Applications’ View of a Relational Database Management System (RDBMS)

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

- **Relational Databases:** Schema + Data
  - **Schema:**
    - collection of *tables* (also called *relations*)
    - each table has a set of *attributes*
    - no repeating relation names, no repeating attributes in one table
  - **Data** (also called *instance*):
    - set of *tuples*
    - tuples have one *value* for each attribute

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

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

### How to Design a Database and Avoid Bad Decisions

- With experience...
- Learn in CSE132A normalization rules of database design
  - a well-developed mathematical theory about how to fix step by step a “bad” schema
- CSE135: Think **entities and relationships** – then translate to relations
  - Do not confuse the data flow diagrams with the E/R diagrams: they are different formalisms and serve different purposes
Data Structure: Relational Model

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

Solution:...

Example 1: E/R-Based Design

![Diagram showing E/R-Based Design for students and classes enrollment]
E/R→ Relational Schema: Basic Translation

• For every entity
  - create corresponding table
  - For each attribute of the entity, add a corresponding attribute in the table
  - Include an ID attribute in the table even if not in E/R

• For every relationship
  - create corresponding table
  - For each attribute of the relationship, add a corresponding attribute in the table
  - For each referenced entity \( E_i \) include in the table a required foreign key attribute referencing ID of \( E_i \)

Sample relational database, per previous page’s algorithm

<table>
<thead>
<tr>
<th>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>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>Enrollment</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>class</td>
<td>student</td>
<td>credits</td>
</tr>
<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>

<table>
<thead>
<tr>
<th>Students</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>pid</td>
<td>first_name</td>
<td>last_name</td>
</tr>
<tr>
<td>1</td>
<td>8888888</td>
<td>John</td>
<td>Smith</td>
</tr>
<tr>
<td>2</td>
<td>1111111</td>
<td>Mary</td>
<td>Doe</td>
</tr>
<tr>
<td>3</td>
<td>2222222</td>
<td>null</td>
<td>Chen</td>
</tr>
</tbody>
</table>
### Declaration of schemas in SQL’s Data Definition Language

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

If we had "ID INTEGER PRIMARY KEY" we would be responsible for coming up with ID values. **SERIAL** leads to a counter that automatically provides ID values upon insertion of new tuples.

```sql
CREATE TABLE students (  
    ID SERIAL PRIMARY KEY,  
    pid TEXT,  
    first_name TEXT,  
    last_name TEXT  
)  
```

Foreign key declaration: Every value of `enrollment.class` must also appear as `classes.ID`.

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

Declaration of "required" constraint: `enrollment.student` cannot be null (notice, it would make no sense to have an enrollment tuple without a student involved).

---

### ... 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 8.4**, and select **Connect**
  - Right click on **Databases**, and select **New Database**
  - Enter a new name for the database, and click **Okay**
  - Highlight the database, and select **Tools -> Query Tool**
  - Write SQL code (or open the examples), and select **Query -> Execute**
Creating a schema and inserting some data

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

Example 2a

Movies have a title, a year of release and length (in minutes).
Actors have names and address.
Actors appear in movies.
A movie is (co-)owned by studios.
Studios have a name and address.
CREATE TABLE movies {
    ID SERIAL PRIMARY KEY,
    title TEXT,
    year INTEGER,
    length INTEGER,
}
CREATE TABLE stars {
    ID SERIAL PRIMARY KEY,
    name TEXT,
    address TEXT
}
CREATE TABLE studios {
    ID SERIAL PRIMARY KEY,
    name TEXT,
    address TEXT
}
CREATE TABLE starsin {
    ID SERIAL,
    movie INTEGER REFERENCES movies (ID) NOT NULL,
    star INTEGER REFERENCES stars (ID) NOT NULL
}
CREATE TABLE ownership {
    ID SERIAL,
    movie INTEGER REFERENCES movies (ID) NOT NULL,
    owner INTEGER REFERENCES studios (ID) NOT NULL
}

