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

- **Persistent data structure**
  - Large volume of data

- **High-level language/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
**OLTP Vs OLAP use cases**

**OLTP**
- Support quick ACID transactions
- Eg, Bank application that manages transactions

**OLAP**
- Perform analytics on the database
- Eg, Bank application analyzing customer profiles towards marketing

- All well-known databases can do both
- But may not be very efficient in analytics
- Many new databases focused on analytics
  - Organizations may have two databases – OLTP vs OLAP
    - Or 3+
  - The jury is out on whether two kinds of databases will be needed

**Data Structure: Relational Model**

**Relational Databases:** Schema + Data

**Schema:**
- collection of *tables* (also called *relations*)
- each table has a set of *attributes* (aka *columns*)
- no repeating table names, no repeating attributes in one table

**Data** (also called *instance*):
- set of *tuples* (aka *rows*)
- tuples have one atomic *value* for each attribute

<table>
<thead>
<tr>
<th>Movie</th>
<th>ID</th>
<th>Title</th>
<th>Actor</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>Wild</td>
<td>Winger</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>Sky</td>
<td>Winger</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>Reds</td>
<td>Beatty</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>Tango</td>
<td>Brando</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>Tango</td>
<td>Winger</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>Tango</td>
<td>Snyder</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Schedule</th>
<th>ID</th>
<th>Theater</th>
<th>Movie</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>Odeon</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>Forum</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>Forum</td>
<td>2</td>
</tr>
</tbody>
</table>
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 example 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...
- Normalization rules of database design instruct how to turn a “bad” design into a “good” one
  - a well-developed mathematical theory
  - no guidance on how to start
  - does not solve all problems
- MAS 201: Think **entities and relationships** – then translate them to tables
- MAS 201: The special case of star & snowflake schemas

Designing Schemas Using Entity-Relationship modeling

The Basics
Example Problem:

- Represent the students classes of the CSE department in Winter, 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.
  - Dismiss the possibility of two Winter classes (or class sections) for the same course
- A student enrolls for a number of credits in a class.

Solution:

Example 1a: E/R-Based Design

```
<table>
<thead>
<tr>
<th>Attribute</th>
<th>Entity</th>
<th>Relationship</th>
</tr>
</thead>
<tbody>
<tr>
<td>DateCode</td>
<td>Classes</td>
<td></td>
</tr>
<tr>
<td>Number</td>
<td></td>
<td>Start</td>
</tr>
<tr>
<td>Name</td>
<td></td>
<td>End</td>
</tr>
<tr>
<td>FirstNam</td>
<td>Students</td>
<td></td>
</tr>
<tr>
<td>PID</td>
<td></td>
<td>LastNam</td>
</tr>
<tr>
<td>Credits</td>
<td>Enrollment</td>
<td></td>
</tr>
</tbody>
</table>
```

(2-way many-to-many) Relationship
**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 many-to-many 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>Classes</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>name</td>
<td>number</td>
<td>date_code</td>
<td>start_time</td>
<td>end_time</td>
</tr>
<tr>
<td>1</td>
<td>Web stuff</td>
<td>MAS201</td>
<td>TuTh</td>
<td>2:00</td>
<td>3:20</td>
</tr>
<tr>
<td>2</td>
<td>Databases</td>
<td>CSE132A</td>
<td>TuTh</td>
<td>3:30</td>
<td>4:50</td>
</tr>
<tr>
<td>4</td>
<td>VLSI</td>
<td>CSE121</td>
<td>F</td>
<td>null</td>
<td>null</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>4</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td></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>1</td>
<td>888888</td>
<td>John</td>
<td>Smith</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>111111</td>
<td>Mary</td>
<td>Doe</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>222222</td>
<td>null</td>
<td>Chen</td>
</tr>
</tbody>
</table>
Declaration of schemas in SQL’s Data Definition Language

CREATE TABLE classes (  
   ID          SERIAL PRIMARY KEY,  
   name        TEXT,  
   number      TEXT,  
   date_code   TEXT,  
   start_time  TIME,  
   end_time    TIME  
)
CREATE TABLE students (  
   ID          SERIAL PRIMARY KEY,  
   pid         INTEGER,  
   first_name  TEXT,  
   last_name   TEXT  
)
CREATE TABLE enrollment (  
   ID              SERIAL,  
   class           INTEGER REFERENCES classes (ID) NOT NULL,  
   student         INTEGER REFERENCES students (ID) NOT NULL,  
   credits         INTEGER  
)

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.

