CS 232A: Database System Principles

Introduction: Prerequisites checklist & Course Overview

Introduction

• (Quick) Applications’ View of a Relational Database Management System (RDBMS)
• The Big Picture of UCSD’s DB program
• (Quick) Relational Model Overview
• (Quick) SQL Overview
Applications’ View of a Relational Database Management (RDBMS) System

- Applications: ..........
- Persistent data structure
  - Large volume of data
  - “Independent” from processes using the data
- SQL high-level programming interface for access & modification
  - Automatically optimized
- Transaction management (ACID)
  - Atomicity: all or none happens, despite failures & errors
  - Concurrency
  - Isolation: appearance of “one at a time”
  - Durability: recovery from failures and other errors

CSE232A and the rest of UCSD’s database course program

- CSE132A: Basics of relational database systems
  - Application view orientation
  - Basics on algebra, query processing
- CSE132B: Application-oriented project course
  - How to design and use in applications complex databases
  - Active database aspects and materialized views
  - JDBC issues
- CSE135: Online Analytics Applications
  - Data cubes
  - Live analytics dashboards
  - Application server aspects pertaining to JDBC
CSE232A and the rest of UCSD’s database course program

- **CSE232** is mostly about how databases work internally
  - rather than how to make databases for applications
  - yet, knowing internals makes you a master database programmer

- **CSE233**: Database Theory
  - Theory of query languages
  - Deductive and Object-Oriented databases

- **CSE232B**: Advanced Database Systems
  - Non-conventional database systems, such as
    - mediators & distributed query processing
    - object-oriented and XML databases

- **CSE291**: Databases & ML

---

**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

---

**Movie**

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

**Schedule**

<table>
<thead>
<tr>
<th>ID</th>
<th>Theater</th>
<th>Movie</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Odeon</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Forum</td>
<td>3</td>
</tr>
<tr>
<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 and common sense...
• 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
• Think entities and relationships – then translate them to tables
• The special case of star & snowflake schemas

Designing Schemas Using Entity-Relationship modeling

The Basics
**Data Structure: Relational Model**

**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**

[Diagram showing entities and relationships]

- Classes: Name, Number, Start, DateCode, End
- Students: PID, FirstName, LastName
- Enrollment: Credits

(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>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>
```

```
<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>CSE135</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>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>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>3</td>
<td>3</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>4</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>3</td>
<td>3</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  
)  
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

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>

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>CSE135</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>

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>

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

```sql
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  
)  
```
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-at-most-one relationship

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

The movie-studio relationship is a many-to-at-most-one relationship. "Movie" is the "many" side. "Studio" is the "one" side.

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 $E_o$
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: many-to-exactly-one relationship

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

CREATE TABLE movies (  
  ID SERIAL PRIMARY KEY,  
  title TEXT,  
  year INTEGER,  
  length INTEGER,  
  owner INTEGER REFERENCES studios (ID) NOT NULL  
)
A sample database

<table>
<thead>
<tr>
<th>movies</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>title</td>
<td>year</td>
<td>length</td>
<td>owner</td>
</tr>
<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>

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

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

<table>
<thead>
<tr>
<th>starsin</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>movie</td>
<td>star</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Programming on Databases with SQL
Writing programs on databases: JDBC

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

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
**SQL: Basic, single-table queries**

- Basic form
  ```sql
  SELECT r.A_1, ..., r.A_N
  FROM R 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, ..., 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';
```

**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>3</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 ;
```

<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>
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>1</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>88..</td>
<td>John</td>
<td>Smith</td>
<td>3</td>
<td>1</td>
<td>2</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>2</td>
<td>4</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>3</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>11..</td>
<td>Mary</td>
<td>Doe</td>
<td>4</td>
<td>4</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>3</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>4</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>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>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>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>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>

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 \( e \)

FROM enrollment \( e \)
WHERE \( e.class = 1 \)
• 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
ten 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
JOIN    students s
ON       e.student = s.id
WHERE    e.class = 1
SELECT s.pid, s.first_name, s.last_name
        e.credits
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 `CSE135', along with the number of credit units in his/her `CSE135' 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 = 'CSE135')
        AND s.id = e.student
```

Nested queries modularize the task:
Nested query finds the id of the CSE135 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 `CSE135', along with the number of credit units in his/her `CSE135' 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 = 'CSE135')
        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.

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.

```
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

```
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" in the sense that the nested query could be a standalone query.