Example 2b: many-to-one relationship

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

The movie-studio relationship is a many-to-one relationship.
"Movie" is the "many" side.
"Studio" is the "one" side.
E/R\rightarrow Relational: Basic Translation revisited for many-to-one relationship

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

```sql
CREATE TABLE movies (  
    ID SERIAL PRIMARY KEY,  
    title TEXT,  
    year INTEGER,  
    length INTEGER,  
    owner INTEGER REFERENCES studios (ID)
)

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

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

CREATE TABLE starsin (  
    ID SERIAL,  
    movie INTEGER REFERENCES movies (ID) NOT NULL,  
    star INTEGER REFERENCES stars (ID) NOT NULL
)```
Example 2c: Constraints: uniqueness; required attributes

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

Example 2d: Constraints: Required relationship; cardinality ranges

In addition to Example 2c’s assumptions, let us also assume that:
- a movie is owned by exactly one studio
- so far we had not assumed that the owning studio has to be known (not null)
- a movie must have at least one actor and no more than 100
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>star</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ID</th>
<th>title</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 II: Introduce such constraints on your E/R, despite the fact that you will not be able to translate them to the SQL schema
Vice versa: SQL allows some constraints that are not in plain E/R

Notable cases:

- **Attribute value ranges**
  - Example: Declare that the year of movies is after 1900

- **Multi-attribute UNIQUE**
  - Example: Declare that the (title, year) attribute value combination is unique

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

---

**Added constraints of previous slide to schema of Example 2d**

```sql
CREATE TABLE movies (
    ID SERIAL PRIMARY KEY,
    title TEXT NOT NULL,
    year INTEGER NOT NULL CHECK (year > 1900),
    length INTEGER NOT NULL,
    owner INTEGER REFERENCES studios (ID) NOT NULL,
    UNIQUE (title, year)
)

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

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

CREATE TABLE starsin (
    ID SERIAL,
    movie INTEGER REFERENCES movies (ID) NOT NULL,
    star INTEGER REFERENCES stars (ID) NOT NULL
)
```
Example 3: one-to-one relationships

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

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
2nd candidate

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

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

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

Example 4: 3-Way Relationship

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

Example 5a: Self-Relationships with Roles

```
CREATE TABLE contract (  
  ID SERIAL,  
  movie INTEGER REFERENCES movies (ID) NOT NULL,  
  star INTEGER REFERENCES stars (ID) NOT NULL,  
  owner INTEGER REFERENCES studios (ID) NOT NULL,  
  fee INTEGER
)
```

---

**Example 5a: Self-Relationships with Roles**

```
CREATE TABLE contract (  
  ID SERIAL,  
  movie INTEGER REFERENCES movies (ID) NOT NULL,  
  star INTEGER REFERENCES stars (ID) NOT NULL,  
  owner INTEGER REFERENCES studios (ID) NOT NULL,  
  fee INTEGER
)
```

---

**Example 5a: Self-Relationships with Roles**

```
CREATE TABLE contract (  
  ID SERIAL,  
  movie INTEGER REFERENCES movies (ID) NOT NULL,  
  star INTEGER REFERENCES stars (ID) NOT NULL,  
  owner INTEGER REFERENCES studios (ID) NOT NULL,  
  fee INTEGER
)
```

---

**Example 5a: Self-Relationships with Roles**

```
CREATE TABLE 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 movies (  
  ID SERIAL PRIMARY KEY,
  -- 
)

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

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

Example 5b: Combo: One-to-one Self-Relationship

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

Modeling movie sequels by “DirectSequelOf” is preferable to using “SequelOf” of previous slide

A lesson about database design:
• Good designs avoid redundancy.
• No stored piece of data should be inferable from other stored pieces of data
Example 6: Subclassing