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)

Example 1b: Using a semantic, immutable key

Assume that each PID (the id number on UCSD cards) is unique, not null and immutable (will never change)
Example 1b: Sample, using the pid instead of the id to identify students

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

Classes

<table>
<thead>
<tr>
<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>MAS201</td>
<td>TuTh</td>
<td>2:00</td>
<td>3:20</td>
</tr>
<tr>
<td>2</td>
<td>Databases</td>
<td>CSE132A</td>
<td>TuTh</td>
<td>3:30</td>
<td>4:50</td>
</tr>
<tr>
<td>3</td>
<td>VLSI</td>
<td>CSE121</td>
<td>F</td>
<td>null</td>
<td>null</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>8888888</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1111111</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>2222222</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>2222222</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>

Example 1b: Schema revisited, for using pid for students’ primary key

CREATE TABLE classes (  
    ID SERIAL PRIMARY KEY,  
    name TEXT,  
    number TEXT,  
    date_code TEXT,  
    start_time TIME,  
    end_time TIME  
)  
CREATE TABLE students (  
    ID SERIAL PRIMARY KEY,  
    pid INTEGER PRIMARY KEY,  
    first_name TEXT,  
    last_name TEXT  
)  
CREATE TABLE enrollment (  
    ID SERIAL,  
    class INTEGER REFERENCES classes (ID) NOT NULL,  
    student INTEGER REFERENCES students (pid) NOT NULL,  
    credits INTEGER  
)
... 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,
- 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
  - You can run a command by highlighting it with the cursor and click run
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.

```sql
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-at-most-one relationship**

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

![Diagram](image)

**E/R⇒ Relational: Translation revisited for many-to-at-most-one relationship**

- For every entity, do the usual...
- For every **many-to-many** relationship, do the usual...
- For every **2-way many-to-at-most-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, i.e., potentially NULL) foreign key attribute referencing the ID of the table corresponding to $E_o$
Example 2c: many-to-exactly-one relationship

Modification to Example 2a:
A movie must be owned by one studio.

```sql
CREATE TABLE movies (  
    ID SERIAL PRIMARY KEY,  
    title TEXT,  
    year INTEGER,  
    length INTEGER,  
    owner INTEGER REFERENCES studios (ID) NOT NULL
)
```
Example 2d: one-to-one relationship

Consider a database of heterosexual couples (we neglect homosexual couples, amazons and Warren Jeffs followers)

![Diagram of one-to-one relationship]

