CS2102 Part 1

L1: Introduction

- A data model is a collection of concepts for describing data
- A schema is a description of the structure of a database using a data model
- A schema instance is the content of the database at a particular time
- Degree/Arity = Number of columns (attributes)
- Cardinality = Number of rows (tuple/records)
- Each instance of schema R is a relation which is a subset of $\{ (a1,a2,\cdots,an) \mid ai \in domain(Ai) \cup \{null\} \}$
- A relational database schema consists of a set of relation schemas

```
Students (
  studentId: integer,
  name: text,
  birthDate: date,
```

```
  cap: numeric
)

Courses (
  courseId: integer,
  name: text,
  credits: integer
)

Enrolls (
  sid: integer,
  cid: integer,
  grade: numeric
)

// each one is called relational schema,
// altogether they are called database schema
```

- A relational database is a collection of tables

Students

studentId	*name*	*birthDate*	*cap*
3118	Alice	1999-12-25	3.8
1423	Bob	2000-05-27	4.3
5609	Carol	1999-06-11	4.0

Courses

courseId	*name*	*credits*
101	Programming in C	5
112	Discrete Mathematics	4
204	Analysis of Algorithms	4
311	Database Systems	5

Enrolls

sid	*cid*	*grade*
3118	101	5.0
3118	112	4.0
3118	204	3.0
1423	112	4.5
. . . . .	. . . . .	. . . . .

- Relational database schema = relational schemas + data constraints
- Integrity constraint: a condition that restricts the data that can be stored in database instance
  - domain constraints restrict attribute values of relations
  - key constraints
  - foreign key constraints
  - other general constraints

Key Constraints

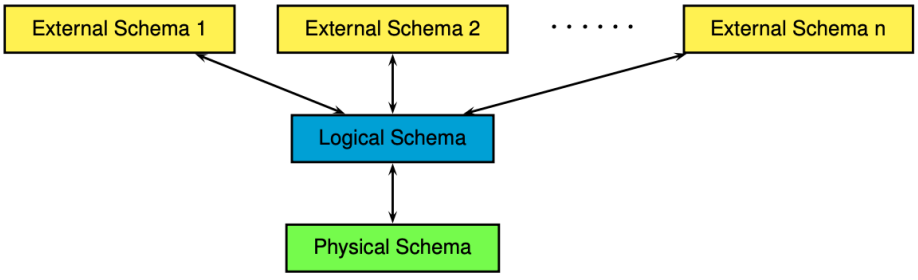
- A superkey is a subset of attributes in a relation that uniquely identifies its tuples

- No two distinct tuples of a relation have the same values in all attributes of superkey

- A key is a superkey that satisfies the additional property:
  - No *proper subset* of the key is a superkey
  - a key is a minimal subset of attributes in a relation that uniquely identifies its tuples
- Key attribute values cannot be *null*
- Candidate keys: all possible keys
- Primary key: selected candidate key
- Foreign key: primary key of a 2nd relation
  - foreign key constraints (referential integrity constraints): each foreign key value in referencing relation must either

    (1) appear as primary key value in referenced relation or

    (2) be a null value or
  - foreign key references can be duplicates in the referred table
  - the referenced column(s) must be a **primary key of the referenced table** or are **declared to be unique**



- Data in DBMS is described at three levels of abstractions
- **Logical Schema** - logical structure of data in DBMS
- **Physical Schema** - how the data described by logical schema is physically organized in DBMS
- **External Schema** - A customized view of logical schema for a group of users or an individual user

```
// Logical Database Schema:
Students (studentId, sname, birthDate, cap)
Profs (profId, pname, email, office)
Courses (courseId, cname, credits, profId, lectureTime)
Enrolls (sid, cid, grade)

// External Schema for Alice:
CourseEnrollment (cname, pname, lectureTime, totalEnrollment)

// External Schema for Bob:
StudentInfo (studentId, sname)
CourseInfo (courseId, cname, credits, profId, lectureTime)
EnrollInfo (sid, cid, cname, pname, lectureTime)
```

Data Independence

- Data independence is achieved via the three levels of abstraction
- **Physical data independence** protection from changes in physical schema
- **Logical data independence** protection from changes in logical schema

Transactions

- Abstraction for representing a logical unit of work
- **ACID Properties**
  - Atomicity: Either all the effects of a transaction are reflected in the database or none are
  - Consistency: The execution of a transaction in isolation preserves the consistency of the database
  - Isolation: The execution of a transaction is isolated from the effects of other concurrent transaction executions
  - Durability: The effects of a committed transaction persists in the database even in the presence of system failures

```
// e.g. Transfer(X, Y, amount)
fromBal := read balance from X’s account
if fromBal ≥ amount then
    toBal := read balance from Y’s account
    update Y’s balance to toBal + amount
```

```
    update X's balance to fromBal - amount
end if
```

L2: Relational Algebra

- Result of a comparison operation involving *null* value is *unknown*
- Result of an arithmetic operation involving *null* value is *null*

Unary Operators

Selection: $\sigma_c(R)$

$\sigma_c(R)$ selects tuples from relation *R* that satisfy the *selection condition c*

- For each $t \in R, t \in \sigma_c(R)$ iff *c* evaluates to **true** on *t*
- The output table has the same schema as *R*

Sells

rname	pizza	price
Corleone Corner	Diavola	24
Corleone Corner	Hawaiian	25
Corleone Corner	Margherita	19
Gambino Oven	Siciliana	16
Lorenzo Tavern	Funghi	23
Mamma's Place	Marinara	22
Pizza King	Diavola	17
Pizza King	Hawaiian	21

$\sigma_{price<20}$(Sells)

rname	pizza	price
Corleone Corner	Margherita	19
Gambino Oven	Siciliana	16
Pizza King	Diavola	17

Selection condition is a boolean combination of *terms*
A term is one of the following forms:

1. attribute **op** constant
2. attribute_1 **op** attribute_2
3. term_1 **and** term_2
4. term_1 **or** term_2
5. **not** term_1
6. (term_1)

op $\in \{=, <>, <, \leq, >, \geq\}$

Operator precedence: (), *op*, not, and, or

Projection: $\pi_l(R)$

$\pi_l(R)$ projects attributes given by a list l of attributes from relation *R*

- Each attribute in l must be an attribute in *R*
- The schema of the output table is determined by l
- remove duplicate

Sells

rname	pizza	price
Corleone Corner	Diavola	24
Corleone Corner	Hawaiian	25
Corleone Corner	Margherita	19
Gambino Oven	Siciliana	16
Lorenzo Tavern	Funghi	23
Mamma's Place	Marinara	22
Pizza King	Diavola	17
Pizza King	Hawaiian	21

$\pi_{\text{rname,pizza}}(\text{Sells})$

rname	pizza
Corleone Corner	Diavola
Corleone Corner	Hawaiian
Corleone Corner	Margherita
Gambino Oven	Siciliana
Lorenzo Tavern	Funghi
Mamma's Place	Marinara
Pizza King	Diavola
Pizza King	Hawaiian

Renaming: $\rho_l(R)$

$\rho_l(R)$ renames the attributes in *R* based on a list of attribute renamings l (*a*1:*b*1, ⋯ , *an*:*bn*)

- Each attribute in *R* can be renamed at most once
- The order of the attribute renamings in l does not matter

Restaurants

rname	area
Corleone Corner	North
Gambino Oven	Central
Lorenzo Tavern	Central
Mamma's Place	South
Pizza King	East

$\rho_{\text{area:region,rname:sname}}(\text{Restaurants})$

sname	region
Corleone Corner	North
Gambino Oven	Central
Lorenzo Tavern	Central
Mamma's Place	South
Pizza King	East

Binary Operators

Union: *R* ∪ *S*

Intersection: *R* ∩ *S*

Set-difference: *R* − *S*

- Union (∪), intersection (∩), and set-difference (−) operators require input relations to be union compatible

Union Compatibility

Two relations are union compatible if

1. they have the same number of attributes, and
2. the corresponding attributes have the same domains

- Union compatible relations do not necessarily use the same attribute names

Restaurants

rname	area
Corleone Corner	North
Gambino Oven	Central
Lorenzo Tavern	Central
Mamma's Place	South
Pizza King	East

Customers

cname	area
Homer	West
Lisa	South
Maggie	East
Moe	Central
Ralph	Central
Willie	North

Customers ∪ Restaurants

cname	area
Homer	West
Lisa	South
Maggie	East
Moe	Central
Ralph	Central
Willie	North
Corleone Corner	North
Gambino Oven	Central
Lorenzo Tavern	Central
Mamma's Place	South
Pizza King	East

Restaurants ∪ Customers

rname	area
Homer	West
Lisa	South
Maggie	East
Moe	Central
Ralph	Central
Willie	North
Corleone Corner	North
Gambino Oven	Central
Lorenzo Tavern	Central
Mamma's Place	South
Pizza King	East

