Applications' View of a Relational Database Management System (RDBMS): Why use it?

- Persistent data structure
  - Large volume of data
- High-level API for reading (querying) & writing (inserting, deleting, updating)
  - Automatically optimized
- Transaction management (ACID)
  - Atomicity: all or none happens, despite failures & errors
  - Consistency
  - Isolation: appearance of "one at a time"
  - Durability: recovery from failures and other errors

Data Structure: Relational Model

- Relational Databases: Schema + Data
- Schema:
  - collection of tables (also called relations)
  - each table has a set of attributes
  - no repeating relation names, no repeating attributes in one table
- Data (also called instance):
  - set of tuples
  - tuples have one value for each attribute
Data Structure: Primary Keys; Foreign Keys are value-based pointers

- "ID is primary key of Schedule" => its value is unique in Schedule.ID
- "Schedule.Movie is foreign key (referring) to Movie.ID" means every Movie value of Schedule also appears as Movie.ID
- Intuitively, Schedule.Movie operates as pointer to Movie(s)

Schema design has its own intricacies

- This is a bad schema design!
- Problems
  - Change the name of a theater
  - Change the name of a movie's director
  - What about theaters that play no movie?

How to Design a Database and Avoid Bad Decisions

- With experience...
- Learn in CSE132A normalization rules of database design
  - a well-developed mathematical theory about how to fix step by step a "bad" schema
- CSE135: Think entities and relationships – then translate to relations
  - You will need to provide both E/R diagrams and tables for Assignment 2
Data Structure: Relational Model

Example Problem:
- Represent the students and Spring classes of the CSE department, including the enrollment of students in classes.
- Students have pid, first name and last name.
- Classes have a name, a number, date code (TR, MW, MWF) and start/end time.
- A student enrolls for a number of credits in a class.

Solution:...

Example 1: E/R-Based Design

E/R → Relational Schema: Basic Translation

- For every entity
  - create corresponding table
  - For each attribute of the entity, add a corresponding attribute in the table
  - Include an ID attribute in the table even if not in E/R
- For every relationship
  - create corresponding table
  - For each attribute of the relationship, add a corresponding attribute in the table
  - For each referenced entity $E_i$ include in the table a required foreign key attribute referencing ID of $E_i$
Sample relational database, per previous page's algorithm

<table>
<thead>
<tr>
<th>id</th>
<th>pid</th>
<th>first_name</th>
<th>last_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8888888</td>
<td>John</td>
<td>Smith</td>
</tr>
<tr>
<td>2</td>
<td>1111111</td>
<td>Mary</td>
<td>Doe</td>
</tr>
<tr>
<td>3</td>
<td>2222222</td>
<td>null</td>
<td>Chen</td>
</tr>
</tbody>
</table>

Enrollment

<table>
<thead>
<tr>
<th>id</th>
<th>class</th>
<th>student</th>
<th>credits</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

Students

<table>
<thead>
<tr>
<th>id</th>
<th>pid</th>
<th>first_name</th>
<th>last_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8888888</td>
<td>John</td>
<td>Smith</td>
</tr>
<tr>
<td>2</td>
<td>1111111</td>
<td>Mary</td>
<td>Doe</td>
</tr>
<tr>
<td>3</td>
<td>2222222</td>
<td>null</td>
<td>Chen</td>
</tr>
</tbody>
</table>

### Declaration of schemas in SQL's Data Definition Language

```sql
CREATE TABLE classes {
    ID SERIAL PRIMARY KEY,
    name TEXT,
    number TEXT,
    date_code TEXT,
    start_time TIME,
    end_time TIME
}
```

- If we had "ID INTEGER PRIMARY KEY" we would be responsible for coming up with ID values. SERIAL leads to a counter that automatically provides ID values upon insertion of new tuples.
- Changed name from "end" to "end_time" since "end" is reserved keyword.

```sql
CREATE TABLE students {
    ID SERIAL PRIMARY KEY,
    pid TEXT,
    first_name TEXT,
    last_name TEXT
}
```

- Foreign key declaration: Every value of enrollment.class must also appear as classes.ID.
- Declaration of "required" constraint: enrollment.student cannot be null (notice, it would make no sense to have an enrollment tuple without a student involved).

```sql
CREATE TABLE enrollment (
    ID SERIAL,
    class INTEGER REFERENCES classes (ID) NOT NULL,
    student INTEGER REFERENCES students (ID) NOT NULL,
    credits INTEGER
}
```

- For managing and accessing the Postgresql server, use the pgAdmin graphical client:
  - Right click on Postgresql, and select Connect
  - Right click on Databases, and select New Database
  - Enter a new name for the database, and click Okay
  - Highlight the database, and select Tools -> Query Tool
  - Write SQL code (or open the examples), and select Query -> Execute

... some easy hands-on experience

- Install the Postgresql open source database
- For educational and management purposes use the pgAdmin client to define schemas, insert data:
  - See online instructions
- For managing and accessing the Postgresql server, use the pgAdmin graphical client:
  - Right click on Postgresql, and select Connect
  - Right click on Databases, and select New Database
  - Enter a new name for the database, and click Okay
  - Highlight the database, and select Tools -> Query Tool
  - Write SQL code (or open the examples), and select Query -> Execute
Creating a schema and inserting some data

- Open file enrollment.sql
- Copy and paste its CREATE TABLE and INSERT commands in the Query Tool
- Run it – you now have the sample database!
- Run the first 3 SELECT commands to see the data you have in the database