```sql
CREATE TABLE couple (  
    husband INTEGER REFERENCES females (ID) NOT NULL UNIQUE,  
    wife INTEGER REFERENCES males (ID) NOT NULL UNIQUE  
)
```

Example 3: 3-Way Many-to-Many 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  
)
CREATE TABLE persons {
    ID SERIAL PRIMARY KEY,
    --
}

CREATE TABLE following {
    ID SERIAL,
    follows INTEGER REFERENCES persons (ID) NOT NULL,
    isFollowed INTEGER REFERENCES persons (ID) NOT NULL
}

Notice the use of roles as attributes names for the foreign keys

Example 4b: Self-Relationships with Roles

Prequels and Sequels

![Diagram of Movies, Prequel, Sequel, and SequelOf relationships]
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  
)  

Example 4b: Self-Relationships with Roles – Questions on Meaning

What exactly are the prequel-sequel pairs?

“Terminator II: Judgment Day” is a sequel of “Terminator”

“Terminator III: Raise of the Machines” is a sequel of  
“Terminator II: Judgment Day”

Is “Terminator III: Raise of the Machines” a sequel of  
“Terminator”?

"Terminator II: Judgment Day" is a sequel of “Terminator”

“Terminator III: Raise of the Machines” is a sequel of  
“Terminator II: Judgment Day”

Is “Terminator III: Raise of the Machines” a sequel of  
“Terminator”?
Example 4c: Interpreting sequels non-transitively

Is “Terminator III: Raise of the Machines” a direct sequel of “Terminator”? NO

A movie has at most one direct “prequel” and at most one direct “sequel”

Modeling movie sequels by “DirectSequelOf” is preferable in OLTP to using transitive “SequelOf”

A lesson about good (OLTP?) database design:
- Good designs avoid redundancy.
- No stored piece of data should be inferable from other stored pieces of data

To be Redundant or Not to be?

NOT
- Too many Friends-of-Friends
  - Even more Friends-of-Friends-of-Friends
    - If “Six Degrees of Separation” is true, the 6-step friends is not even saying anything
- A database with derivative data is harder to maintain

YES
- Some derivations, interesting to OLAP, are too expensive to compute live
- If OLAP, maintenance is not primary concern
**Self-relationships without roles**

Twitter “followship” is a self-relationship with roles

![Diagram of Twitter followship](https://via.placeholder.com/150)

Facebook “friendship” is a self-relationship without real roles

![Diagram of Facebook friendship](https://via.placeholder.com/150)

**A case where redundancy may be welcome**

```sql
CREATE TABLE friend (  
    subject INTEGER REFERENCES user (ID) NOT NULL,  
    object INTEGER REFERENCES user (ID) NOT NULL  
);
```

If Subject is Facebook friend of Object, then Object is Facebook friend of Subject. Is it redundant to explicitly represent both facts in “friend”? Yes, but makes some queries much easier and faster.
Example 5a: 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 5b: 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 5a, 5b

```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>ID</th>
<th>Name</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<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>ID</th>
<th>Name</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>20th Century Fox</td>
<td>Century City, CA</td>
</tr>
<tr>
<td>2</td>
<td>Paramount Productions</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>ID</th>
<th>Movie</th>
<th>Year</th>
<th>Length</th>
<th>Owner</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Forrest Gump</td>
<td>1994</td>
<td>142</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>The Godfather</td>
<td>1972</td>
<td>175</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>Star Wars</td>
<td>1977</td>
<td>121</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>Scent of a Woman</td>
<td>1992</td>
<td>157</td>
<td>3</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 I: 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
### Added constraints of previous slide

```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)
)
```

```sql
CREATE TABLE stars (
    ID SERIAL PRIMARY KEY,
    name TEXT NOT NULL,
    address TEXT
)
```

```sql
CREATE TABLE studios (
    ID SERIAL PRIMARY KEY,
    name TEXT NOT NULL UNIQUE,
    address TEXT
)
```

```sql
CREATE TABLE starsin (
    ID SERIAL,
    movie INTEGER REFERENCES movies (ID) NOT NULL,
    star INTEGER REFERENCES stars (ID) NOT NULL
)
```

### Example 6: tricky flavors of 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.

![Diagram](image.png)
1\textsuperscript{st} candidate

```
CREATE TABLE presidents (
    ID SERIAL PRIMARY KEY,
    name TEXT,
    age INTEGER,
)

CREATE TABLE studios (
    ID SERIAL PRIMARY KEY,
    name TEXT,
    address TEXT
)

CREATE TABLE management (
    manager INTEGER REFERENCES presidents (ID) NOT NULL UNIQUE,
    manages INTEGER REFERENCES studios (ID) NOT NULL UNIQUE
)
```

One may be a president WITHOUT managing any studio
=> This design fails to capture a given constraint

2\textsuperscript{nd} 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
)
```

Guarantees that in order to be president, one has to manage a studio
Guarantees that no two presidents may manage the same studio
3rd candidate

3rd candidate is also not preferred. Why? What constraint it misses?