Cross-Product (aka Cartesian product)

- Consider relations $R(A, B, C)$ and $S(X , Y)$
- Cross-product: $R \times S$ outputs a relation with schema (A, B, C, X, Y): $R \times S = \{(a, b, c, x, y) \mid (a, b, c) \in R, (x, y) \in S\}$
- The number of records of the resulting relation may be **smaller/equal/greater** than the number of records of R or the number of records of S
  - if R has 0 records and S has more than 0 records. As a result, R X S will have 0 records, which is smaller than that of S.
  - i.e. empty x any = empty

Customers

cname	area
Homer	West
Lisa	South
Maggie	East
Moe	Central
Ralph	Central
Willie	North

Restaurants

rname	area
Corleone Corner	North
Gambino Oven	Central
Lorenzo Tavern	Central
Mamma's Place	South
Pizza King	East

$\pi_{cname}(\sigma_{area='Central'}(\text{Customers}))$

cname
Moe
Ralph

$\pi_{rname}(\sigma_{area='Central'}(\text{Restaurants}))$

rname
Gambino Oven
Lorenzo Tavern

$\pi_{cname}(\sigma_{area='Central'}(\text{Customers})) \times \pi_{rname}(\sigma_{area='Central'}(\text{Restaurants}))$

cname	rname
Moe	Gambino Oven
Moe	Lorenzo Tavern
Ralph	Gambino Oven
Ralph	Lorenzo Tavern

Join Operators

Inner Join (Join): $R \bowtie_c S$

$R \bowtie_c S = \sigma_c(R \times S)$

Likes

cname	pizza
Homer	Hawaiian
Homer	Margherita
Lisa	Funghi
Maggie	Funghi
Moe	Funghi
Moe	Sciliana
Ralph	Diavola

$\pi_{cname,cname2}(Likes \bowtie_c \rho_{cname:cname2,pizza:pizza2}(Likes))$
where $c = (pizza = pizza2)$ *and* $(cname < cname2)$

cname	cname2
Lisa	Maggie
Lisa	Moe
Maggie	Moe

Natural Join: $R \bowtie S$

$R \bowtie S = \pi_l(R \bowtie_c \rho_{a_1:b_1,\ldots,a_n:b_n}(S))$

- A = common attributes between R & S = { *a*1, *a*2, ⋯, *an*}
- *c* is "(*a*1 = *b*1) *and* ⋯ *and* (*an* = *bn*)"
- l is the list of attributes in *R* that are also in *A*, followed by the list of attributes in *R* that are not in *A*, and the list of attributes in *S* that are not in *A*

Restaurants

rname	area
Corleone Corner	North
Gambino Oven	Central
Lorenzo Tavern	Central
Mamma's Place	South
Pizza King	East

Sells

rname	pizza	price
Corleone Corner	Diavola	24
Corleone Corner	Hawaiian	25
Corleone Corner	Margherita	19
Gambino Oven	Siciliana	16
Lorenzo Tavern	Funghi	23
Mamma's Place	Marinara	22
Pizza King	Diavola	17
Pizza King	Hawaiian	21

Restaurants ⋈ Sells

rname	area	pizza	price
Corleone Corner	North	Diavola	24
Corleone Corner	North	Hawaiian	25
Corleone Corner	North	Margherita	19
Gambino Oven	Central	Siciliana	16
Lorenzo Tavern	Central	Funghi	23
Mamma's Place	South	Marinara	22
Pizza King	East	Diavola	17
Pizza King	East	Hawaiian	19

Outer Joins

- A dangling tuple is a tuple in a join operand that does not participate in the join operation
- To preserve dangling tuples in the join result, use **outer joins**

R

D	B	A
0	x	100
2	y	100
4	w	400
5	z	200

S

E	A2	D2	C
a	100	0	i
b	300	1	j
c	200	5	k

Inner join: $R \bowtie_{D=D2} S$

D	B	A	E	A2	D2	C
0	x	100	a	100	0	i
5	z	200	c	200	5	k

Left outer join: $R \rightarrow_{D=D2} S$

D	B	A	E	A2	D2	C
0	x	100	a	100	0	i
5	z	200	c	200	5	k
2	y	100	null	null	null	null
4	w	400	null	null	null	null

Right outer join: $R \leftarrow_{D=D2} S$

D	B	A	E	A2	D2	C
0	x	100	a	100	0	i
5	z	200	c	200	5	k
null	null	null	b	300	1	j

Full outer join: $R \leftrightarrow_{D=D2} S$

D	B	A	E	A2	D2	C
0	x	100	a	100	0	i
5	z	200	c	200	5	k
2	y	100	null	null	null	null
4	w	400	null	null	null	null
null	null	null	b	300	1	j

Customers

cname	area
Homer	West
Lisa	South
Maggie	East
Moe	Central
Ralph	Central
Willie	North

Likes

cname	pizza
Homer	Hawaiian
Homer	Margherita
Lisa	Funghi
Maggie	Funghi
Moe	Funghi
Moe	Sciliana
Ralph	Diavola

cname	pizza
Homer	Hawaiian
Homer	Margherita
Lisa	Funghi
Maggie	Funghi
Moe	Funghi
Moe	Sciliana
Ralph	Diavola
Willie	null

$$\pi_{cname,pizza}(Customers \rightarrow Likes)$$

natural left outer join

L3: SQL

Constraints

- IS NULL
  - "*x* IS NOT NULL" = "NOT (x IS NULL)"
- IS DISTINCT FROM:
  - is equivalent to "x <> y" if both x & y are non-null values
  - evaluates to *false* if both the values are *null*
  - evaluates to *true* if only one of the values is *null*
  - "x IS NOT DISTINCT FROM y" = "NOT (x IS DISTINCT FROM y)"
- A constraint is violated if it evaluates to *false*
  - primary key = unique not null
  - foreign key allows for null values
  - if foreign key values are partially null, will not be checked against the Rooms data -> allowed
- general constraints are enforced using triggers
  - SQL has create assertion statement to support, but most DBMSs do not have

```
create table Census (
  city       varchar(50),
  state      char(2),
  population integer,
  unique(city, state)
);

/*
The unique constraint is violated if there exists two records x,y ∈ Census where “(x.c
ity <> y.city) or (x.state <> y.state)” evaluates to false
*/

create table Rooms (
  level integer,
  number integer,
  capacity integer,
  primary key(level, number)
);

create table Events (
  eid    integer,
  ename  varchar(100),
  date   date,
  level  integer,
  number integer,
  primary key (eid),
  foreign key (level, number)
      references Rooms(level, number)
      // no need specify (level, number) if its the same as primary key
      match full // disallow null values
);

/****************** Ver 2 ******************/
create table Rooms (
  rid integer,
  level integer,
  number integer,
  capacity integer,
  primary key(rid),
  unique (level, number) // no longer the pri key
);

create table Events (
  eid    integer,
  ename  varchar(100),
  date   date,
  level  integer,
  number integer,
  primary key (eid),
  foreign key (level, number)
      references Rooms(level, number)
      // column names must be mentioned in this case
);
```

```
create table Enrolls (
    sid             integer references Students (studentId),
    cid             integer,
    grade           char(2),
    primary key     (sid, cid),
    foreign key     (cid) references Courses (courseId));
```

both red references syntax can be used to declare foreign keys. If the column name (optional) is not specified, it refers to the primary key.

```
create table Lectures (
    cname       char(5),
    pname       varchar(50)
                constraint lectures_pname check (pname is not null),
    day         smallint
                constraint lectures_day check (day in (1,2,3,4,5)),
    hour        smallint
                constraint lectures_hour check (hour >= 8 and hour <= 17),
    constraint lectures_pri_key primary key (cname,day,hour),
    constraint lectures_cand_key unique (pname,day,hour)
);
```

Read data

```
select * from Customers
```

Data Modifications: Insert

```
create table Students (
  studentId integer primary key,
  name varchar(100) not null,
  birthDate date,
  dept varchar(20),
);

insert into Students -- default insert full entry
values (12345, 'Alice', '1999-12-25', 'Math');
```

```
insert into Students (name, studentId) -- columns unspecified will be null
values ('Bob', 67890), ('Carol', 11122);
```

Data Modifications: Delete

```
-- remove all students
delete from Students;

-- remove all students from Math dept
delete from Students
where dept = 'Maths';
```

Data Modifications: Update

```
-- Add 2% interest to all accounts
update Accounts
set balance = balance * 1.02;

-- Add $500 to account 12345 & change name to 'Alice'
update Accounts
set balance = balance + 500, name = 'Alice'
where accountId = 12345;
```