Example 2a

Movies have a title, a year of release and length (in minutes).
Actors have names and address.
Actors appear in movies.
A movie is (co-)owned by studios.
Studios have a name and address.

```
CREATE TABLE movies (
  id SERIAL PRIMARY KEY,
  title TEXT,
  year INTEGER,
  length INTEGER,
)
CREATE TABLE stars (
  id SERIAL PRIMARY KEY,
  name TEXT,
  address TEXT
)
CREATE TABLE studios (
  id SERIAL PRIMARY KEY,
  name TEXT,
  address TEXT
)
CREATE TABLE starsin (
  id SERIAL,
  movie INTEGER REFERENCES movies (id) NOT NULL,
  star INTEGER REFERENCES stars (id) NOT NULL
)
CREATE TABLE ownership (
  id SERIAL,
  movie INTEGER REFERENCES movies (id) NOT NULL,
  owner INTEGER REFERENCES studios (id) NOT NULL
)
```
Example 2b: many-to-one relationship

Modification to Example 2a:
A movie is owned by at most one studio.

E/R⇒ Relational: Basic Translation revisited for many-to-one relationship

- For every entity, do the usual...
- For every many-to-many relationship, do the usual...
- For every 2-way many-to-one relationship, where
  - \( E_m \) is the "many" side
  - \( E_o \) is the "one" side (pointed by the arrow)
  - do not create table, instead:
  - In the table corresponding to \( E_m \) add a (non-required) foreign key attribute referencing the ID of the table corresponding to \( E_o \)

```
CREATE TABLE movies (  
    ID SERIAL PRIMARY KEY,  
    title TEXT,  
    year INTEGER,  
    length INTEGER,  
    owner INTEGER REFERENCES studios (ID)  
)  
CREATE TABLE stars (  
    ID SERIAL PRIMARY KEY,  
    name TEXT,  
    address TEXT  
)  
CREATE TABLE studios (  
    ID SERIAL PRIMARY KEY,  
    name TEXT,  
    address TEXT  
)  
CREATE TABLE starsin (  
    ID SERIAL,  
    movie INTEGER REFERENCES movies (ID) NOT NULL,  
    star INTEGER REFERENCES stars (ID) NOT NULL  
)```
Example 2c: Constraints: uniqueness; required attributes

In addition to Example 2b’s assumptions, let us also assume that:
- title, year, length, star name and studio name are required attributes of the respective entities
- default is that an attribute value may be null
- studios have unique names, i.e., no two studios may have the same name

Example 2d: Constraints: Required relationship; cardinality ranges

In addition to Example 2c’s assumptions, let us also assume that:
- a movie is owned by exactly one studio
- so far we had not assumed that the owning studio has to be known (not null)
- a movie must have at least one actor and no more than 100

SQL Schema for Examples 2c, 2d

```sql
CREATE TABLE movies (  ID SERIAL PRIMARY KEY,  title TEXT NOT NULL,  year INTEGER NOT NULL,  length INTEGER NOT NULL,  owner INTEGER REFERENCES studios (ID) NOT NULL )
CREATE TABLE stars (  ID SERIAL PRIMARY KEY,  name TEXT NOT NULL,  address TEXT )
CREATE TABLE studios (  ID SERIAL PRIMARY KEY,  name TEXT NOT NULL UNIQUE,  address TEXT )
CREATE TABLE starsin (  ID SERIAL,  movie INTEGER REFERENCES movies (ID) NOT NULL,  star INTEGER REFERENCES stars (ID) NOT NULL )
```
### A sample database

<table>
<thead>
<tr>
<th>stars</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>name</td>
<td>address</td>
</tr>
<tr>
<td>1</td>
<td>Al Pacino</td>
<td>New York, NY</td>
</tr>
<tr>
<td>2</td>
<td>Harrison Ford</td>
<td>Beverly Hills, CA</td>
</tr>
<tr>
<td>3</td>
<td>Tom Hanks</td>
<td>Santa Monica, CA</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>studios</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>name</td>
<td>address</td>
</tr>
<tr>
<td>1</td>
<td>20th Century Fox</td>
<td>Century City, CA</td>
</tr>
<tr>
<td>2</td>
<td>Paramount Pictures</td>
<td>Hollywood, CA</td>
</tr>
<tr>
<td>3</td>
<td>Universal Pictures</td>
<td>Universal City, CA</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>movie</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>title</td>
<td>year</td>
</tr>
<tr>
<td>1</td>
<td>Forreest Gump</td>
<td>1994</td>
</tr>
<tr>
<td>2</td>
<td>The Godfather</td>
<td>1972</td>
</tr>
<tr>
<td>3</td>
<td>Star Wars</td>
<td>1977</td>
</tr>
<tr>
<td>4</td>
<td>Scent of a Woman</td>
<td>1993</td>
</tr>
</tbody>
</table>

### Why do we want constraints? What happens when they are violated?

- Protect the database from erroneous data entry
- Prevent database states that are inconsistent with the rules of the business process you want to capture
- Whenever you attempt to change (insert, delete, update) the database in a way that violates a constraint the database will prevent the change
  - Try it out on the sample databases of the class page

### Some constraints are not implemented by some SQL database systems

- Consider the cardinality constraint that a movie has between 1 and 100 actors.
- The SQL standard provides a way, named CHECK constraints, to declare such
  - its specifics will make more sense once we have seen SQL queries
- However, no open source database implements the CHECK constraints.
- Project Phase II: Introduce such constraints on your E/R, despite the fact that you will not be able to translate them to the SQL schema
Vice versa: SQL allows some constraints that are not in plain E/R

Notable cases:
• Attribute value ranges
  – Example: Declare that the year of movies is after 1900
• Multi-attribute UNIQUE
  – Example: Declare that the (title, year) attribute value combination is unique

Include the above cases (if applicable) to your SQL schema of Project Phase II

---

Added constraints of previous slide to schema of Example 2d