```sql
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 6: Subclassing**

![Subclassing diagram](image)
Schemas for subclassing: Candidate 1

CREATE TABLE student(
    ID     SERIAL PRIMARY KEY,
    pid    TEXT NOT NULL UNIQUE,
    name   TEXT NOT NULL,
    major  INTEGER REFERENCES subject(ID)
)

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 subject(
    ID     SERIAL PRIMARY KEY,
    ...
)

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

Schemas for subclassing: Candidate 2

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)
)

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

CREATE TABLE faculty(
    ID     SERIAL PRIMARY KEY,
    ...
)
Not covered E/R features

- Weak entities
  - double-lined entities and relationships
- Many-to-Many-to-One 3-way (or more) relationships
- Necessary participation of entity in relationship
- ... more

Programming on Databases with SQL
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
- Internal Equivalent of SQL: Relational Algebra
  - used internally by the database system
  - procedural (operational): describes how query is executed
- The solutions to the following examples are on the class page download
SQL: Basic, single-table queries

- Basic form
  
  ```sql
  SELECT \( r.A_1, \ldots, r.A_N \)
  FROM \( R \)
  WHERE <condition>
  ```

- WHERE clause is optional

- Have tuple variable \( r \) range over the tuples of \( R \), qualify the ones that satisfy the (boolean) condition and return the attributes \( A_1, \ldots, A_N \)

  - Find first names and last names of all students
    
    ```sql
    SELECT s.first_name, s.last_name
    FROM students s;
    ```

  - Display all columns of all students whose first name is John; project all attributes
    
    ```sql
    SELECT s.id, s.pid, s.first_name, s.last_name
    FROM students s
    WHERE s.first_name = 'John';
    ```

