign key (*dept_name*) **references** *department* (*dept_name*)

With foreign keys enforcing integrity
The inserts become harder, as do deletes

ID must exist in people4 before inserting into nuclearfamily4

```
insert into people4(name) values('a');
insert into people4(name) values('b');
insert into people4(name) values('c');
insert into people4(name) values('d');
insert into people4(name) values('e');
insert into people4(name) values('f');
insert into people4(name) values('g');
insert into people4(name) values('h');
insert into people4(name) values('i');
insert into people4(name) values('j');
insert into nuclearfamily4(parent1, parent2, child1, child2) values(1,2,3,4);
insert into nuclearfamily4(parent1, parent2, child1, child2) values(5,6,7,8);
insert into nuclearfamily4(parent1, child1) values(9,10);
```

```
family=# delete from people4 where person_id=10;
ERROR:  update or delete on table "people4" violates foreign key constraint "nuclearfamily4_child1_
on table "nuclearfamily4"
DETAIL:  Key (person_id)=(10) is still referenced from table "nuclearfamily4".
Time: 1.635 ms
```

Cannot delete because this ID is used in nuclearfamily4

Views

- Views allow you to write a query and save it.
  - A view can often be used just like a real table.
  - Some views are made to give a public representation of a table that contains non-public rows

```
create or replace view parents4(p1, p2) as select parent1, parent2 from nuclearfamily4;
```

  - Such views can be updated like a table;

```
family=# update parents4 set p1=2 where family_id=1;
```

Views

- More complex views (involving more than one table) typically do not allow direct updates
- nuclearfamily4 makes sense DB-wise, but is annoying for common users who just want the names of people in each family.
- Create a view showing the names of all family members in the nuclearfamily4 table

```
create or replace view familyname4(id, p1, p2, c1,c2) as
select family_id, p4a.name, p4b.name, p4c.name, p4d.name
from nuclearfamily4
     left outer join people4 as p4a on parent1=p4a.person_id
     left outer join people4 as p4b on parent2=p4b.person_id
     left outer join people4 as p4c on child1=p4c.person_id
     left outer join people4 as p4d on child2=p4d.person_id;
```

materialized views

- Essentially a snapshot of a database.
- Views are generally used when data is to be accessed infrequently and data in table get updated on frequent basis. On other hand Materialized Views are used when data is to be accessed frequently and data in table not get updated on frequent basis.

Integrity Constraints

- Integrity constraints guard against accidental damage to the database, by ensuring that authorized changes to the database do not result in a loss of data consistency.
  - A checking account must have a balance greater than $0.00
  - A salary of a bank employee must be at least $4.00 an hour
  - A customer must have a (non-null) phone number

Constraints on a Single Relation

- **not null**
- **primary key**
- **unique** (multiple items may be null)
- **check** (P), where P is a predicate
- **Default**

The SQL PRIMARY KEY constraint combines the UNIQUE and NOT NULL constraints, where the column or set of columns that are participating in the PRIMARY KEY cannot accept a NULL value. If the PRIMARY KEY is defined in multiple columns, you can insert duplicate values on each column individually, but the combination values of all PRIMARY KEY columns must be unique. Take into consideration that you can define only one PRIMARY KEY per each table, and it is recommended to use small or INT columns in the PRIMARY KEY.

Not Null and default Constraints

- not null -- do not allow null values
- default -- used to give values in place of null
  - commonly paired with not null

```
create table people7 (
    person_id INT GENERATED ALWAYS AS IDENTITY,
    name varchar not null,
    reltype varchar not null default 'child',
    primary key (person_id)
);

family=# insert into people7(name) values('hi');
INSERT 0 1
Time: 101.692 ms
family=# select * from people7;
 person_id | name | reltype
-----------+------+---------
         1 | hi   | child
(1 row)

family=# insert into people7(name) values(null);
ERROR:  null value in column "name" of relation "people7" violates not-null constraint
DETAIL:  Failing row contains (2, null, child).
```

Unique Constraints

- **unique** (A_1, A_2, …, A_m)
  - The unique specification states that the attributes A_1, A_2, …, A_m form a candidate key.
  - Candidate keys are permitted to be null (in contrast to primary keys).

```
create table people5 (
    person_id INT GENERATED ALWAYS AS IDENTITY,
    name varchar,
    primary key (person_id),
    unique (name)
);
```

Why have person_id is name is guaranteed unique??