```sql
CREATE TABLE movies {
    ID SERIAL PRIMARY KEY,
    title TEXT NOT NULL,
    year INTEGER NOT NULL CHECK (year > 1900),
    length INTEGER NOT NULL,
    owner INTEGER REFERENCES studios (ID) NOT NULL,
    UNIQUE (title, year)
}
CREATE TABLE stars {
    ID SERIAL PRIMARY KEY,
    name TEXT NOT NULL,
    address TEXT
}
CREATE TABLE studios {
    ID SERIAL PRIMARY KEY,
    name TEXT NOT NULL UNIQUE,
    address TEXT
}
CREATE TABLE starsin {
    ID SERIAL,
    movie INTEGER REFERENCES movies (ID) NOT NULL,
    star INTEGER REFERENCES stars (ID) NOT NULL
}
```

---

Example 3: one-to-one relationships

Assume that a president manages exactly one studio and a studio may have at most one president. Notice: a studio may not have a president but in order to be a president one has to manage a studio.
1st candidate

CREATE TABLE presidents {
    id SERIAL PRIMARY KEY,
    name TEXT,
    age INTEGER,
    manages INTEGER REFERENCES studios (ID) NOT NULL UNIQUE
}

CREATE TABLE studios {
    id SERIAL PRIMARY KEY,
    name TEXT,
    address TEXT,
}

2nd candidate

2nd candidate is not preferred. Why? What constraint it misses?

CREATE TABLE presidents {
    id SERIAL PRIMARY KEY,
    name TEXT,
    age INTEGER
}

CREATE TABLE studios {
    id SERIAL PRIMARY KEY,
    name TEXT,
    address TEXT,
    managedBy INTEGER REFERENCES presidents (id) UNIQUE
}

Example 4: 3-Way Relationship

- A studio has contracted with a particular star to act in a particular movie
  - No ownership of movies by studios
CREATE TABLE contract (
    ID SERIAL,
    movie INTEGER REFERENCES movies (ID) NOT NULL,
    star INTEGER REFERENCES stars (ID) NOT NULL,
    owner INTEGER REFERENCES studios (ID) NOT NULL,
    fee INTEGER
)

Example 5a : Self-Relationships with Roles

CREATE TABLE movies (ID SERIAL PRIMARY KEY,

CREATE TABLE sequelof (ID SERIAL,
    prequel INTEGER REFERENCES movies (ID) NOT NULL,
    sequel INTEGER REFERENCES movies (ID) NOT NULL
)

Notice the use of roles as attributes names for the foreign keys
Example 5b: Combo: One-to-one Self-Relationship

A movie has at most one direct "prequel" and at most one direct "sequel".

Example 6: Subclassing

Schemas for subclassing: Candidate 1