Foreign Key Constraint Violations

- NO ACTION: rejects delete/update if it violates constraint (default option)
- RESTRICT: similar to *NO ACTION* except that constraint checking can't be deferred
- CASCADE: propagates delete/update to referencing tuples
- SET DEFAULT: updates foreign keys of referencing tuples to some default value
- SET NULL: updates foreign keys of referencing tuples to *null* value

foreign key (sid) **references** Students **on delete cascade**

Students

studentId	*name*	*birthDate*
1	Alice	1999-12-25
2	Bob	2000-05-27
3	Carol	1999-06-11

Enrolls

sid	*cid*	*grade*
1	204	3.0
1	101	5.0
2	204	3.0
3	101	4.0
3	112	4.0

```
delete from
Students
where studentId = 1;
```

Students

studentId	*name*	*birthDate*
2	Bob	2000-05-27
3	Carol	1999-06-11

Enrolls

sid	*cid*	*grade*
2	204	3.0
3	101	4.0
3	112	4.0

foreign key (sid) **references** Students **on update cascade**

Students

studentId	*name*	*birthDate*
1	Alice	1999-12-25
2	Bob	2000-05-27
3	Carol	1999-06-11

Enrolls

sid	*cid*	*grade*
1	204	3.0
1	101	5.0
2	204	3.0
3	101	4.0
3	112	4.0

```
update Students
set studentId = 4
where studentId = 1;
```

Students

studentId	*name*	*birthDate*
4	Alice	1999-12-25
2	Bob	2000-05-27
3	Carol	1999-06-11

Enrolls

sid	*cid*	*grade*
4	204	3.0
4	101	5.0
2	204	3.0
3	101	4.0
3	112	4.0

```
create table  Employees (
      eid           integer primary key,
      ename         varchar(100),
      managerId     integer default 1,
      foreign key   (managerId) references Employees
                    on delete set default
);
```

Employees

eid	*ename*	*managerId*
1	Alice	null
2	Bob	1
3	Carol	2
4	Dave	2

```
delete from
Employees
where eid = 2;
```

Employees

eId	*ename*	*managerId*
1	Alice	null
3	Carol	1
4	Dave	1

Transactions

- A **transaction** consists of one or more update/retrieval operations (i.e., SQL statements)

- Abstraction for representing a logical unit of work
- The **begin** command starts a new transaction
- Each transaction must end with either a **commit** or **rollback** command
- w/o `begin` and `commit` , some parts of the transactions may be successful while some parts are not → inconsistent results, does not guarantee atomicity

```
begin;
update Accounts
set balance = balance + 1000 where accountId = 2;

update Accounts
set balance = balance - 1000 where accountId = 1;
commit;
```

- Atomicity: Either all the effects of a transaction are reflected in the database or none are
- Consistency: The execution of a transaction in isolation preserves the consistency of the database
- Isolation: The execution of a transaction is isolated from the effects of other concurrent transaction executions
- Durability: The effects of a committed transaction persists in the database even in the presence of system failures
- By default, constraints are checked immediately at the end of SQL statement execution
    - A violation will cause the statement to be rollbacked

Deferrable Constraints

- unique, primary key, foreign key
- `deferrable initially deferred` : checking done after `commit`
- `deferrable initially immediate` : check immediately unless set constraints to be deferred

```
create table Employees (
  eid integer primary key,
  ename varchar(100),
  managerId integer,
```

```
  constraint employees_fkey foreign key (managerId) references Employees
      deferrable initially immediate
);

insert into Employees values (1, 'Alice', null), (2, 'Bob', 1), (3, 'Carol', 2);

begin;
set constraints employees_fkey deferred;
delete from Employees where eid = 2;
update Employees set managerId = 1 where eid = 3;
commit;
```

Modifying Schema

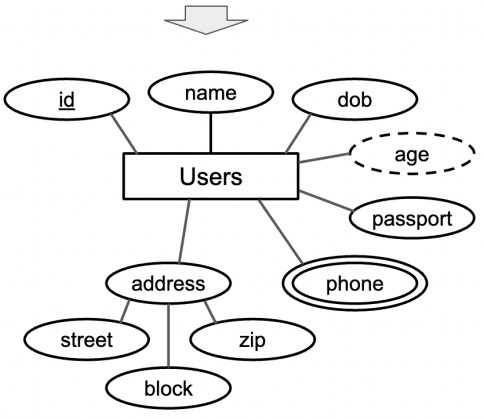
- Add/remove/modify columns
  - **alter table** Students **alter column** dept **drop default**;
  - **alter table** Students **drop column** dept;
  - **alter table** Students **add column** faculty **varchar**(20);
- Add/remove constraints
  - **alter table** Students **add constraint** fk_grade **foreign key** (grade) **references** Grades;

L4: Entity-Relationship (ER) Model

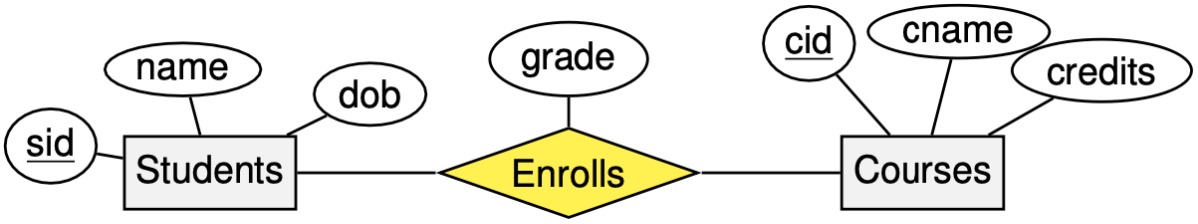
Attributes

- **Attribute**:
  - specific information describing an entity
  - represented by an oval in ER diagrams
- 4 subtypes of attributes
  - **Key attribute(s)**: uniquely identifies each entity (oval with the attribute name(s) underlines)
  - **Composite attribute**: composed of multiple other attributes (oval comprising of ovals)
  - **Multivalued attribute**: may consist of more than one value for a given entity (double-lined oval)
  - **Derived attribute**: derived from other attributes (dashed oval)

*For a valid booking, we need the **user's name**, **sex**, **address**, **phone number(s)**, and the **passport number**. Users are only able to pay via* credit card. *[...]*



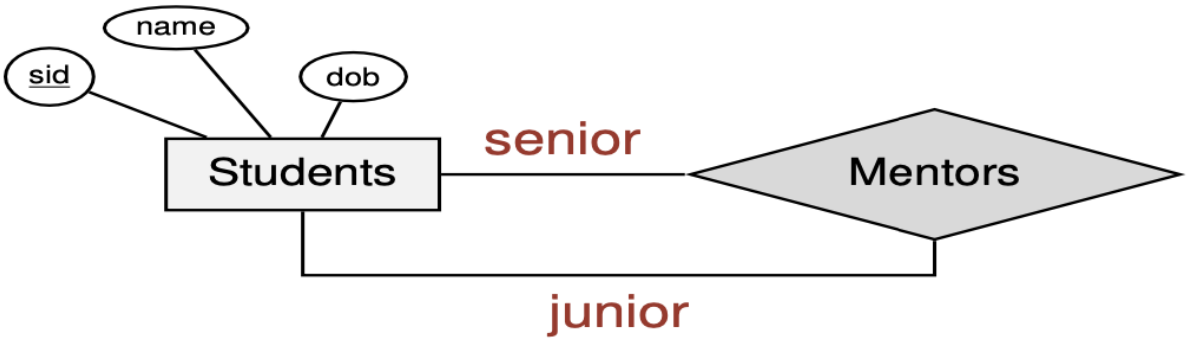
- **Relationship** is an association among two or more entities
- **Relationship set** is a collection of similar relationships
  - Relationship sets are represented by diamonds



Many-to-Many Relationship Sets

Relationship Roles

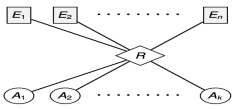
- The role is typically named the same as the entity set name & is not shown explicitly
- **Roles** are shown explicitly when one entity set appears two or more times in a relationship set



Degree of Relationship Sets

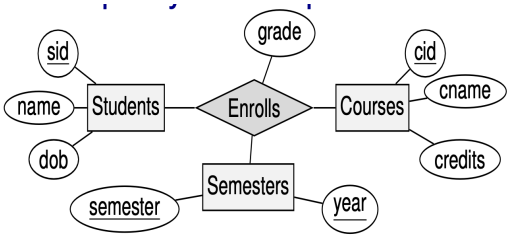
- An n-ary relationship set involves *n* entity roles
- n = **degree of relationship set**
  - n = 2: binary relationship set
  - n = 3: *ternary relationship set*

- Consider a n-ary relationship set R involving entity sets $E_1, \cdots, E_n$ with relationship attributes $\{A_1, \cdots, A_k\}$