The check clause

- The **check** (P) clause specifies a predicate P that must be satisfied by every tuple in a relation.
- Example: ensure that semester is one of fall, winter, spring or summer

```
create table people6 (
    person_id INT GENERATED ALWAYS AS IDENTITY,
    name varchar,
    reltype varchar,
    primary key (person_id),
    unique (name),
    check (reltype in ('parent', 'child'))
);

insert into people6(name, reltype) values('az', 'parent');
insert into people6(name, reltype) values('ay', 'parentt');

insert into people5(name, reltype) values('ay', 'parentt');
ERROR:  new row for relation "people5" violates check constraint "people5_reltype_che
DETAIL:  Failing row contains (2, ay, parentt).
```

A complex table

```
univ=# \d section
                        Table "public.section"
    Column     |          Type          | Collation | Nullable | Default
---------------+------------------------+-----------+----------+---------
 course_id     | character varying(8)   |           | not null |
 sec_id        | character varying(8)   |           | not null |
 semester      | character varying(6)   |           | not null |
 year          | numeric(4,0)           |           | not null |
 building      | character varying(15)  |           |          |
 room_number   | character varying(7)   |           |          |
 time_slot_id  | character varying(4)   |           |          |
Indexes:
    "section_pkey" PRIMARY KEY, btree (course_id, sec_id, semester, year)
Check constraints:
    "section_semester_check" CHECK (semester = ANY (ARRAY['Fall', 'Winter', 'Spring', 'Summer']))
    "section_year_check" CHECK (year > 1701 AND year < 2100)
Foreign-key constraints:
    "section_building_room_number_fkey" FOREIGN KEY (building, room_number) REFERENCES
classroom(building, room_number)
    "section_course_id_fkey" FOREIGN KEY (course_id) REFERENCES course(course_id)
```

PostgreSQL rephrasing. Generic SQL would be semester in ('fall', ...)

Large-Object Types

- Large objects (photos, videos, CAD files, etc.) are stored as a *large object*:
  - **blob**: binary large object -- object is a large collection of uninterpreted binary data (whose interpretation is left to an application outside of the database system)
  - **clob**: character large object -- object is a large collection of character data
  - PostgreSQL does not have these, but something equivalent
- When a query returns a large object, a pointer is returned rather than the large object itself.

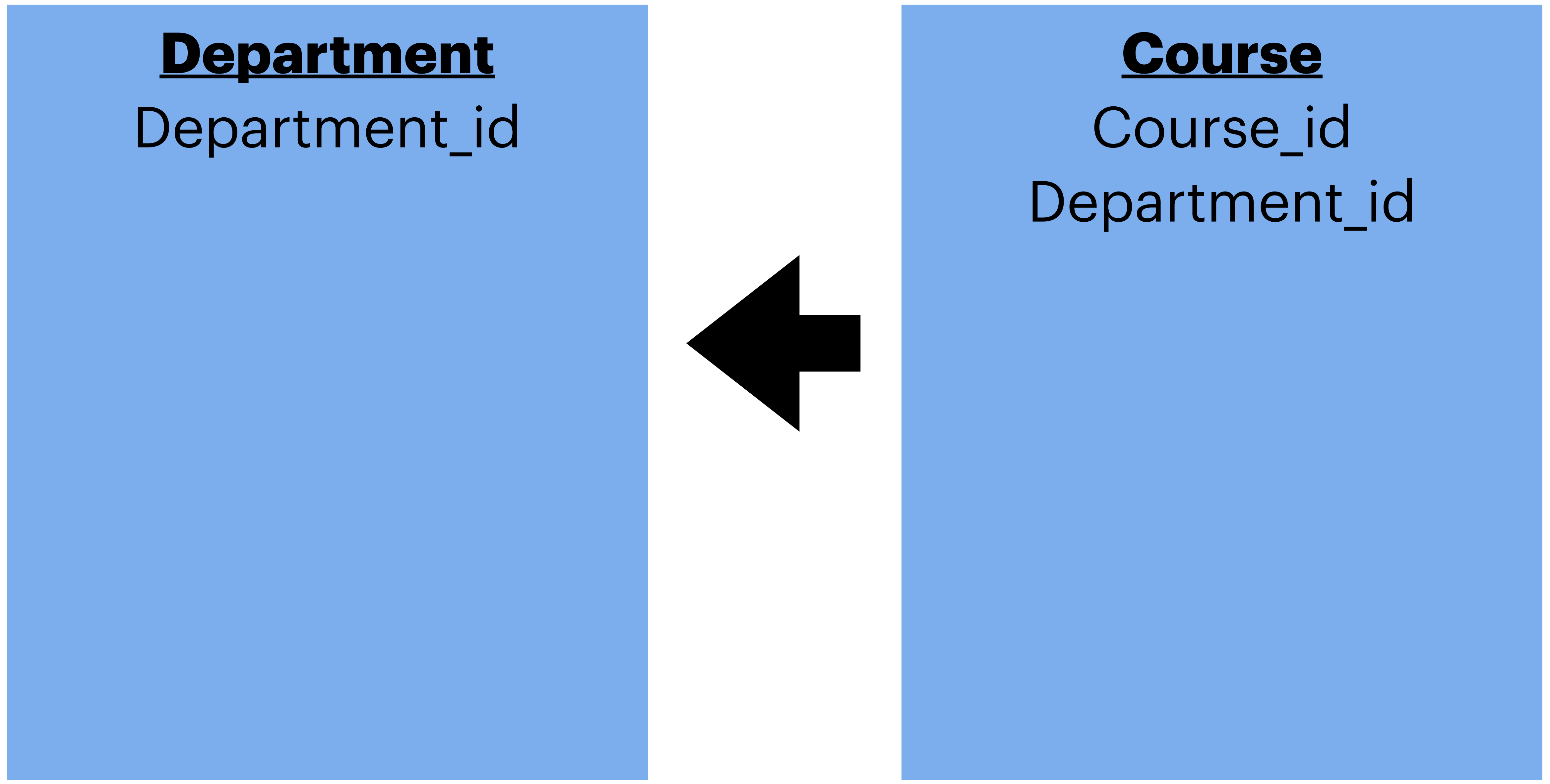
- It is common to store large blobs in the file system rather than the database
  - That is store a file name (and path) in the database and store the blob in the named file.
  - "The simple answer is: BLOBs smaller than 256KB are more efficiently handled by a database, while a filesystem is more efficient for those greater than 1MB. Of course, this will vary between different databases and filesystems." (Sears et al, https://arxiv.org/ftp/cs/papers/0701/0701168.pdf)
- CLOBs are probably better in DB -- max CLOB size in DB is 2TB.