```
CREATE TABLE subject{
    id SERIAL PRIMARY KEY,
    --
}
CREATE TABLE undergrad{
    studentid INTEGER REFERENCES student(id) NOT NULL,
    minor INTEGER REFERENCES subject(id),
    --
}
CREATE TABLE graduate{
    studentid INTEGER REFERENCES student(id) NOT NULL,
    degree TEXT NOT NULL CHECK (degree="PhD" OR degree="MS"),
    advisor INTEGER REFERENCES faculty(id) NOT NULL,
    --
}
CREATE TABLE student{
    id SERIAL PRIMARY KEY,
    pid TEXT NOT NULL UNIQUE,
    name TEXT NOT NULL,
    major INTEGER REFERENCES subject(id),
    --
}
CREATE TABLE faculty{
    id SERIAL PRIMARY KEY,
    --
}
CREATE TABLE faculty{
    id SERIAL PRIMARY KEY,
    --
}
CREATE TABLE subject{
    id SERIAL PRIMARY KEY,
    --
}
CREATE TABLE undergrad{
    studentid INTEGER REFERENCES student(id) NOT NULL,
    minor INTEGER REFERENCES subject(id),
    --
}
CREATE TABLE graduate{
    studentid INTEGER REFERENCES student(id) NOT NULL,
    degree TEXT NOT NULL CHECK (degree="PhD" OR degree="MS"),
    advisor INTEGER REFERENCES faculty(id) NOT NULL,
    --
}
CREATE TABLE student{
    id SERIAL PRIMARY KEY,
    pid TEXT NOT NULL UNIQUE,
    name TEXT NOT NULL,
    major INTEGER REFERENCES subject(id),
    --
}
CREATE TABLE faculty{
    id SERIAL PRIMARY KEY,
    --
}
CREATE TABLE subject{
    id SERIAL PRIMARY KEY,
    --
}
CREATE TABLE undergrad{
    studentid INTEGER REFERENCES student(id) NOT NULL,
    minor INTEGER REFERENCES subject(id),
    --
}
CREATE TABLE graduate{
    studentid INTEGER REFERENCES student(id) NOT NULL,
    degree TEXT NOT NULL CHECK (degree="PhD" OR degree="MS"),
    advisor INTEGER REFERENCES faculty(id) NOT NULL,
    --
}
CREATE TABLE student{
    id SERIAL PRIMARY KEY,
    pid TEXT NOT NULL UNIQUE,
    name TEXT NOT NULL,
    major INTEGER REFERENCES subject(id),
    --
}
CREATE TABLE faculty{
    id SERIAL PRIMARY KEY,
    --
}
CREATE TABLE subject{
    id SERIAL PRIMARY KEY,
    --
}
CREATE TABLE undergrad{
    studentid INTEGER REFERENCES student(id) NOT NULL,
    minor INTEGER REFERENCES subject(id),
    --
}
CREATE TABLE graduate{
    studentid INTEGER REFERENCES student(id) NOT NULL,
    degree TEXT NOT NULL CHECK (degree="PhD" OR degree="MS"),
    advisor INTEGER REFERENCES faculty(id) NOT NULL,
    --
}
CREATE TABLE student{
    id SERIAL PRIMARY KEY,
    pid TEXT NOT NULL UNIQUE,
    name TEXT NOT NULL,
    major INTEGER REFERENCES subject(id),
    --
}
CREATE TABLE faculty{
    id SERIAL PRIMARY KEY,
    --
}
CREATE TABLE subject{
    id SERIAL PRIMARY KEY,
    --
}
CREATE TABLE undergrad{
    studentid INTEGER REFERENCES student(id) NOT NULL,
    minor INTEGER REFERENCES subject(id),
    --
}
CREATE TABLE graduate{
    studentid INTEGER REFERENCES student(id) NOT NULL,
    degree TEXT NOT NULL CHECK (degree="PhD" OR degree="MS"),
    advisor INTEGER REFERENCES faculty(id) NOT NULL,
    --
}
CREATE TABLE student{
    id SERIAL PRIMARY KEY,
    pid TEXT NOT NULL UNIQUE,
    name TEXT NOT NULL,
    major INTEGER REFERENCES subject(id),
    --
}
CREATE TABLE faculty{
    id SERIAL PRIMARY KEY,
    --
}
CREATE TABLE subject{
    id SERIAL PRIMARY KEY,
    --
}
CREATE TABLE undergrad{
    studentid INTEGER REFERENCES student(id) NOT NULL,
    minor INTEGER REFERENCES subject(id),
    --
}
CREATE TABLE graduate{
    studentid INTEGER REFERENCES student(id) NOT NULL,
    degree TEXT NOT NULL CHECK (degree="PhD" OR degree="MS"),
    advisor INTEGER REFERENCES faculty(id) NOT NULL,
    --
}
CREATE TABLE student{
    id SERIAL PRIMARY KEY,
    pid TEXT NOT NULL UNIQUE,
    name TEXT NOT NULL,
    major INTEGER REFERENCES subject(id),
    --
}
CREATE TABLE faculty{
    id SERIAL PRIMARY KEY,
    --
}
CREATE TABLE subject{
    id SERIAL PRIMARY KEY,
    --
}
CREATE TABLE undergrad{
    studentid INTEGER REFERENCES student(id) NOT NULL,
    minor INTEGER REFERENCES subject(id),
    --
}
CREATE TABLE graduate{
    studentid INTEGER REFERENCES student(id) NOT NULL,
    degree TEXT NOT NULL CHECK (degree="PhD" OR degree="MS"),
    advisor INTEGER REFERENCES faculty(id) NOT NULL,
    --
}
CREATE TABLE student{
    id SERIAL PRIMARY KEY,
    pid TEXT NOT NULL UNIQUE,
    name TEXT NOT NULL,
    major INTEGER REFERENCES subject(id),
    --
}
CREATE TABLE faculty{
    id SERIAL PRIMARY KEY,
    --
}
CREATE TABLE subject{
    id SERIAL PRIMARY KEY,
    --
}
CREATE TABLE undergrad{
    studentid INTEGER REFERENCES student(id) NOT NULL,
    minor INTEGER REFERENCES subject(id),
    --
}
CREATE TABLE graduate{
    studentid INTEGER REFERENCES student(id) NOT NULL,
    degree TEXT NOT NULL CHECK (degree="PhD" OR degree="MS"),
    advisor INTEGER REFERENCES faculty(id) NOT NULL,
    --
}
CREATE TABLE student{
    id SERIAL PRIMARY KEY,
    pid TEXT NOT NULL UNIQUE,
    name TEXT NOT NULL,
    major INTEGER REFERENCES subject(id),
    --
}
CREATE TABLE faculty{
    id SERIAL PRIMARY KEY,
    --
}
CREATE TABLE subject{
    id SERIAL PRIMARY KEY,
    --
}
CREATE TABLE undergrad{
    studentid INTEGER REFERENCES student(id) NOT NULL,
    minor INTEGER REFERENCES subject(id),
    --
}
CREATE TABLE graduate{
    studentid INTEGER REFERENCES student(id) NOT NULL,
    degree TEXT NOT NULL CHECK (degree="PhD" OR degree="MS"),
    advisor INTEGER REFERENCES faculty(id) NOT NULL,
    --
}
CREATE TABLE student{
    id SERIAL PRIMARY KEY,
    pid TEXT NOT NULL UNIQUE,
    name TEXT NOT NULL,
    major INTEGER REFERENCES subject(id),
    --
}
CREATE TABLE faculty{
    id SERIAL PRIMARY KEY,
    --
}
CREATE TABLE subject{
    id SERIAL PRIMARY KEY,
    --
}
CREATE TABLE undergrad{
    studentid INTEGER REFERENCES student(id) NOT NULL,
    minor INTEGER REFERENCES subject(id),
    --
}
CREATE TABLE graduate{
    studentid INTEGER REFERENCES student(id) NOT NULL,
    degree TEXT NOT NULL CHECK (degree="PhD" OR degree="MS"),
    advisor INTEGER REFERENCES faculty(id) NOT NULL,
    --
}
CREATE TABLE student{
    id SERIAL PRIMARY KEY,
    pid TEXT NOT NULL UNIQUE,
    name TEXT NOT NULL,
    major INTEGER REFERENCES subject(id),
    --
}
CREATE TABLE faculty{
    id SERIAL PRIMARY KEY,
    --
}
CREATE TABLE subject{
    id SERIAL PRIMARY KEY,
    --
}
CREATE TABLE undergrad{
    studentid INTEGER REFERENCES student(id) NOT NULL,
    minor INTEGER REFERENCES subject(id),
    --
}
CREATE TABLE graduate{
    studentid INTEGER REFERENCES student(id) NOT NULL,
    degree TEXT NOT NULL CHECK (degree="PhD" OR degree="MS"),
    advisor INTEGER REFERENCES faculty(id) NOT NULL,
    --
}
CREATE TABLE student{
    id SERIAL PRIMARY KEY,
    pid TEXT NOT NULL UNIQUE,
    name TEXT NOT NULL,
    major INTEGER REFERENCES subject(id),
    --
}
CREATE TABLE faculty{
    id SERIAL PRIMARY KEY,
    --
}
CREATE TABLE subject{
    id SERIAL PRIMARY KEY,
    --
}
CREATE TABLE undergrad{
    studentid INTEGER REFERENCES student(id) NOT NULL,
    minor INTEGER REFERENCES subject(id),
    --
}
CREATE TABLE graduate{
    studentid INTEGER REFERENCES student(id) NOT NULL,
    degree TEXT NOT NULL CHECK (degree="PhD" OR degree="MS"),
    advisor INTEGER REFERENCES faculty(id) NOT NULL,
    --
}
CREATE TABLE student{
    id SERIAL PRIMARY KEY,
    pid TEXT NOT NULL UNIQUE,
    name TEXT NOT NULL,
    major INTEGER REFERENCES subject(id),
    --
}
CREATE TABLE faculty{
    id SERIAL PRIMARY KEY,
    --
}
CREATE TABLE subject{
    id SERIAL PRIMARY KEY,
    --
}
CREATE TABLE undergrad{
    studentid INTEGER REFERENCES student(id) NOT NULL,
    minor INTEGER REFERENCES subject(id),
    --
}
CREATE TABLE graduate{
    studentid INTEGER REFERENCES student(id) NOT NULL,
    degree TEXT NOT NULL CHECK (degree="PhD" OR degree="MS"),
    advisor INTEGER REFERENCES faculty(id) NOT NULL,
    --
}
CREATE TABLE student{
    id SERIAL PRIMARY KEY,
    pid TEXT NOT NULL UNIQUE,
    name TEXT NOT NULL,
    major INTEGER REFERENCES subject(id),
    --
}
CREATE TABLE faculty{
    id SERIAL PRIMARY KEY,
    --
}
CREATE TABLE subject{
    id SERIAL PRIMARY KEY,
    --
}
CREATE TABLE undergrad{
    studentid INTEGER REFERENCES student(id) NOT NULL,
    minor INTEGER REFERENCES subject(id),
    --
}
CREATE TABLE graduate{
    studentid INTEGER REFERENCES student(id) NOT NULL,
Schemas for subclassing: Candidate 2

```sql
CREATE TABLE student(
    ID SERIAL PRIMARY KEY,
    pid TEXT NOT NULL UNIQUE,
    name TEXT NOT NULL,
    kind CHAR(1) CHECK (kind='U' OR kind='S'),
    major INTEGER REFERENCES subject(ID),
    minor INTEGER REFERENCES subject(ID),
    degree TEXT CHECK (degree="PhD" OR degree="MS"),
    advisor INTEGER REFERENCES faculty(ID)
)
```

```sql
CREATE TABLE subject(
    ID SERIAL PRIMARY KEY,
    ...
)
```

```sql
CREATE TABLE faculty(
    ID SERIAL PRIMARY KEY,
    ...
)
```