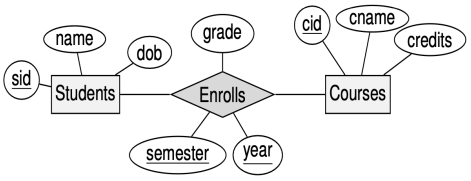


- Let $Key(E_i)$ denote the set of attributes that define the primary key of entity set E_i

- The key for R (denoted by Key(R)) is specified by some subset $A' \subseteq \{A_1, \cdots, A_k\}$ and some subset $E' \subseteq \{E_1, \cdots, E_n\}$ such that
  - $Key(R) = A' \cup \bigcup_{E_i \in E'} Key(E_i)$ is a minimal subset of attributes whose values uniquely identify a relationship instance of R
- Relationship attributes in $\{A_1, \cdots, A_k\}$ that form part of the relationship key are underlined.
  - Each attribute in A' is underlined in ER diagram

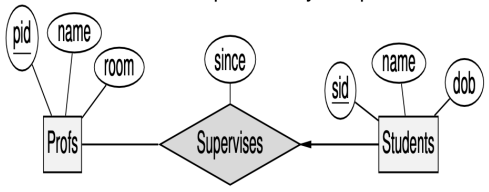


- Each instance of Enrolls has attributes $\{sid, cid, year, semester, grade\}$
- $A' = \emptyset$, $E' = \{\text{Students}, \text{Courses}, \text{Semesters}\}$
- Key(Enrolls) = $\{sid, cid, year, semester\}$
- Each (sid,cid,year,semester) appears at most once in `Enrolls` relationship set



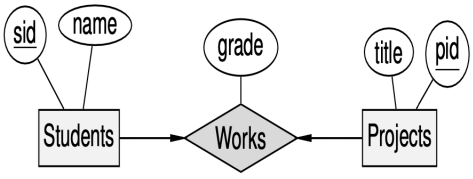
- Each instance of Enrolls has attributes $\{sid, cid, year, semester, grade\}$
- $A' = \{\text{year, semester}\}$, $E' = \{\text{Students}, \text{Courses}\}$
- Key(Enrolls) = $\{sid, cid, year, semester\}$
- Each (sid,cid,year,semester) appears at most once in `Enrolls` relationship set

- Each student can be supervised by at most one professor
- **one-to-many** relationship from Profs to Students / **many-to-one** relationship from Students to Profs
  - Each professor can supervise many students
  - Each student can be supervised by one professor

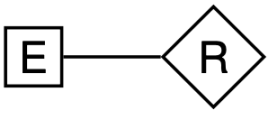


- Key(Supervises) = {sid}

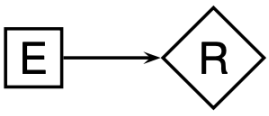
- **One-to-one** relationship between `Students` and `Projects`
- Each student can work on at most one project
- Each project can be worked on by at most one student



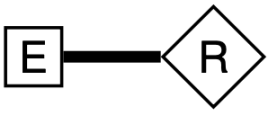
- Key(Works) = {sid} or {pid}



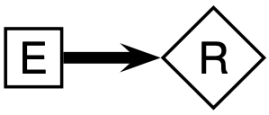
Each instance of E participates in 0 or more instances of R



Each instance of E participates in at most one instance of R



Each instance of E participates in at least one instance of R



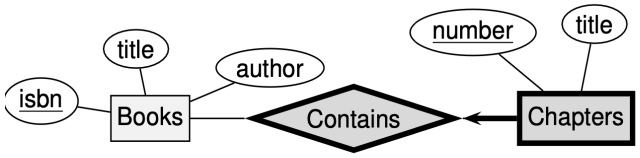
Each instance of E participates in exactly one instance of R

E is a weak entity set with identifying owner E' & identifying relationship set R

1. partial participation constraint;
2. key constraint on E wrt R;
3. total participation constraint on E wrt R;
4. one-one relationship

Weak-Entity Set

- A *weak entity* can only be uniquely identified by considering the primary key of another entity (called **owner entity**)
  - There must be a *many-to-one relationship* (called **identifying relationship**) from the weak entity set to an owner entity set
  - Weak entity set must have *total participation* in identifying relationship
- **Partial key** of a weak entity set is a set of attributes of weak entity set that uniquely identifies a weak entity for a given owner entity
- A weak entity's existence is dependent on the existence of its owner entity: owner deleted → weak entities deleted
- Weak entity set & its identifying relationship set are represented by a single relation

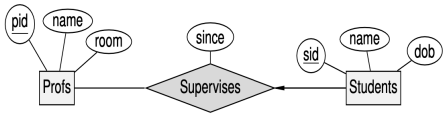



```
create table Books (
        isbn      char(30),
        title     char(50),
        author    char(60),
        primary key (isbn)
);
```

```
create table Chapters (
        number    integer,
        title     char(50),
        isbn      char(30),
        primary key (isbn, number),
        foreign key  (isbn) references Books
                     on delete cascade
);
```

Relationship Sets with Key Constraints

1. Represent relationships using separate tables (Key)



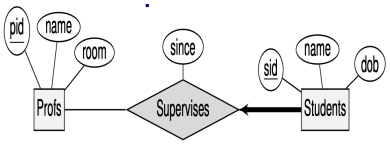
First approach: Represent `Supervises` using a separate table

```
create table Profs (
        pid      char(7),
        name     char(30),
        room     char(6),
        primary key (pid)
);
```

```
create table Students (
        sid      char(20),
        name     char(30),
        dob      date,
        primary key (sid)
);
```

```
create table Supervises (
        sid      char(20),
        pid      char(7) not null,
        since    date,
        primary key (sid),
        foreign key  (sid)
          references Students,
        foreign key  (pid)
          references Profs
);
```

equivalent



First approach: Represent `Supervises` using a separate table

```
create table Profs (
        pid      char(7),
        name     char(30),
        room     char(6),
        primary key (pid)
);
```

```
create table Students (
        sid      char(20),
        name     char(30),
        dob      date,
        primary key (sid)
);
```

```
create table Supervises (
        sid      char(20),
        pid      char(7) not null,
        since    date,
        primary key (sid),
        foreign key  (sid)
          references Students,
        foreign key  (pid)
          references Profs
);
```

not good: Total participation constraint of Students w.r.t. Supervises is not enforced by database schema

2. Combine relationships with one of the entities (Total part + Key)

```
create table Profs (
        pid      char(7),
        name     char(30),
        room     char(6),
        primary key (pid)
);
```

```
create table Students (
        sid      char(20),
        name     char(30),
        dob      date,
        pid      char(7),
        since    date,
        primary key (sid),
        foreign key  (pid)  references Profs
);
```

equivalent

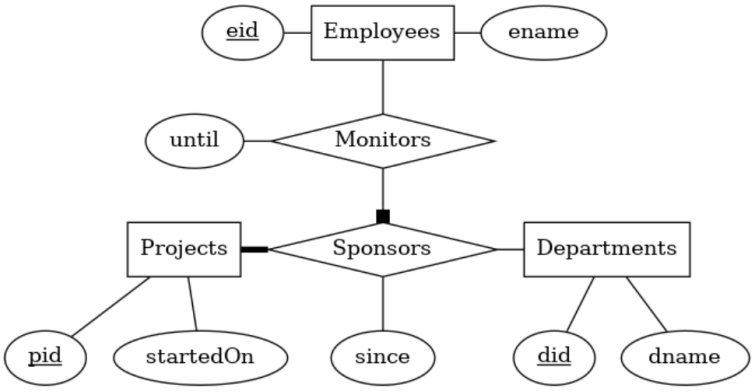
```
create table Profs (
        pid      char(7),
        name     char(30),
        room     char(6),
        primary key (pid)
);
```

```
create table Students (
        sid      char(20),
        name     char(30),
        dob      date,
        pid      char(7) not null,
        since    date,
        primary key (sid),
        foreign key  (pid)  references Profs
);
```

better

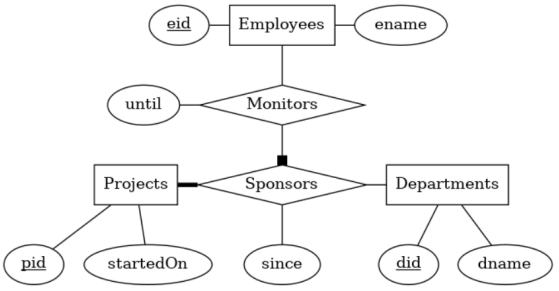
Aggregation

- treat a relationship as an entity so that it can participate in other relationship sets