One-to-One Relationship

- one-to-one relationship between a person's first name and last name
- one-to-one often should just be in a single table.
  - are there any tables in the univ database that have a one-one relation??

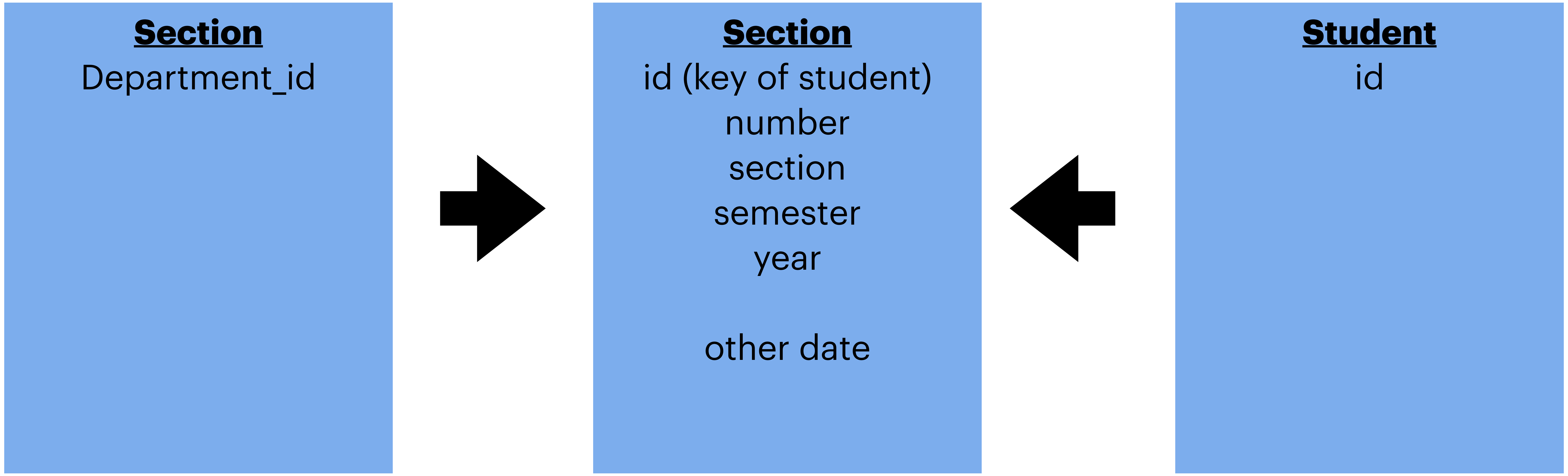
One-to-Many Relationship

- one-to-many relationship between an *course and a department*
- *A course is associated with a single department*
  - *a department may have several courses.*



Many-to-Many Relationship

- An section is associated with many students (possibly 0) via takes
- A student is associated with several (possibly 0) sections via takes



Choice of Primary key for Binary Relationships

- Many-to-Many relationships.   The union of the primary keys is a minimal superkey and is chosen as the primary key.
  - in the takes table the primary key is all of: id, num, section, semester, year
- One-to-Many relationships . The primary key of the “Many” side is a minimal superkey and is used as the primary key.
- One-to-one relationships. The primary key of either one of the participating entity sets forms a minimal superkey, and either one can be chosen as the primary key.

Revisiting The NuclearFamily and people tables

- Problems!!!