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

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

SQL Queries: Joining together multiple tables

- Basic form
  
  ```sql
  SELECT \( ..., r_i.A_j, ... \)
  FROM \( R_1, r_1, ..., R_M, r_M \)
  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

  - 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
    
    ```sql
    SELECT s.pid, s.first_name, s.last_name, e.credits
    FROM students s, enrollment e
    WHERE s.id = e.student
    AND e.class = 1;
    ```
Take One: Understanding FROM as producing all combinations of tuples from the tables of the FROM clause

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

"FROM“ produces all 12 tuples made from one “students” tuple and one “enrollment” tuple
**Take One: or understanding FROM as nested loops (producing all combinations)**

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

for s ranging over students tuples
for e ranging over enrollment tuples
output a tuple with all s attributes and e attributes

<table>
<thead>
<tr>
<th>s.id</th>
<th>s.pid</th>
<th>s.first_name</th>
<th>s.last_name</th>
<th>e.id</th>
<th>e.class</th>
<th>e.student</th>
<th>e.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>4</td>
</tr>
<tr>
<td>1</td>
<td>88..</td>
<td>John</td>
<td>Smith</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>88..</td>
<td>John</td>
<td>Smith</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>88..</td>
<td>John</td>
<td>Smith</td>
<td>4</td>
<td>1</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>11..</td>
<td>Mary</td>
<td>Doe</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>4</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>2</td>
<td>11..</td>
<td>Mary</td>
<td>Doe</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>11..</td>
<td>Mary</td>
<td>Doe</td>
<td>4</td>
<td>1</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>22..</td>
<td>null</td>
<td>Chen</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>22..</td>
<td>null</td>
<td>Chen</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>22..</td>
<td>null</td>
<td>Chen</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>22..</td>
<td>null</td>
<td>Chen</td>
<td>4</td>
<td>1</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

**The order in FROM clause is unimportant**

- FROM students s, enrollment e
- FROM enrollment e, students s

produce the same combinations (pairs) of student + enrollment
... with shorter column names

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

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

<table>
<thead>
<tr>
<th>s.id</th>
<th>pid</th>
<th>first_name</th>
<th>last_name</th>
<th>e.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>4</td>
</tr>
<tr>
<td>1</td>
<td>88..</td>
<td>John</td>
<td>Smith</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>88..</td>
<td>John</td>
<td>Smith</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>88..</td>
<td>John</td>
<td>Smith</td>
<td>4</td>
<td>1</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>11..</td>
<td>Mary</td>
<td>Doe</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>4</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>2</td>
<td>11..</td>
<td>Mary</td>
<td>Doe</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>11..</td>
<td>Mary</td>
<td>Doe</td>
<td>4</td>
<td>1</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>22..</td>
<td>null</td>
<td>Chen</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>22..</td>
<td>null</td>
<td>Chen</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>22..</td>
<td>null</td>
<td>Chen</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>22..</td>
<td>null</td>
<td>Chen</td>
<td>4</td>
<td>1</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

Understanding WHERE as qualifying the tuples that satisfy the condition

SELECT  s.pid, s.first_name, s.last_name, e.credits
FROM    students s, enrollment e
WHERE s.id = e.student AND e.class = 1 ;
Understanding SELECT as keeping the listed columns (highlighted below)

<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>4</td>
</tr>
<tr>
<td>1</td>
<td>88..</td>
<td>John</td>
<td>Smith</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>88..</td>
<td>John</td>
<td>Smith</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>88..</td>
<td>John</td>
<td>Smith</td>
<td>4</td>
<td>1</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>11..</td>
<td>Mary</td>
<td>Doe</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>4</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>2</td>
<td>11..</td>
<td>Mary</td>
<td>Doe</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>11..</td>
<td>Mary</td>
<td>Doe</td>
<td>4</td>
<td>1</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>22..</td>
<td>null</td>
<td>Chen</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>22..</td>
<td>null</td>
<td>Chen</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>22..</td>
<td>null</td>
<td>Chen</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>22..</td>
<td>null</td>
<td>Chen</td>
<td>4</td>
<td>1</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

SELECT s.pid, s.first_name, s.last_name, e.credits

Take Two on the previous exercises: The algebraic way

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

FROM clause result

<table>
<thead>
<tr>
<th>s.id</th>
<th>pid</th>
<th>first_name</th>
<th>last_name</th>
<th></th>
<th></th>
<th></th>
<th></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>4</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>22..</td>
<td>null</td>
<td>Chen</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

WHERE clause result

<table>
<thead>
<tr>
<th>s.id</th>
<th>pid</th>
<th>first_name</th>
<th>last_name</th>
<th></th>
<th></th>
<th></th>
<th></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>4</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>22..</td>
<td>null</td>
<td>Chen</td>
<td>4</td>
<td>1</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

Net result of the query is

<table>
<thead>
<tr>
<th>s.pid</th>
<th>first_name</th>
<th>last_name</th>
<th>credits</th>
</tr>
</thead>
<tbody>
<tr>
<td>88..</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>

Heuristics on writing queries

- Do you understand how queries work but have difficulty writing these queries yourself?

- The following heuristics will help you translate a requirement expressed in English into a query
  - The key point is to translate informal English into a precise English statement about which tuples your query should find in the database
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 class 1, along with the number of credit units in his/her class 1 enrollment

- Find every enrollment tuple e
- that is an enrollment in class 1
  - i.e., e.class = 1
- and find the student tuple s that is connected to e
  - i.e., the student’s id s.id appears in the enrollment tuple e as the foreign key e.student
- display the pid, first_name, last_name of s and the credits of e

```
<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>4</td>
</tr>
</tbody>
</table>
```

```
<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>
</tbody>
</table>
```
• Find every enrollment tuple e
• that is an enrollment in class 1
  • i.e., e.class = 1
• and find the student tuple s that is connected to e
  • i.e., the student's id s.id appears in the enrollment tuple e as the foreign key e.student
• display the pid, first_name, last_name of s and the credits of e

FROM    enrollment e, students s
WHERE    e.class = 1
         AND e.student = s.id
FROM    enrollment e
JOIN    students s
ON       e.student = s.id
WHERE    e.class = 1

We could have also said “and find every student tuple s that is connected” but we used our knowledge that there is exactly one connected student and instead said “the student”

SQL Queries: Nesting

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

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

• Also
  - EXISTS <query>
  - NOT EXISTS <query>
Nesting: Break the task into smaller

Produce a table that shows the pid, first name and last name of every student enrolled in the class named `MAS201', along with the number of credit units in his/her `MAS201' enrollment
Note: We assume that there are no two classes with the same name