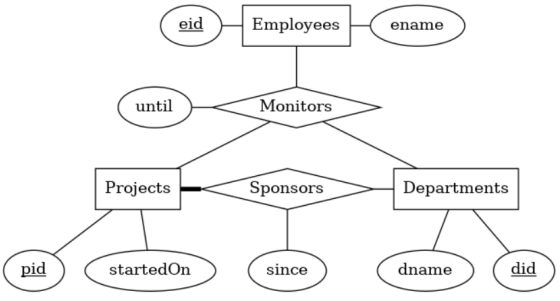

```
create table Sponsors (
    pid     char(30)
            references Projects,
    did     char(30),
            references Departments,
    since   date,
    primary key (pid,did)
);
```

```
create table Monitors (
    eid     char(20) references Employees,
    pid     char(30),
    did     char(30),
    until   date,
    primary key (eid,pid,did),
    foreign key (pid,did)
            references Sponsors (pid,did)
);
```




```
create table Monitors (
    eid     char(20) references Employees,
    pid     char(30),
    did     char(30),
    until   date,
    primary key (eid,pid,did),
    foreign key (pid,did)
            references Sponsors (pid,did)
);
```

aggregation → more accurate

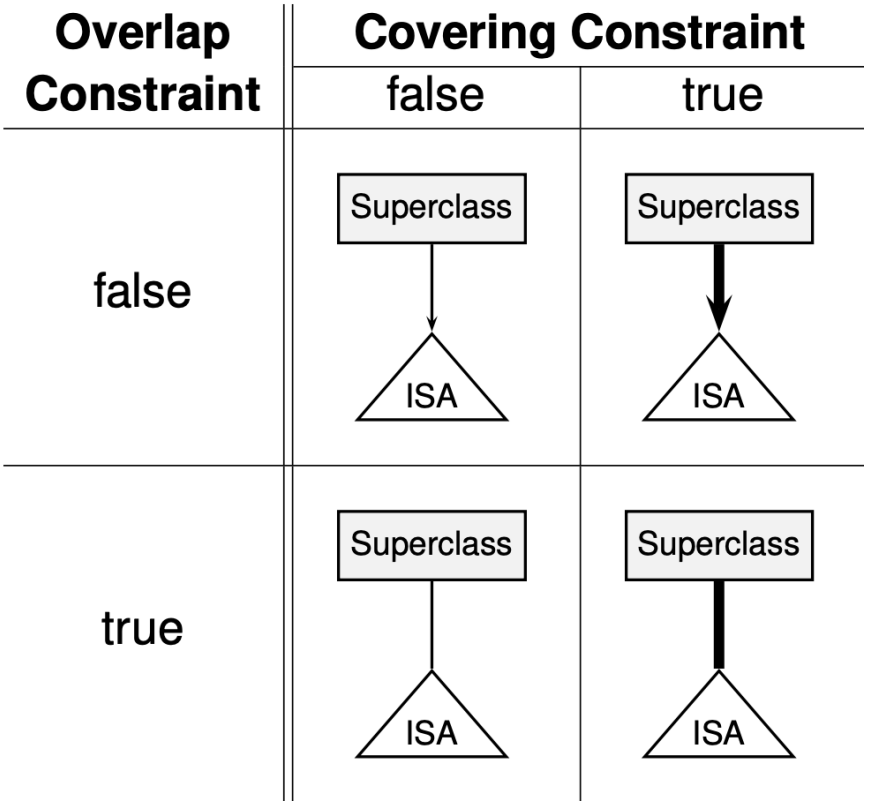


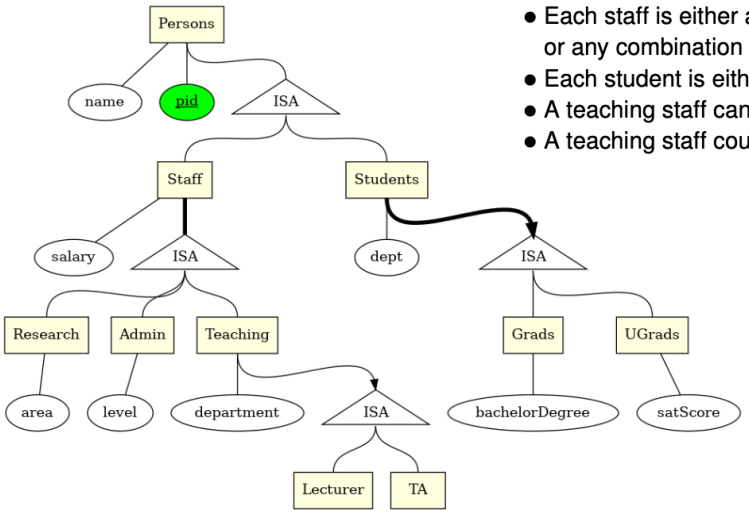

```
create table Monitors (
    eid     char(20) references Employees,
    pid     char(30), references Projects,
    did     char(30) references Departments,
    until   date,
    primary key (eid,pid,did),
);
```

many-many: more general

ISA Hierarchies

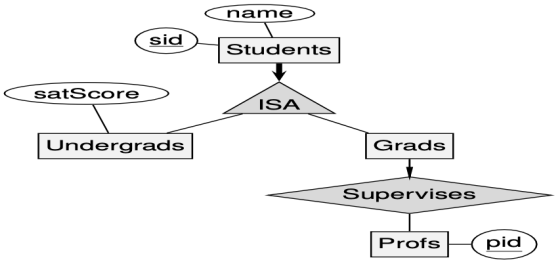
- Every entity in a subclass entity set is an entity in its superclass entity set
- Overlap constraint: A ISA hierarchy satisfies the overlap constraint if an entity in a superclass could belong to multiple subclasses
  - Satisfies: Undirected edge from superclass to ISA triangle (*)
  - Does not satisfy: Directed edge from superclass to ISA triangle(0..1)
- Covering constraint: A ISA hierarchy satisfies the covering constraint if every entity in a superclass has to belong to some subclass
  - Satisfies: thick edge from superclass to ISA triangle (1..)
  - Does not satisfy: thin edge from superclass to ISA triangle (0..)





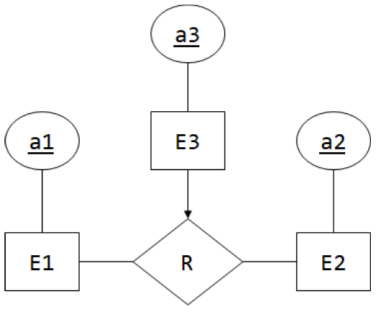
- A person could be both a student & staff
- A person could be neither a student nor staff
- Each staff is either a research, teaching, admin, or any combination of these three roles
- Each student is either an undegraduate or a graduate student
- A teaching staff can't be both a lecturer and TA
- A teaching staff could be neither a lecturer nor a TA

Simplest approach: One relation per subclass/superclass




```
create table Students (
  sid      char(20)
           primary key,
  name     char(30));

create table Undergrads (
  sid          char(20) primary key references Students
               on delete cascade,
  satScore     numeric);

create table Grads (
  sid          char(20) primary key references Students
               on delete cascade,
  supervisor   char(7) references Profs(pid));
```

Consider the ER diagram shown below, with the following properties:



- **There are 20 entities in E1**
- **There are 5 entities in E2**
- **There are 10 entities in E3**

Minimum = 0

- No entity-set has total participation constraints

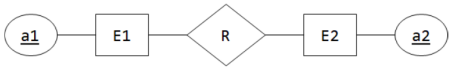
- At minimum, no entity is required to participate in R

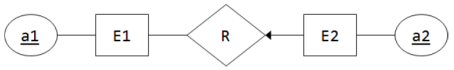
Maximum = 10

- E3 has key constraint
- E3 can participate in R at most 1 time
- 10 entities in E3 → max is achieved when all E3 has participated

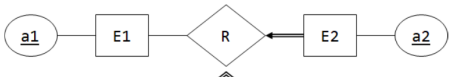
```
CREATE TABLE E1 (
  a1  integer  PRIMARY KEY,
);

CREATE TABLE E2 (
  a1  integer REFERENCES E1 (a1) ON DELETE cascade,
  a2  integer,
  PRIMARY KEY (a1, a2)
);
```

Which of the following ER diagram has its constraints captured by the relational schema above?

A.


B.


C.


D.


E.


Answer: C,E

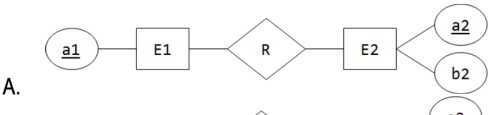
- Reasoning for C
  - Consider table E2 to mean combined relation R and entity E2
  - PK (a_1, a_2) ensures that
    - Every a_2 is in R
      - total participation constraint
    - There can be (a_1^1, a_2) and (a_1^2, a_2)
      - no key constraint
    - Every unique a_2 can uniquely identify an entity E2
      - by virtue of not having any other attributes
- Reasoning for E
  - By translation of weak entity-set to SQL from lecture note

```
CREATE TABLE E1 (
  a1  integer  PRIMARY KEY,
);

CREATE TABLE E2 (
  a1  integer REFERENCES E1 (a1) ON DELETE cascade,
  a2  integer,
  b2  integer,
```

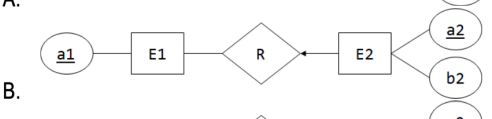
```
  PRIMARY KEY (a1, a2)
);
```

Which of the following ER diagram has its constraints captured by the relational schema above?