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,
    ...
)
```

+ captures constraints
- Information about graduates is spread on two tables
- Creating a report about students is a tricky query
To appreciate the above wait till we discuss SQL
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,
    ...
)

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

Writing programs on databases: JDBC

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

[Diagram showing the relationship between App Server, Web Application, RDBMS, JDBC Client, pgAdmin (desktop client), Relational Database, JDBC Server, SQL commands, Relations, cursors, other...]

37

38
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 always a table (regardless of the query language used)

- Internal Equivalent of SQL: Relational Algebra
  - used internally by the database system
  - **procedural** (operational): we describe **how** we retrieve

- CSE132A, CSE232A

- The solutions to the following examples are on the class page download

### SQL Queries: The Basic From

- Basic form
  
  ```sql
  SELECT A_1, ..., A_N
  FROM R_1, ..., R_M
  WHERE <condition>
  WHERE clause is optional
  ```

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

- Example queries:
  
  - Find first names and last names of all students
    
    ```sql
    SELECT first_name, last_name
    FROM students;
    ```
  
  - Find all students whose first name is John; project all attributes
    
    ```sql
    SELECT *
    FROM students
    WHERE first_name = 'John';
    ```
  
  - 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
    
    ```sql
    ```
Attention: Combining data is the most important & different aspect of SQL

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

SQL Queries: Nesting

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

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

- Also
  - EXISTS <query>
  - NOT EXISTS <query>

Find the CSE135 students who take a Friday 11:00am class
SQL Queries, advanced: Aliases

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

```
SELECT c_others.name, first_name, last_name
FROM classes AS c_135, enrollment AS e_135,
     students,
     enrollment AS e_others, classes AS c_others
WHERE c_135.number = 'CSE135'
  AND c_135.id = e_135.class
  AND e_135.student = students.id
  AND students.id = e_others.student
  AND e_others.class = c_others.id
  AND NOT (c_others.number = 'CSE135')
```
Second, show just the other classes. Notice use of DISTINCT

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

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

- New construct in FROM clause
- R LEFT OUTER JOIN S ON R.<attr of R>=S.<attr of J>
- 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>RJ</td>
</tr>
<tr>
<td>RV</td>
<td>SJ</td>
</tr>
<tr>
<td>RV1</td>
<td>1</td>
</tr>
<tr>
<td>RV2</td>
<td>Null</td>
</tr>
</tbody>
</table>

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>SV1</td>
</tr>
<tr>
<td>2</td>
<td>Null</td>
<td>Null</td>
</tr>
<tr>
<td>Null</td>
<td>3</td>
<td>SV3</td>
</tr>
</tbody>
</table>

An application of outerjoin

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

```
SELECT students.id, first_name, last_name, SUM(credits)
FROM students LEFT OUTER JOIN enrollment ON students.id = enrollment.student
GROUP BY students.id, first_name, last_name
```
SQL: More Bells and Whistles ...

- Pattern matching conditions
  - `<attr>` LIKE `<pattern>`

  Retrieve all students whose name contains “Sm”
  
  ```sql
  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

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
SQL as a Data Manipulation Language: Insertions

- Inserting tuples
  ```sql
  INSERT INTO R (A_1, ..., A_k) 
  VALUES (v_1, ..., v_k);
  ```
- Some values may be left NULL
- Use results of queries for insertion
  ```sql
  INSERT INTO R 
  SELECT ... 
  FROM ... 
  WHERE ... 
  ```

- Insert in Students ‘John Doe’ with A# 99999999
- Enroll all CSE135 students into CSE132A

SQL as a Data Manipulation Language: Updates and Deletions

- Deletion basic form: delete every tuple that satisfies `<cond>`:
  ```sql
  DELETE FROM R 
  WHERE <cond>
  ```
- Update basic form: update every tuple that satisfies `<cond>` in the way specified by the `SET` clause:
  ```sql
  UPDATE R 
  SET A_1=<exp_1>, ..., A_k=<exp_k> 
  WHERE<cond>
  ```

- Delete “John” “Smith”
- Update the registered credits of all CSE135 students to 5