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
Exercise, on thinking cardinalities of tuples in the results

```
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 = 'CSE135')
                        AND s.id = e.student
EXERCISE: Under what condition the above query always produces the same result with the query below?

SELECT  s.pid, s.first_name, s.last_name, e.credits
FROM    students s, enrollment e, classes c
WHERE   c.number = 'CSE135'
        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 CSE135 class along with the number of credit units in his/her 135 enrollment

**Take One:**
```
SELECT  s.pid, s.first_name, s.last_name, e.credits
FROM    students s, enrollment e, classes c
WHERE   c.number = 'CSE135' AND s.id = e.student AND e.class = c.id
```

**Take Two:**
```
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 = '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;

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 CSE135
  - also showing the students, i.e., produce a table where each row has the data of a CSE135 student and a Friday class he/she takes
Find the CSE135 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 e135.student
     FROM enrollment e135, classes c135
     WHERE c135.id = e135.class
     AND c135.number = 'CSE135'
    )
```

Nested query computes the id’s of students enrolled in CSE135

**Multiple aliases may appear in the same FROM clause**

Find the CSE135 students who take a Friday class

```sql
SELECT s.first_name, s.last_name, cf.number
FROM students s, enrollment ef, classes cf, enrollment e135, classes c135
WHERE cf.date_code = 'F'
    AND ef.class = cf.id
    AND s.id = ef.student =
    AND s.id = e135.student
    AND c135.id = e135.class
    AND c35.number = 'CSE135'
```

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

Find the other classes taken by CSE135 students
(I don’t care which students)

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 = 'CSE135'

UNION ALL

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

( 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'
)
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:

```sql
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:

```sql
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:
```
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

```
SELECT s.id, s.first_name, s.last_name, SUM(e.credits) AS TotalCreds
FROM students s, enrollment e
WHERE s.id = e.student
GROUP BY s.id, s.first_name, s.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 e.class = cF.id
AND NOT(e.class = 2)
AND e.student IN
  (SELECT student
   FROM enrollment e2
   WHERE e2.class = 2
  )
GROUP BY cF.id, cF.number
ORDER BY COUNT(*)
LIMIT 5
```

Grouping by both id and number ensures correctness even if multiple classes have the same number
### The outerjoin operator

- New construct in FROM clause

<table>
<thead>
<tr>
<th></th>
<th>R</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>RJ</td>
<td>RV</td>
<td>SJ</td>
</tr>
<tr>
<td>1</td>
<td>RV1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>RV2</td>
<td>3</td>
</tr>
</tbody>
</table>

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

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

```sql
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:

<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
Finding the 5 classes whose students have the busiest course load, i.e., the 5 classes whose students have the highest average of quarter credits.

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

Avoid repeating aggregates

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

Equivalent
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
```

Also defines a table, identical to the "courseload" except it has no name

and nesting in the SELECT 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(
    ( 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
```
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.

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

\[
\text{INSERT INTO } R \\
\text{SELECT } ... \\
\text{FROM } ... \\
\text{WHERE } ... 
\]

Insert in Students 'John Doe' with A# 99999999

\[
\text{INSERT INTO students} \\
\text{(pid, first_name, last_name)} \\
\text{VALUES} \\
\text{('99999999', 'John', 'Doe')}
\]

Enroll all CSE135 students into CSE132A

\[
\text{INSERT INTO enrollment (class, student)} \\
\text{SELECT c132a.id, student} \\
\text{FROM classes AS c135, enrollment, classes AS c132a} \\
\text{WHERE c135.number='CSE135' AND} \\
\text{enrollment.class=c135.id AND} \\
\text{c132a.number='CSE132A'}
\]
SQL as a Data Manipulation Language: Updates and Deletions

- Deletion basic form: delete every tuple that satisfies \(<con>\):
  
  ```sql
  DELETE FROM R
  WHERE \(<con>\)
  ```

- Update basic form: update every tuple that satisfies \(<con>\) in the way specified by the `SET` clause:
  
  ```sql
  UPDATE R
  SET \(A_1 = <exp_1>, \ldots, A_k = <exp_k>\)
  WHERE \(<con>\)
  ```

- Delete "John" "Smith"
  
  ```sql
  DELETE FROM students
  WHERE first_name = 'John' AND
  last_name = 'Smith'
  ```

- Update the registered credits of all CSE135 students to 5
  
  ```sql
  UPDATE enrollment
  SET credits = 5
  WHERE class = 1
  ```

  ```sql
  UPDATE enrollment
  SET credits = 5
  WHERE class IN
  (SELECT id FROM classes
   WHERE number = 'CSE135')
  ```

Course Topics

- **Hardware aspects** (very brief)
- **Physical Organization Structure** (very brief)
  
  Records in blocks, dictionary, buffer management,…

- **Indexing**
  
  B-Trees, hashing,…

- **Query Processing**
  
  rewriting, physical operators, cost-based optimization, semantic optimization,…

- **Crash Recovery**
Course Topics

- Concurrency Control
  Correctness, locks, deadlocks...
- Materialized views
  Incremental view maintenance, answering queries using views
- Federated databases
  Distributed query optimization
- Parallel query processing
- Column databases

Database System Architecture

Query Processing

- SQL query
- Parser
  relational algebra
- Query Rewriter and Optimizer
- View definitions
- Statistics & Catalogs & System Data
- Execution Engine
- Buffer Manager
- Data + Indexes

Transaction Management

- Calls from Transactions (read, write)
- Transaction Manager
- Concurrency Controller
  Lock Table
- Recovery Manager
- Log