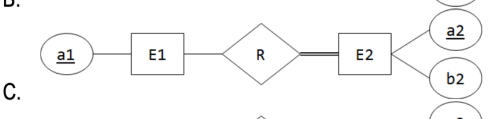
A.



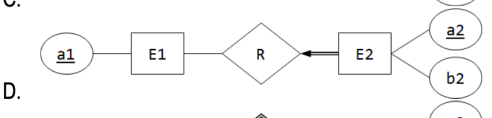
B.



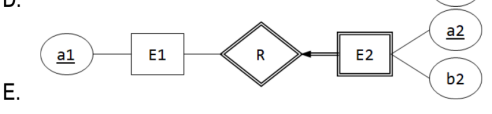
C.



D.



E.



Answer: E

- Reasoning for NOT C
  - Consider table E2 to mean combined relation R and entity E2
  - PK (a_1, a_2) ensures that
    - Every a_2 is in R
      - total participation constraint
    - There can be (a_1^1, a_2) and (a_1^2, a_2)
      - no key constraint
    - a_2 cannot uniquely identify entity in E2
      - we can have (a_1^1, a_2, b_2^1) and (a_1^2, a_2, b_1^2) and thus violate PK of E2
- Reasoning for E
  - By translation of weak entity-set to SQL from lecture note

C's ER diagram shows that b2 is determined by a2, but from the schema, b2 is determined by (a1, a2)

Which of the following statements are TRUE? Select all that applies.

1. Any primary key constraints can always be enforced by column constraints
   a. counter: `PRIMARY KEY (a, b)`
2. Any foreign key constraints can always be enforced by table constraints
3. Any not null constraints can always be enforced by table constraints
4. Any unique constraints can always be enforced by column constraints
   a. counter `UNIQUE (a, b)`
5. All the statements above are FALSE

Consider an application about a fictional world of RPG. You are given the following constraints:

C1. Every adventurer (identified by aid) must either be a fighter, a mage, or a healer.

C2. Every adventurer must belong to only one of the three classes (fighter, mage, or healer).

C3. Every adventurer must be registered to exactly one guild (identified by gid)

C4. Each guild must have at least one adventurer but may have more than one adventurer.

C5. Each party must have exactly one fighter, one mage, and one healer.

```
CREATE TABLE Adventurer (
  aid integer PRIMARY KEY,
  type varchar(10) NOT NULL,
  CHECK (type = ‘fighter’ OR type = ‘mage’ OR type = ‘healer’)
);

CREATE TABLE Guild (
  gid  integer PRIMARY KEY,
  aid  integer NOT NULL REFERENCES Adventurer (aid)
);

CREATE TABLE Party (
  fighter integer REFERENCES Adventurer (aid),
  mage integer REFERENCES Adventurer (aid),
  healer integer REFERENCES Adventurer (aid),
  PRIMARY KEY (fighter, mage, healer)
);
/*
C3 not enforced: gid is not part of Advernturer table
C4 not enforced: gid is a PK, cannot have (gid1, aid1), (gid1, aid2)
C5: not checked the 3 adventurers correspond to the three types specifically
*/
```

Lec 5: SQL 2

```
select [distinct] slist -- projection
from f-list -- default cross product
[where qualification] -- selection

-- Two tuples (a1, c1) and (a2, c2) are considered to be distinct if the following eva
luates to true:
-- (a1 IS DISTINCT FROM a2) or (c1 IS DISTINCT FROM c2)

select item as product, price * qty as cost -- as for renaming
from Orders;
```

```
select ’Price of ’ || pizza || ’ is ’ || round(price / 1.3) || ’ USD’ as menu
from Sells
where rname = ’Corleone Corner’;
-- || is the string concatenation operator
```

```
round(value, 2) -- round to 2dp
round(value, -3) -- round to nearest 1000
```

Sells

rname	pizza	price
Corleone Corner	Diavola	24
Corleone Corner	Hawaiian	25
Corleone Corner	Margherita	19
Gambino Oven	Siciliana	16
Lorenzo Tavern	Funghi	23
Mamma's Place	Marinara	22
Pizza King	Diavola	17
Pizza King	Hawaiian	21

menu
Price of Diavola is 11 USD
Price of Hawaiian is 20 USD
Price of Margherita is 15 USD

Set Operators

- eliminate duplicates
- intersect has higher precedence
- Q1 union Q2 = $Q_1 \cup Q_2$
- Q1 intersect Q2 = $Q1 \cap Q_2$
- Q1 except Q2 = $Q1 - Q2$
- union all, intersect all, except all preserves duplicate records

Multi-relation Queries

```
-- Find distinct pairs of customers and restaurants that are located in the same area
-- equivalent queries
select cname, rname
from   Customers as C, Restaurants as R -- as is optional
where  C.area = R.area;

select cname, rname
from   Customers C join Restaurants R -- or inner join
       on C.area = R.area

select cname, rname
from Customers natural join Restaurants

-- Find customers and the pizzas they like; include also customers who don't like any
pizza
select C.cname, L.pizza
```

```
from   Customers C left join Likes L -- or left outer join
       on C.cname, L.cname;

select cname, pizzafrom Customers natural left join Likes; -- or natural left outer jo
in

-- Find all customer-pizza pairs (C,P) where the pizza is sold by some restaurant that
is located in the same area as that of the customer. Include customers whose associate
d set of pizzas is empty
select cname, pizza
from   Customers natural left join
       (Restaurants natural join Sells)
```

Subquery Expressions

- `EXISTS`
  - Returns *true* if the the output of the subquery is non-empty; otherwise, *false*

```
-- Find distinct customers who like some pizza sold by “Corleone Corner”
select distinct cname
from   Likes L
where  [not] exists(
       select 1 -- as long as this subquery is not empty, exists returns true, so
use 1 for simplicity
       from Sells S
       where S.rname = 'Corleone Corner'
       and S.pizza = L.pizza
       );
```

- `IN` (subset of `ANY`)
  - subquery must return exactly one column
  - Returns *false* if the output of the subquery is empty; otherwise return the result of the boolean expression $((v = v_1) or (v = v_2) or ... or (v = v_n))$ where *v* denote the result of expression, {v1, v2, ..., vn} denote the output of the subquery

```
-- Find distinct customers who like some pizza sold by “Corleone Corner”
select distinct cname
from   Likes
where  pizza in (
       select pizza
```

```
       from Sells
       where rname = 'Corleone Corner'
       );

-- Find pizzas that contain ham or seafood
select distinct pizza from Contains
where ingredient in ('ham', 'seafood');

select distinct pizza from Contains
where ingredient = 'ham' or ingredient = 'seafood'

select pizza from Contains where ingredient = 'ham'
union
select pizza from Contains where ingredient ='seafood'
/***********************************/
/*
SELECT * FROM runners;
+----+--------------+
| id | name         |
+----+--------------+
|  1 | John Doe     |
|  2 | Jane Doe     |
|  3 | Alice Jones  |
|  4 | Bobby Louis  |
|  5 | Lisa Romero  |
+----+--------------+

sql> SELECT * FROM races;
+----+----------------+-----------+
| id | event          | winner_id |
+----+----------------+-----------+
|  1 | 100 meter dash |  2        |
|  2 | 500 meter dash |  3        |
|  3 | cross-country  |  2        |
|  4 | triathalon     |  NULL     |
+----+----------------+-----------+
*/
SELECT *
FROM runners
WHERE id NOT IN (SELECT winner_id FROM races)
-- returns empty set
-- If the set being evaluated by the SQL NOT IN condition contains any values that
are null, then the outer query here will return an empty set

SELECT *
FROM runners
WHERE id NOT IN (
  SELECT winner_id
  FROM races
  WHERE winner_id IS NOT null
) -- gives the expected behaviour
```

- `ANY` / `SOME`
  - Subquery must return exactly one column
  - Returns *false* if the output of the subquery is empty;
    otherwise return the result of the boolean expression
    $((v\ op\ v_1)\ or\ (v\ op\ v_1)\ or\ ...\ or\ (v\ op\ v_n))$
    - op: =, <>, <, >, ≤, ≥

```
-- Find distinct restaurants that sell some pizza P1 that is more expensive than s
ome pizza P2 sold by “Corleone Corner”. P1 and P2 are not necessarily the same piz
za. Exclude “Corleone Corner” from the query result.
select distinct rname
from   Sells
where rname <> 'Corleone Corner'
and   price > any (
      select price
      from Sells
      where rname = 'Corleone Corner'
      );
```