- misses constraints
  E.g., notice that it does not capture that a graduate student must have an advisor since we had to make the advisor attribute non-required in order to accommodate having undergraduates in the same table

---

Not covered E/R features

- Weak entities
  - double-lined entities and relationships
- Necessary participation of entity in relationship
- ... more

---

Writing programs on databases: JDBC

- How client opens connection with a server
- How access & modification commands are issued
- ...

---
... some easy hands-on experience

- Install the Postgresql open source database
- For educational and management purposes use the pgAdmin client to define schemas, insert data,
  - See online instructions
- For managing and accessing the Postgresql server, use the pgAdmin graphical client
  - Right click on Postgresql, and select Connect
  - Right click on Databases, and select New Database
  - Enter a new name for the database, and click Okay
  - Highlight the database, and select Tools -> Query Tool
  - Write SQL code (or open the examples), and select Query -> Execute

Creating a schema and inserting some data

- Open file enrollment.sql
- Copy and paste its CREATE TABLE and INSERT commands in the Query Tool
- Run it – you now have the sample database!
- Run the first 3 SELECT commands to see the data you have in the database

Access (Query) & Modification Language: SQL

- SQL
  - used by the database user
  - declarative: we only describe what we want to retrieve
  - based on tuple relational calculus
- The result of a query is a table (regardless of the query language used)
- Internal Equivalent of SQL: Relational Algebra
  - procedural (operational): we describe how we retrieve
- CSE132A, CSE232A
- The solutions to the following examples are on the class page download
### SQL Queries: Basic One-table