- Solutions??

```
create table nf10 (
    family_id INT GENERATED ALWAYS AS IDENTITY,
    description varchar,
    primary key (family_id)
);
```

```
create table people10 (
    person_id INT GENERATED ALWAYS AS IDENTITY,
    family_id int,
    name varchar,
    reltype varchar,
    primary key (person_id),
    foreign key (family_id) references nf10(family_id),
    check (reltype in ('parent', 'child'))
);
```

```
insert into nf10(description) values('small family');
insert into nf10(description) values('medium family');
insert into nf10(description) values('big family');
insert into people10(family_id, name, reltype) values(1, 'a', 'parent');
insert into people10(family_id, name, reltype) values(1, 'b', 'child');
insert into people10(family_id, name, reltype) values(2, 'c', 'parent');
insert into people10(family_id, name, reltype) values(2, 'd', 'child');
```

```
 family_id | parents | children
-----------+---------+-----------
         1 | {a}     | {b}
         2 | {c,e}   | {d,f}
         3 | {g,i}   | {h,k,m,o}
(3 rows)
```

```
create table nf10 (
    family_id INT GENERATED ALWAYS AS IDENTITY,
    description varchar,
    primary key (family_id)
);
```

```
create table people10 (
    person_id INT GENERATED ALWAYS AS IDENTITY,
    family_id int,
    name varchar,
    reltype varchar,
    primary key (person_id),
    foreign key (family_id) references nf10(family_id),
    check (reltype in ('parent', 'child'))
);
```

```
insert into nf10(description) values('small family');
insert into nf10(description) values('medium family');
insert into nf10(description) values('big family');
insert into people10(family_id, name, reltype) values(1, 'a', 'parent');
insert into people10(family_id, name, reltype) values(1, 'b', 'child');
insert into people10(family_id, name, reltype) values(2, 'c', 'parent');
insert into people10(family_id, name, reltype) values(2, 'd', 'child');
```

More family Views

```
create or replace view familyname10 as
  with child as (select nf10.family_id, array_agg(people10.name) as children
                 from nf10
                      join people10 on nf10.family_id=people10.family_id
                 where people10.reltype='child'
                 group by nf10.family_id),
       parent as (select nf10.family_id, array_agg(people10.name) as parents
                  from nf10
                       join people10 on nf10.family_id=people10.famil
                  where people10.reltype='parent'
                  group by nf10.family_id)
select child.family_id, parents, children
from child
     join parent on child.family_id=parent.family_id;
```

```
family=# select * from familyname10;
 family_id | parents | children
-----------+---------+-----------
         1 | {a}     | {b}
         2 | {c,e}   | {d,f}
         3 | {g,i}   | {h,k,m,o}
(3 rows)
```

Views are NOT tables … especially views that merge 2 or more tables

```
family=# update familyname4 set p1='aaa' where id=1;
ERROR:  cannot update view "familyname4"
DETAIL:  Views that do not select from a single table or view are not automatically updatable.
HINT:  To enable updating the view, provide an INSTEAD OF UPDATE trigger or an unconditional ON UPDATE DO INSTEAD rule.
Time: 0.629 ms
```

That View was UGLY

- Did almost the same query twice to get parents and children
- Do better (using functions with loops and if ... then) !!!
  - Usually if you are writing a loop, you are wrong

```
create or replace function seltt(arr1 varchar[], arr2 varchar[], filt varchar)
returns varchar[]
as $$
declare
    rtn varchar[];
begin
    for i in array_lower(arr1, 1)..array_upper(arr1, 1) loop
        if arr2[i]=filt then rtn := rtn || arr1[i]; end if;
    end loop;
    return rtn;
end;
$$
language plpgsql;


select family_id,
       seltt(array_agg(people10.name), array_agg(people10.reltype), 'parent') as parents,
       seltt(array_agg(people10.name), array_agg(people10.reltype), 'child') as children
from people10
group by family_id;
```

PostgreSQL only. The array type, and these array functions are Postgres specific.

materialized views

- Essentially a snapshot of a database.
- Views are generally used when data is to be accessed infrequently and data in table get updated on frequent basis. On other hand Materialized Views are used when data is to be accessed frequently and data in table not get updated on frequent basis.

```
create materialized view mfn4(id, p1, p2, c1,c2) as (select family_id, p4a.name,
p4b.name, p4c.name, p4d.name from nuclearfamily4 left outer join people4 as p4a
on parent1=p4a.person_id left outer join people4 as p4b on parent2=p4b.person_id
left outer join people4 as p4c on child1=p4c.person_id left outer join people4
as p4d on child2=p4d.person_id);

family=# update mfn4 set p1='c' where id=1;
ERROR:  cannot change materialized view "mfn4"

family=# refresh materialized view mfn4;
REFRESH MATERIALIZED VIEW
```