```
SELECT  s.pid, s.first_name, s.last_name, e.credits
FROM    students s, enrollment e
WHERE e.class = (SELECT c.id
FROM classes c
WHERE c.number = 'MAS201')
AND s.id = e.student

Nested queries modularize the task:
Nested query finds the id of the MAS201 class.
Then the outer query behaves as if there were a "1" in lieu of the subquery
```

IN

Produce a table that shows the pid, first name and last name of every student enrolled in the class named `MAS201', along with the number of credit units in his/her `MAS201' enrollment
Note: We assume that there are no two classes with the same name

```
SELECT  s.pid, s.first_name, s.last_name, e.credits
FROM    students s, enrollment e
WHERE e.class IN (SELECT c.id
FROM classes c
WHERE c.number = 'MAS201')
AND s.id = e.student
```
**Students + enrollments in class 1**

Vs **Students who enrolled in class 1**

Imagine a weird university where a student is allowed to enroll multiple times in the same class.

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.

=> The same student may appear many times, once for each enrollment.

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

Produce a table that shows the pid, first name and last name of every student who has enrolled at least once in the “class 1”.

=> Each student will appear at most once.

```sql
SELECT s.pid, s.first_name, s.last_name
FROM students s
WHERE s.id IN ( SELECT e.student
FROM enrollment e
WHERE e.class = 1 )
```

**Uncorrelated subquery**

“Uncorrelated” in the sense that the nested query could be a standalone query.

```sql
SELECT s.pid, s.first_name, s.last_name
FROM students s
WHERE s.id IN ( SELECT e.student
FROM enrollment e
WHERE e.class = 1 )
```

Some nested queries are correlated (example later).
**EXISTS**

Display the students enrolled in class 1, only if the enrollment of class 2 is not empty

```
SELECT  s.pid, s.first_name, s.last_name
FROM    students s
WHERE  s.id IN ( SELECT e.student
                FROM enrollment e
                WHERE e.class = 1
            )
        AND EXISTS ( SELECT *
                      FROM enrollment e
                      WHERE e.class = 2
                    )
```

**Correlated with EXISTS**

Display the students enrolled in class 1

```
SELECT  s.pid, s.first_name, s.last_name
FROM    students s
WHERE  EXISTS ( SELECT e.student
                FROM enrollment e
                WHERE e.class = 1
                AND e.student = s.id
            )
```

Correlation: the variable \( s \) comes from the outer query

Uncorrelated, also
**Exercise, on thinking cardinalities of tuples in the results**

```sql
SELECT s.pid, s.first_name, s.last_name, e.credits
FROM students s, enrollment e
WHERE e.class IN (SELECT c.id
    FROM classes c
    WHERE c.number = 'MAS201'
)
    AND s.id = e.student

EXERCISE: Under what condition the above query always produces the same result with the query below?

```sql
SELECT s.pid, s.first_name, s.last_name, e.credits
FROM students s, enrollment e, classes c
WHERE c.number = 'MAS201'
    AND s.id = e.student
    AND e.class = c.id
```

**Exercise: Multiple JOINs**

Produce a table that shows the pid, first name and last name of every student enrolled in the MAS201 class along with the number of credit units in his/her 135 enrollment

**Take One:**
```sql
SELECT s.pid, s.first_name, s.last_name, e.credits
FROM students s, enrollment e, classes c
WHERE c.number = 'MAS201' AND s.id = e.student AND e.class = c.id
```

**Take Two:**
```sql
SELECT s.pid, s.first_name, s.last_name, e.credits
FROM (students s JOIN enrollment e ON s.id = e.student)
    JOIN classes c ON e.class = c.id
WHERE c.number = 'MAS201'
```
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 = 'MAS201'
  AND students.id = student
  AND class = classes.id ;
```

---

**SQL Queries, Aliases**

- Use the same relation more than once in the same query or even the same FROM clause
- **Problem:** Find the Friday classes taken by students who take MAS201
  - also showing the students, i.e., produce a table where each row has the data of a MAS201 student and a Friday class he/she takes
Find the MAS201 students who take a Friday 11:00 am class