- Basic form
  
  ```
  SELECT A_1, ..., A_n
  FROM R
  WHERE <condition>
  ```

- Find all tuples of R that satisfy the (boolean) condition and return their attributes A_1, ..., A_n

- WHERE clause is optional

### SQL Queries: Putting together multiple tables

- Basic form
  
  ```
  SELECT A_1, ..., A_n
  FROM R_1, ..., R_m
  WHERE <condition>
  ```

- When more than one relations in the FROM clause have an attribute named A, we refer to a specific A attribute as `<RelationName>.A`

- Hardest to get used to, yet most important feature of SQL

---

#### (repeat)

<table>
<thead>
<tr>
<th>Classes</th>
<th>id</th>
<th>name</th>
<th>number</th>
<th>date_code</th>
<th>start_time</th>
<th>end_time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Web stuff</td>
<td>CS111X</td>
<td>TuTh</td>
<td>2:00</td>
<td>3:20</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Databases</td>
<td>CS130A</td>
<td>TuTh</td>
<td>3:30</td>
<td>4:50</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>VLSI</td>
<td>CS121</td>
<td>F</td>
<td>null</td>
<td>null</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Enrollment</th>
<th>id</th>
<th>class</th>
<th>student</th>
<th>credits</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>1</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Students</th>
<th>id</th>
<th>pid</th>
<th>first_name</th>
<th>last_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>88888888</td>
<td>John</td>
<td>Smith</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>111111111</td>
<td>Mary</td>
<td>Doe</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>222222222</td>
<td>null</td>
<td>Chen</td>
<td></td>
</tr>
</tbody>
</table>
Take One: Understanding FROM as producing all combinations

```
SELECT  students.pid, students.first_name, students.last_name, enrollment.credits
FROM    students, enrollment
WHERE   students.id = enrollment.student
        AND enrollment.class = 1 ;
```

"FROM" produces all 12 tuples made from one "students" tuple and one "enrollment" tuple

Students part of the tuple

<table>
<thead>
<tr>
<th>id</th>
<th>pid</th>
<th>first_name</th>
<th>last_name</th>
<th>credits</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>88</td>
<td>John</td>
<td>Smith</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>88</td>
<td>John</td>
<td>Smith</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>11</td>
<td>Mary</td>
<td>Doe</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>11</td>
<td>Mary</td>
<td>Doe</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>null</td>
<td>Chen</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>null</td>
<td>Chen</td>
<td></td>
<td>3</td>
</tr>
</tbody>
</table>

Enrollment part of the tuple

<table>
<thead>
<tr>
<th>id</th>
<th>class</th>
<th>student</th>
<th>credits</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>88</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>null</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>null</td>
<td>3</td>
</tr>
</tbody>
</table>

Understanding WHERE as qualifying the tuples that satisfy the condition

```
```

Understanding SELECT as keeping the listed columns (highlighted below)

```
```
Generalize to any number of tables

Produce a table that shows the pid, first name and last name of every student enrolled in the CSE135 class along with the number of credit units in his/her 135 enrollment

```
SELECT  students.pid, students.first_name, students.last_name, enrollment.credits
FROM    students, enrollment, classes
WHERE   classes.number = 'CSE135'
AND      students.id = enrollment.student
AND      enrollment.class = classes.id ;
```

How to understand the FROM and WHERE (at least until we talk about duplicates):
Find the students, whose students.id appears in an enrollment tuple as enrollment.student, and the enrollment.class of this tuple is the class.id of a class tuple whose number is CSE135.

You can omit table names in SELECT, WHERE when attribute is unambiguous

```
SELECT pid, first_name, last_name, credits
FROM   students, enrollment, classes
WHERE  number = 'CSE135'
AND     students.id = student
AND     class = classes.id ;
```

Take Two on the previous exercises: The algebraic way to express joins

Produce a table that shows the pid, first name and last name of every student enrolled in the class with ID 1, along with the number of credit units in the “class 1” enrollment

```
SELECT students.pid, students.first_name, students.last_name, enrollment.credits
FROM   students, enrollment
JOIN    enrollment
ON      students.id = enrollment.student
WHERE   enrollment.class = 1 ;
```
Take two cont’d

<table>
<thead>
<tr>
<th>Students.id</th>
<th>pid</th>
<th>first_name</th>
<th>last_name</th>
<th>Enrollment.id</th>
<th>class</th>
<th>student</th>
<th>credits</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>88</td>
<td>John</td>
<td>Smith</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>11</td>
<td>Mary</td>
<td>Doe</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>null</td>
<td>Chen</td>
<td></td>
<td>3</td>
<td></td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>null</td>
<td>Chen</td>
<td></td>
<td>4</td>
<td>1</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

SELECT clause result

<table>
<thead>
<tr>
<th>Students.pid</th>
<th>Students.first_name</th>
<th>Students.last_name</th>
<th>Enrollment.credits</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>John</td>
<td>Smith</td>
<td>4</td>
</tr>
<tr>
<td>11</td>
<td>Mary</td>
<td>Doe</td>
<td>3</td>
</tr>
<tr>
<td>22</td>
<td>null</td>
<td>Chen</td>
<td>3</td>
</tr>
</tbody>
</table>

Net result of the query is

Tae two (algebraic approach)
Second example

Produce a table that shows the pid, first name and last name of every student enrolled in the CSE135 class along with the number of credit units in his/her 135 enrollment