- `ALL`
  - subquery must return exactly one column
  - Returns *true* if the output of the subquery is empty;
    otherwise return $((v\ op\ v_1)\ and\ (v\ op\ v_1)\ and\ ...\ and\ (v\ op\ v_n))$

```
-- For each restaurant, find the name and price of its most expensive pizzas. Excl
ude restaurants that do not sell any pizza. Assume that all prices are non-null va
lues.
select rname, pizza, price
from   Sells S1
where  price >= all (
       select S2.price
       from Sells S2
       where S2.rname = S1.rname
       );

SELECT continent, name, area
FROM world x
WHERE area >= ALL
    (SELECT area FROM world y
        WHERE y.continent=x.continent
          AND area is not null) -- area is not null cannot be left out
```

Row Constructors

- return subqueries with >1 columns

```
-- Find information on lectures that are scheduled after Wednesday 11am.
select *
from Lectures
where day > 3 or ((day = 3) and (hour > 11));

select *
from Lectures
where row(day, hour) > row(3, 11);

-- For each lecturer, find the time of his/her earliest lecture during the week.
select pname, day, hour
from   Lectures L
where  row(day, hour) <= all (
       select day, hour
       from Lectures L2
       where L2.pname = L.pname
       );
```

Scalar Subqueries

- a subquery that returns at most one tuple with one column
- If the subquery's output is empty, its return value is null

```
-- For each restaurant that sells Funghi, find its name, area, and selling price.
select rname,
       (select R.area from Restaurants R where R.rname = S.rname), price
from Sells S
where pizza = 'Funghi';
```

Usage of Subqueries

```
-- Find distinct pizzas that are sold by restaurants located in the east area.
select distinct pizza
from   Sells natural join
       (
          select rname
          from Restaurants
          where area = 'East'
       ) as East_Restaurants(rname)l
-- subqueries in FROM clause must be enclosed in () and assigned a table alias

select cname, pname, str as day, hour
from   Lectures natural join
       (
```

```
          values (1, 'Mon'), (2, 'Tue'), (3, 'Wed'), (4, 'Thu'), (5, 'Fri'), (6, 'Sa
t'), (7, 'Sun')
       ) as day_str (day, str);
```

Lectures

cname	pname	day	hour
CS101	Alice	1	10
CS123	Alice	3	10
CS200	Bob	4	8
MA300	Bob	3	14

cname	pname	day	hour
CS101	Alice	Mon	10
CS123	Alice	Wed	10
CS200	Bob	Thu	8
MA300	Bob	Wed	14

ORDER BY

- A query that is not a group-by query can’t have any aggregate function in the order-by clause

```
-- For each restaurant that sells some pizza, find its name, area, and the pizzas it s
ells together with their prices. Show the output in ascending order of the area, follo
wed by in descending order of the price.
select *
from Restaurants, Sells
where Restaurants.rname = Sells.rname
order by area asc, price desc;

select *
from Restaurants, Sells
where Restaurants.rname = Sells.rname
order by area, price desc; -- default asc

select a -- not in group
from r
order by avg(c) --invalid

select sum(b * y)
from r, s
order by avg(c); -- valid: single group group-by
```

LIMIT

- limit the number of tuples

OFFSET

- removes the specified number of tuples

```
-- Assume that Sells.price is unique. Find the 4th and 5th most expensive pizzas. Show
the pizza name, the name of the restaurant that sells it, and its selling price for ea
ch output tuple; and show the output in descending order of price.
select pizza, rname, price
from Sells
order by price desc
offset 3
limit 2;
```

L6: SQL 3

Aggregate Functions

- computes a single value from a set of tuples
- empty/null set ⇒ result = null (count = 0)

Query	Meaning
`select min(A) from R`	min NON-NULL value in A
`select max(A) from R`	max NON-NULL value in A
`select avg(A) from R`	avg NON-NULL value in A
`select sum(A) from R`	sum of NON-NULL values in A
`select count(A) from R`	count number of NON-NULL values in A
`select count(*) from R`	count number of rows (incl those with NULL) in R
`select avg(distinct(A)) from R`	avg distinct NON-NULL values in A
`select sum(distinct(A)) from R`	sum of distinct NON-NULL values in A
`select count(distinct(A)) from R`	count number of distinct NON-NULL values in A

```
-- Find the most expensive pizzas and the restaurants that sell them (at the most expe
nsive price)
select pizza, rname
from Sells
where price = (select max(price) from Sells);
```

GROUP BY

- `GROUP BY a1, a2, ..., an` two tuples t & t' belong to the same group if the following expression evaluates to true: `t.a1 IS NOT DISTINCT FROM t'.a1) AND ... AND (t.an IS NOT DISTINCT FROM t'.an)`

- For each column *A* in relation *R* that appears in the `SELECT` clause, one of the following conditions must hold:
  1. A appears in `GROUP BY` clause
  2. A appears in an aggregated expression in the `SELECT` clause (e.g. `min(A)`), or
  3. the primary key of R appears in the `GROUP BY` clause (i.e. GROUP BY primary key)
- A query that is not a group-by query can't have any aggregate function in the order-by clause

```
-- For each restaurant that sells some pizza, find the minimum and maximum prices of i
ts pizzas
-- 1. partition tuples in Sells into groups based on rname
-- 2. compute min and max price for each group
-- 3. output one tuple for each gorup
select rname, min(price), max(price)
from Sells
group by rname;

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

select distinct rname
from Sells;

-- is equivalent to

select rname
from Sells
group by rname

select a
from r
order by avg(c) --invalid

select sum(b * y)
from r, s
order by avg(c); -- valid: single group group-by
```

HAVING

- For each column A in relation R that appears in the `HAVING` clause, one of the following conditions must hold:
  1. A appears in the `GROUP BY` clause,
  2. A appears in an aggregated expression in the `HAVING` clause, or

3. the primary key of R appears in the `GROUP BY` clause

```
-- Find restaurants located in the ‘East’ area that sell pizzas with an average sellin
g price higher than the minimum selling price at Pizza King
select rname
from   Sells
where  rname in (
       select rname
       from Restaurants
       where area = 'East'
      )
group by rname
having avg(price) > (
       select min(price)
       from Sells
       where rname = 'Pizza King');
```

Q1:
```
select      rname, avg(price)
from        (Restaurants natural join Sells) RS
group by    rname
having      avg(price) > (
            select  avg(price)
            from    Sells natural join Restaurants
            where   area = RS.area
);
```

Q2:
```
select      rname, avg(price)
from        Restaurants R natural join Sells
group by    R.rname
having      avg(price) > (
            select  avg(price)
            from    Sells natural join Restaurants
            where   area = R.area
);
```

Q3:
```
select      rname, avg(price)
from        (Sells natural join Restaurants) SR
group by    rname
having      avg(price) > (
            select  avg(price)
            from    Sells natural join Restaurants
            where   area = SR.area
);
```

Q4:
```
select      rname, avg(price)
from        Sells S natural join Restaurants R
group by    S.rname
having      avg(price) > (
            select  avg(price)
            from    Sells natural join Restaurants
            where   area = R.area
);
```

1, 2 valid; 3, 4 invalid: rname is not primary key of Sells

```
-- valid
select rname, min(price) as min_price
from Sells
group by rname
having avg(price) > 30
order by sum(price);

-- valid
select *
from Sells
where price = (select max(price) from Sells);

-- invalid
select *
```

```
from Sells
where price = max(price) -- cannot eval max(price)
```

Common Table Expressions (CTEs)

```
-- syntax
with
    R1 as (Q1),
    R2 as (Q2),
    ...,
    Rn as (Qn)
select/insert/update/delete statement S;

-- example
with rname_avgprice as (
  select rname, avg(price) as avg_price
  from Sells
  group by rname
),

area_avgprice as (
  select area, avg(price) as avg_price
  from Sells natural join Restaurants
  group by area
)

select rname
from rname_avgprice R
where avg_price > (
  select avg_price
  from area_avgprice
  where area = (select area from Restaurants where rname = R.rname)
);
```

Views

- a virtual relation that can be used for querying

```
Courses (courseId*, cname, credits, profId, lecTime, quota)
Profs (profId*, pname, officeRoom, contactNum)
Students (studentId, sname, email, birthDate)
Enrollment (courseid, numUG, numPG, numExchange, numAudit)

create view CourseInfo as
  select cname, pname, lecTime,
  numUG+numPG+numExchange+numAudit as numEnrolled
  from Courses natural join Profs natural join Enrollment;
```