```sql
SELECT s.id, s.first_name, s.last_name, cF.number
FROM students s, enrollment eF, classes cF
WHERE date_code = 'F'
  AND eF.class = cF.id
  AND s.id = eF.student
  AND s.id IN
    (SELECT student
     FROM enrollment e201, classes c201
     WHERE c201.id = e201.class
     AND c201.number = 'MAS201'
    )
```

Nested query computes the id’s of students enrolled in MAS201.

Multiple aliases may appear in the same FROM clause

Find the MAS201 students who take a Friday 11:00 am class

```sql
SELECT s.first_name, s.last_name, cF.number
FROM students s, enrollment eF, classes cF,
    enrollment e201, classes c201
WHERE cF.date_code = 'F'
  AND eF.class = cF.id
  AND s.id = eF.student
  AND s.id = e201.student
  AND c201.id = e201.class
  AND c201.number = 'MAS201'
```

Under what conditions it computes the same result with previous page?
**DISTINCT**

Find the other classes taken by MAS201 students (I don’t care which students)

```sql
SELECT DISTINCT cOther.number
FROM enrollment eOther, classes cOther, enrollment e201, classes c201
WHERE eOther.class = cOther.id
  AND eOther.student = e201.student
  AND c201.id = e201.class
  AND c201.number = 'MAS201'
```

**UNION ALL**

Find all student ids for students who have taken class 1 or are named ‘John’

```sql
( SELECT e.student
  FROM enrollment e
  WHERE e.class = 1 )
UNION ALL
( SELECT s.id AS student
  FROM student s
  WHERE s.first_name = 'John'
)
```

If a student named John takes class 1 he will appear twice in the result
UNION with non-duplicate results

( SELECT e.student
  FROM enrollment e
  WHERE e.class=1 )

UNION

( SELECT s.id AS student
  FROM student s
  WHERE s.first_name='John' )

SQL Queries: Aggregation & Grouping

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

<table>
<thead>
<tr>
<th>Employee</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Dept</td>
<td>Salary</td>
</tr>
<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:

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

<table>
<thead>
<tr>
<th>AvgSal</th>
</tr>
</thead>
<tbody>
<tr>
<td>42.5</td>
</tr>
</tbody>
</table>

Example: Find the average salary for each department:

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

<table>
<thead>
<tr>
<th>Dept</th>
<th>AvgSal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Toys</td>
<td>40</td>
</tr>
<tr>
<td>PCs</td>
<td>45</td>
</tr>
</tbody>
</table>
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
  ```
**ORDER BY and LIMIT**

Order the student id’s of class 2 students according to their class 2 credits, descending

```sql
SELECT e.student
FROM enrollment e
WHERE e.class = 2
ORDER BY e.credits DESC
```

Order the student id’s of class 2 students according to their class 2 credits, descending and display the Top 10

```sql
SELECT e.student
FROM enrollment e
WHERE e.class = 2
ORDER BY e.credits DESC
LIMIT 10
```