Take One:

```sql
SELECT students.pid, students.first_name, students.last_name, enrollment.credits
FROM students, enrollment, classes
WHERE classes.number = 'CSE135'
AND students.id = enrollment.student
AND enrollment.class = classes.id
```

Take Two:

```sql
SELECT students.pid, students.first_name, students.last_name, enrollment.credits
FROM (students JOIN enrollment ON students.id = enrollment.student)
JOIN classes ON enrollment.class = classes.id
WHERE classes.number = 'CSE135'
```

SQL Queries: Nesting

- The WHERE clause can contain predicates of the form
  - `attr/value IN <query>`
  - `attr/value NOT IN <query>`

- The predicate is satisfied if the `attr` or `value` appears in the result of the nested `<query>`

- Also
  - `EXISTS <query>`
  - `NOT EXISTS <query>`
Nested subquery example (uncorrelated subquery)

Produce a table that shows the pid, first and last name of every student enrolled in CSE135

```
SELECT pid, first_name, last_name
FROM students
WHERE id IN
  ( SELECT student
     FROM enrollment, classes
     WHERE number='CSE135'
     AND class=classes.id
  )
```

"Uncorrelated" in the sense that the nested query could be a standalone query

Nested queries help modularize the task. Nested query finds the id's of the students who take CSE135. Then the outer query prints out pid and name for every student whose id appears in the result of the nested query.

Nested subquery example, correlated

```
SELECT pid, first_name, last_name
FROM students
WHERE EXISTS
  ( SELECT *
     FROM enrollment, classes
     WHERE number='CSE135'
     AND class=classes.id
     AND student = students.id
  )
```

Correlation of nested query to outside query. The nested query is not a standalone.

There may be IN queries that are correlated and EXISTS queries that are uncorrelated.

SQL Queries, advanced: Aliases

- Use the same relation more than once in the FROM clause
- Tuple variables
- Problem: Find the other classes taken by students who take CSE135
  - First, also showing the students, i.e., produce a table where each row has the name of a 135 student and the name of another class he/she takes
produce a table where each row has the name of a 135 student and the name of another class he/she takes

```sql
SELECT c_others.name, first_name, last_name
FROM classes AS c_135, enrollment AS e_135,
     students,
     enrollment AS e_others, classes AS c_others
WHERE c_135.number = 'CSE135'
  AND c_135.id = e_135.class
  AND e_135.student = students.id
  AND students.id = e_others.student
  AND e_others.class = c_others.id
  AND NOT (c_others.number = 'CSE135')
```

Second, show just the other classes. Notice use of DISTINCT

```sql
SELECT DISTINCT c_others.name
FROM classes AS c_135, enrollment AS e_135,
     enrollment AS e_others, classes AS c_others
WHERE c_135.number = 'CSE135'
  AND c_135.id = e_135.class
  AND e_135.student = e_others.student
  AND e_others.class = c_others.id
  AND NOT (c_others.number = 'CSE135')
```

Use of nested subqueries may reduce need for aliases => easier to write, read

Find the CSE135 students who take a Friday 11:00 am class

```sql
SELECT first_name, last_name
FROM students, enrollment, classes
WHERE students.id = student
  AND class = classes.id
  AND number = 'CSE135'
  AND students.id IN
  (  
      SELECT student
        FROM enrollment, classes
        WHERE classes.id = class
        AND date_code = 'F'
        AND start_time = '11:00'
      )
```
SQL Queries: Aggregation & Grouping

- Aggregate functions: SUM, AVG, COUNT, MIN, MAX, and recently user-defined functions as well
- **GROUP BY**

<table>
<thead>
<tr>
<th>Name</th>
<th>Dept</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Joe</td>
<td>Toys</td>
<td>45</td>
</tr>
<tr>
<td>Nick</td>
<td>PCs</td>
<td>50</td>
</tr>
<tr>
<td>Jim</td>
<td>Toys</td>
<td>35</td>
</tr>
<tr>
<td>Jack</td>
<td>PCs</td>
<td>40</td>
</tr>
</tbody>
</table>

**Example:** Find the average salary of all employees:

```sql
SELECT AVG(Salary) AS AvgSal
FROM Employee
```

**Example:** Find the average salary for each department:

```sql
SELECT Dept, AVG(Salary) AS AvgSal
FROM Employee
GROUP BY Dept
```

**SQL Grouping:** Conditions that Apply on Groups

- **HAVING** `<condition>` may follow a **GROUP BY** clause
- If so, the condition applies to each group, and groups not satisfying the condition are eliminated

**Example:** Find the average salary in each department that has more than 1 employee:

```sql
SELECT Dept, AVG(Salary) AS AvgSal
FROM Employee
GROUP BY Dept
HAVING COUNT(Name) > 1
```

Let's mix features we’ve seen: Aggregation after joining tables

- **Problem:** List all enrolled students and the number of total credits for which they have registered

```sql
SELECT students.id, first_name, last_name, SUM(credits)
FROM students, enrollment
WHERE students.id = enrollment.student
GROUP BY students.id, first_name, last_name
```
The outerjoin operator

- New construct in FROM clause

<table>
<thead>
<tr>
<th>R</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>RJ</td>
<td>RV</td>
</tr>
<tr>
<td>1</td>
<td>RV1</td>
</tr>
<tr>
<td>2</td>
<td>RV2</td>
</tr>
<tr>
<td>Null</td>
<td>Null</td>
</tr>
</tbody>
</table>