Conditional Expressions

CASE

```
select name, case
  when marks >= 70, then 'A'
  when marks >= 60, then 'B'
  when marks >= 50, then 'C'
  else 'D'
end as grade
from Scores

select name , case
  when (first = 'pass') or (second = 'pass') or (third = 'pass') then 'pass'
  else 'fail'
end as result
from Tests;

/* swap id */
/*
Input:
Seat table:
+----+---------+
| id | student |
+----+---------+
| 1  | Abbot   |
| 2  | Doris   |
| 3  | Emerson |
| 4  | Green   |
| 5  | Jeames  |
+----+---------+
Output:
+----+---------+
| id | student |
+----+---------+
| 1  | Doris   |
| 2  | Abbot   |
| 3  | Green   |
| 4  | Emerson |
| 5  | Jeames  |
+----+---------+
*/
select (case
        when (id  % 2 = 1 and id < (select count(*) from seat))  then id + 1
        when (id  % 2 = 1 and id = (select count(*) from seat)) then id
        else id - 1
        end) as id, student
from seat
order by id;
```

COALESCE

- returns the first non-null value in its arguments

- returns null if all arguments are null
- `coalesce(col, [value])` returns `value` if `col` is null

```
select name, coalesce(third, second, first) as result
from Tests;
-- eqv to the 2nd query in CASE
```

NULLIF

- `nullif(v1, v2)` returns null if v1 == v2, else return v1

Tests

name	result
Alice	absent
Bob	fail
Carol	pass
Dave	absent
Eve	pass

name	status
Alice	null
Bob	fail
Carol	pass
Dave	null
Eve	pass

select name, **nullif**(result,'absent') as status
from Tests;

LIKE

```
-- Find customer names ending with “e” that consists of at least four characters
select cname
from Customers
where cname like '_ _ _%e' -- no need spacing, its there for to differentiate the spac
es for visualisation

-- _ matches any char
-- % matches any seq of 0 or more char

SELECT name
FROM countries
WHERE name LIKE 'Y%' /* finds the country that start with Y */
'%Y' -- ends with Y
'%x%' -- contains x
'%land' -- ends with land
'%a%a%a%' -- cointains 3 or more a
```

```
'_t%' -- has t as the 2nd letter
'%o__o%' -- two 'o' char separated by two other char e.g. Lesotho and Moldova

SELECT name
from world
where capital = concat(name, ' City') -- returns countries where the capital is the co
untry plus "City".

/* Find the capital and the name where the capital includes the name of the country.
*/
select capital, name
from world
where capital like concat('%', name, '%');

/* Find the capital and the name where the capital is an extension of name of the coun
try. e.g. Mexico City */
select capital, name
from world
where capital like concat(name, '%_')
```

Queries with Universal Quantification

- e.g. “all” ⇒ use negation

```
Courses(couseId*, name, dept)
Students(studentId, name, birthDate)
Enrolls(sid, cid, grade)

with courses_unenrolled as (
      select courseId
      from   Courses C
      where  dept = 'CS'
      and not exists (
             select 1
             from Enrolls E
             where E.cid = C.courseId
             and E.sid = x
      )
select name
from Students S
where not exists courses_unenrolled;
```

exclusive or

```
/* Show the countries that are big by area (more than 3 million) or big by population
(more than 250 million) but not both. Show name, population and area. */
select name, population, area
from world
```

```
where population > 250000000 or area > 3000000
except
select name, population, area
from world
where population > 250000000 and area > 3000000
```

order by multiple columns

```
-- order by multiple columns
SELECT id,
  first_name,
  last_name,
  salary
FROM employee
ORDER BY salary DESC, last_name;
-- decreasing salary followed by increasing last name

-- Show the 1984 winners and subject ordered by subject and winner name; but list Chem
istry and Physics last.
SELECT winner, subject
from nobel
where yr = 1984
order by
case when subject in ('Chemistry', 'Physics') then 1 else 0 end, subject, winner
```

2nd highest

```
SELECT MAX(Salary) AS SecondHighestSalary
FROM Employee
WHERE Salary < (SELECT MAX(Salary) FROM Employee)
```

update gender

```
update Salary
set sex = case sex
    when 'm' then 'f'
    else 'm'
    end;

update salary
set sex = if (sex='m', 'f', 'm');
```

rank w/o rank function

```
/*
- The scores should be ranked from the highest to the lowest.
- If there is a tie between two scores, both should have the same ranking.
- After a tie, the next ranking number should be the next consecutive integer value. I
n other words, there should be no holes between ranks.

Input:
Scores table:
+----+-------+
| id | score |
+----+-------+
| 1  | 3.50  |
| 2  | 3.65  |
| 3  | 4.00  |
| 4  | 3.85  |
| 5  | 4.00  |
| 6  | 3.65  |
+----+-------+
Output:
+-------+------+
| score | rank |
+-------+------+
| 4.00  | 1    |
| 4.00  | 1    |
| 3.85  | 2    |
| 3.65  | 3    |
| 3.65  | 3    |
| 3.50  | 4    |
+-------+------+
*/
select score, (
    select count(distinct(score))
    from scores
    WHERE score >= s.score
) as 'rank'
from scores s
order by score desc;

/* select employees w/ top 3 salary in the dept
Input:
Employee table:
+----+-------+--------+--------------+
| id | name  | salary | departmentId |
+----+-------+--------+--------------+
| 1  | Joe   | 85000  | 1            |
| 2  | Henry | 80000  | 2            |
| 3  | Sam   | 60000  | 2            |
| 4  | Max   | 90000  | 1            |
| 5  | Janet | 69000  | 1            |
| 6  | Randy | 85000  | 1            |
| 7  | Will  | 70000  | 1            |
+----+-------+--------+--------------+
Department table:
+----+-------+
| id | name  |
```

```
+----+-------+
| 1  | IT    |
| 2  | Sales |
+----+-------+
Output:
+------------+----------+--------+
| Department | Employee | Salary |
+------------+----------+--------+
| IT         | Max      | 90000  |
| IT         | Joe      | 85000  |
| IT         | Randy    | 85000  |
| IT         | Will     | 70000  |
| Sales      | Henry    | 80000  |
| Sales      | Sam      | 60000  |
+------------+----------+--------+
*/
select D.name as 'Department', E.name as 'Employee', Salary
from department D join employee E on E.departmentId = D.id
where 3 >= (
    select count(distinct(salary))
    from employee
    where departmentId = D.id
    and salary >= E.salary
);
```

consecutive numbers

```
/*
find all numbers that appear at least three times consecutively.
Input:
Logs table:
+----+-----+
| id | num |
+----+-----+
| 1  | 1   |
| 2  | 1   |
| 3  | 1   |
| 4  | 2   |
| 5  | 1   |
| 6  | 2   |
| 7  | 2   |
+----+-----+
Output:
+-----------------+
| ConsecutiveNums |
+-----------------+
| 1               |
+-----------------+
Explanation: 1 is the only number that appears consecutively for at least three times.
*/
select distinct num as ConsecutiveNums
from logs
where (id+1, num) in (select * from logs)
```

```
and (id+2, num) in (select * from logs);

/*
display the records with three or more rows with consecutive id's, and the number of p
eople is greater than or equal to 100 for each.

Return the result table ordered by visit_date in ascending order.

Input:
Stadium table:
+------+------------+-----------+
| id   | visit_date | people    |
+------+------------+-----------+
| 1    | 2017-01-01 | 10        |
| 2    | 2017-01-02 | 109       |
| 3    | 2017-01-03 | 150       |
| 4    | 2017-01-04 | 99        |
| 5    | 2017-01-05 | 145       |
| 6    | 2017-01-06 | 1455      |
| 7    | 2017-01-07 | 199       |
| 8    | 2017-01-09 | 188       |
+------+------------+-----------+
Output:
+------+------------+-----------+
| id   | visit_date | people    |
+------+------------+-----------+
| 5    | 2017-01-05 | 145       |
| 6    | 2017-01-06 | 1455      |
| 7    | 2017-01-07 | 199       |
| 8    | 2017-01-09 | 188       |
+------+------------+-----------+
Explanation:
The four rows with ids 5, 6, 7, and 8 have consecutive ids and each of them has >= 100
people attended. Note that row 8 was included even though the visit_date was not the n
ext day after row 7.
*/
with GT100 as (
    select id, visit_date, people
    from stadium
    where people >= 100
)

select distinct id, visit_date, people
from GT100
where ((id+1) in (select id from GT100) and (id+2) in (select id from GT100))
or ((id-1) in (select id from GT100) and (id-2) in (select id from GT100))
or ((id+1) in (select id from GT100) and (id-1) in (select id from GT100))

# select distinct G1.id, G1.visit_date, G1.people
# from GT100 G1, GT100 G2, GT100 G3
# where (G1.id + 1 = G2.id and G1.id + 2 = G3.id)
# or (G1.id - 1 = G2.id and G1.id + 1 = G3.id)
# or (G1.id - 1 = G2.id and G1.id - 2 = G3.id)
# order by G1.visit_date
;
```