**Combining features**

Find the Top-5 classes taken by students of class 2, i.e., the 5 classes (excluding class 2 itself) with the highest enrollment of class 2 students, display their numbers and how many class 2 students they have

```sql
SELECT cF.number, COUNT(*)
FROM enrollment e, classes c
WHERE eF.class = cF.id
AND NOT(eF.class = 2)
AND eF.student IN (
    SELECT student
    FROM enrollment e2
    WHERE e201.class = 2
)
GROUP BY cF.id, cF.number
ORDER BY cF.number
LIMIT 5
```
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>SJ</td>
</tr>
<tr>
<td>RV</td>
<td>SV</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>RV</td>
<td>SV</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>RV</td>
<td>SV</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
```
• 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:

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

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

<table>
<thead>
<tr>
<th>count(a)</th>
<th>count(b)</th>
<th>avg(b)</th>
<th>count(*)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>2</td>
<td>1.5</td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>null</td>
<td>2</td>
</tr>
</tbody>
</table>

Universal Quantification by Negation (difficult)

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
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 MAS201 class along with the number of credit units in his/her 135 enrollment.

- Find every combination of a students tuple s, an enrollment tuple e, c where
  - the students tuple s,
  - is connected to the enrollment tuple e
    - i.e., the s.id appears in the enrollment tuple e as e.student,
  - and e is connected to the classes tuple c
    - i.e., the e.class of e appears as c.id of the tuple c,
  - whose c.number is MAS201

FROM students AS s, enrollment AS e, classes AS c
WHERE s.id = e.student
AND c.id = e.class
AND c.number = 'MAS201'

Take One: Declarative
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 = 'MAS201'

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 = 'MAS201'
Find any students tuple s,
that is connected to an enrollment tuple e
  • i.e., whose s.id appears in an enrollment tuple e as e.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 MAS201

**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 = 'MAS201'
Find any students tuple s,
• that is connected to an enrollment tuple e
  • i.e., whose s.id appears in an enrollment tuple e
    as e.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 MAS201

Take One: Declarative
FROM students AS s,
WHERE s.id = e.student

Take Two: Algebraic
FROM students AS s,
JOIN enrollment AS e
ON s.id = e.student

Breaking a query into pieces with WITH

Find the 5 classes whose students have the busiest coursework, i.e.,
the 5 classes whose students have the highest average of quarter credits

WITH coursework AS
  ( SELECT e.student, SUM(credits) AS total_credits
  FROM enrollment e
  GROUP BY e.student )
SELECT e.class, AVG(c.total_credits)
FROM enrollment e, coursework c
WHERE e.student = c.student
GROUP BY e.class
ORDER BY AVG(c.total_credits) DESC
LIMIT 5

Definitions a table “coursework” that lives for the duration of this query only
Avoid repeating aggregates

WITH courseload AS
( SELECT e.student, SUM(credits) AS total_credits
  FROM enrollment e
  GROUP BY e.student )
SELECT e.class, \text{AVG}(c\text{.total\_credits})
FROM enrollment e, courseload c
WHERE e.student = c.student
GROUP BY e.class
ORDER BY \text{AVG}(c\text{.total\_credits}) \text{ DESC}
LIMIT 5

Breaking a query into pieces with nesting in the FROM clause

Find the 5 classes whose students have the busiest courseload, i.e.,
the 5 classes whose students have the highest average of quarter credits

SELECT e.class, AVG(c.total_credits) AS credits_avg
FROM enrollment e,
( SELECT e.student, SUM(credits) AS total_credits
  FROM enrollment e
  GROUP BY e.student ) c
WHERE e.student = c.student
GROUP BY e.class
ORDER BY credits_avg DESC
LIMIT 5
Find the 5 classes whose students have the busiest coursework, i.e., the 5 classes whose students have the highest average of quarter credits.

```
SELECT e.class, AVG(
    ( SELECT SUM(es.credits)
      FROM enrollment es
      WHERE es.student = e.student
    ) AS credits_avg
  )
FROM enrollment e
GROUP BY e.class
ORDER BY credits_avg DESC
LIMIT 5
```

**Discussed in class and discussion section**

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 MAS201 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
  \[\text{INSERT INTO students (pid, first_name, last_name) VALUES ('99999999', 'John', 'Doe')}\]
  - Enroll all MAS201 students into CSE132A
    \[\text{INSERT INTO enrollment (class, student) SELECT c132a.id, student FROM classes AS c135, enrollment, classes AS c132a WHERE c135.number='MAS201' AND enrollment.class=c135.id AND c132a.number='CSE132A'}\]

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:
  \[\text{UPDATE } R \text{ SET } A_1=\langle\text{exp}_1\rangle, \ldots, A_k=\langle\text{exp}_k\rangle \text{ WHERE } <\text{cond}>\]

- Delete “John” “Smith”
  \[\text{DELETE FROM students WHERE first_name='John' AND last_name='Smith'}\]
  - Update the registered credits of all MAS201 students to 5
    \[\text{UPDATE enrollment SET credits=5 WHERE class IN (SELECT id FROM classes WHERE number='MAS201')}\]