- R LEFT OUTER JOIN S ON R.<attr of R> >= S.<attr of J>

<table>
<thead>
<tr>
<th>R</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>RJ</td>
<td>SJ</td>
</tr>
<tr>
<td>1</td>
<td>RV1</td>
</tr>
<tr>
<td>2</td>
<td>RV2</td>
</tr>
<tr>
<td>Null</td>
<td>Null</td>
</tr>
</tbody>
</table>

- R FULL OUTER JOIN S ON R.<attr of R> >= S.<attr of J>

<table>
<thead>
<tr>
<th>R</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>RJ</td>
<td>SJ</td>
</tr>
<tr>
<td>1</td>
<td>RV1</td>
</tr>
<tr>
<td>2</td>
<td>RV2</td>
</tr>
<tr>
<td>Null</td>
<td>Null</td>
</tr>
</tbody>
</table>

An application of outerjoin

- **Problem:** List all students and the number of total credits for which they have registered
  - Notice that you must also list non-enrolled students

```
SELECT students.id, first_name, last_name, SUM(credits)
FROM students LEFT OUTER JOIN enrollment ON students.id = enrollment.student
GROUP BY students.id, first_name, last_name
```

SQL: More Bells and Whistles ...

- Pattern matching conditions
  - `<attr> LIKE <pattern>`
  - Retrieve all students whose name contains "Sm"

```
SELECT *
FROM Students
WHERE name LIKE '%Sm%
```
...and a Few “Dirty” Points

- **Null values**
  - All comparisons involving NULL are false by definition
  - All aggregation operations, except `COUNT(*)`, ignore NULL values

---

Null Values and Aggregates

- Example:

```sql
SELECT COUNT(a), COUNT(b), AVG(b), COUNT(*)
FROM R
GROUP BY a
```

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>null</td>
<td></td>
<td></td>
</tr>
<tr>
<td>null</td>
<td>null</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

Universal Quantification by Negation

Problem:
- Find the students that take every class 'John Smith' takes

Rephrase:
- Find the students such that there is no class that 'John Smith' takes and they do not take
Preview of when you need top-class SQL writing ability

- I don't get SQL. How can I avoid it?
  By using Object Relational Mappers (ORMs) you can avoid writing simple SQL queries
  Also avoid interfacing query results with the programming language objects
  - Ruby’s active record later in class
  - Hibernate
- SQL ability mostly needed in analytic queries:
  - Join data, aggregate
- Conceptually the trouble is the join
  - After years of being trained in one-step-at-a-time algorithms, it takes a while to get declarative programming

Hints for writing FROM/WHERE: Rephrase the statement, see it as a navigation across primary/foreign keys

Produce a table that shows the pid, first name and last name of every student enrolled in the CSE135 class along with the number of credit units in his/her 135 enrollment
Find any students tuple s,
- that is connected to an enrollment tuple e as s.student,
- and e is connected to a classes tuple c
  - i.e., the e.class of e appears as c.id of the tuple c,
- whose c.number is CSE135

Take One: Declarative
FROM students AS s,
enrollment AS e,
classes AS c
WHERE s.id = e.student
    AND c.id = e.class

Take Two: Algebraic
FROM (students AS s
     JOIN enrollment AS e
     ON s.id = e.student)
     JOIN classes AS c
     ON c.id = e.class
WHERE c.number = 'CSE135'
How to solve in easy steps the following complex query:

Create a table that shows all time slots (date, start time, end time) when students of CSE135 attend a lecture of another class; Show also how many students attend a class at each time slot.

**SQL as a Data Manipulation Language: Insertions**

- Inserting tuples
  
  \[
  \text{INSERT INTO } R(A_1, \ldots, A_k) \\
  \text{VALUES } (v_1, \ldots, v_k); \\
  \]

- Some values may be left NULL

- Use results of queries for insertion
  
  \[
  \text{INSERT INTO } R \\
  \text{SELECT } \ldots \\
  \text{FROM } \ldots \\
  \text{WHERE } \ldots
  \]

*Insert in Students 'John Doe' with A# 99999999*

\[
\text{INSERT INTO students} \\
(\text{pid, first_name, last_name}) \\
\text{VALUES} \\
(\text{'9999999', 'John', 'Doe'})
\]

*Enroll all CSE135 students into CSE132A*

\[
\text{INSERT INTO enrollment (class, student)} \\
\text{SELECT c132a.id, student} \\
\text{FROM classes AS c135, enrollment,} \\
\text{classes AS c132a} \\
\text{WHERE c135.number='CSE135' AND} \\
\text{enrollment.class =c135.id AND} \\
\text{c132a.number='CSE132A'}
\]

**SQL as a Data Manipulation Language: Updates and Deletions**

- Deletion basic form: delete every tuple that satisfies \!<\!cond\!>:
  
  \[
  \text{DELETE FROM } R \\
  \text{WHERE } \!<\!\text{cond}\!>
  \]

- Update basic form: update every tuple that satisfies \!<\!cond\!> in the way specified by the \!SET\! clause:
  
  \[
  \text{UPDATE } R \\
  \text{SET } A_i =<\!\exp_1\!>, \ldots, A_k =<\!\exp_k\!> \\
  \text{WHERE}<\!\text{cond}\!>
  \]

*Delete "John" "Smith"*

\[
\text{DELETE FROM students} \\
\text{WHERE first_name='John' AND} \\
\text{last_name='Smith'}
\]

*Update the registered credits of all CSE135 students to 5*

\[
\text{UPDATE enrollment} \\
\text{SET credits=5} \\
\text{WHERE class=1} \\
\text{UPDATE enrollment} \\
\text{SET credits=5} \\
\text{WHERE class IN (SELECT id FROM classes WHERE number='CSE135')}
\]