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

- Persistent data structure
  - Large volume of data
  - “Independent” from processes using the data

- High-level API 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
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
Example Problem:
- Represent the students and Winter classes of the CSE department, including the list of students who take each class.
- Students have A#, 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:...
E/R → Relational Schema: Basic Translation

- For every entity, create corresponding table
  - Include an ID attribute even if not in E/R
- For every relationship, create table
  - For each referenced entity $E_i$ include foreign key attribute referencing ID of $E_i$

Relational schema, as produced by algorithm of previous page

<table>
<thead>
<tr>
<th>Classes</th>
<th>id</th>
<th>name</th>
<th>number</th>
<th>date_code</th>
<th>start_time</th>
<th>end_time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></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></td>
<td>Databases</td>
<td>CSE132A</td>
<td>TuTh</td>
<td>3:30</td>
<td>4:50</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>VLSI</td>
<td>CSE121</td>
<td>F</td>
<td>11:00</td>
<td>12:00</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>88888888</td>
<td>John</td>
<td>Smith</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>11111111</td>
<td>Mary</td>
<td>Doe</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>22222222</td>
<td>Jay</td>
<td>Chen</td>
<td></td>
</tr>
</tbody>
</table>
Example 2

Movies have a title, a year of release and length (in minutes). Zero or more actors may appear in a movie. We want to know their names and address. A movie is owned by one or more studios. We want to know the name and address of studios.

3-Way Relationship: Example 3

- A studio has contracted with a particular star to act in a particular movie
  - No ownership of movies by studios
Relationships with Roles: Example 4

Movies

Title

Year

Length

SequelOf

Original

Sequel

“Subclassing”

Movies

Year

Title

Length

StarsIn

Name

Address

Cartoons

Voices

IsA

Stars
### 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 TEXT,  
    first_name TEXT,  
    last_name TEXT  
)

CREATE TABLE enrollment (  
    id SERIAL PRIMARY KEY,  
    class INTEGER REFERENCES classes (id),  
    student INTEGER REFERENCES students (id),  
    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.

---

### ... and now some hands-on experience

- Install the Postgresql open source database
- For educational and management purposes use the pgAdmin client
  - 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
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 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
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`

Find first names and last names of all students

Find all students whose first name is John; project all attributes

Find the students registered for CSE135

---

SQL Queries: Aliases

• Use the same relation more than once in the FROM clause
• Tuple variables
• **Problem:** Find the classes taken by students who take CSE135
  - First, also showing the students
  - Second, just the classes. Notice use of DISTINCT.
## 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

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

Example: Find the average salary of all employees:

```
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:

```
SELECT Dept, AVG(Salary) AS AvgSal
FROM Employee
GROUP BY Dept
HAVING COUNT(Name) > 1
```
Aggregation Can Involve Many Tables

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

SQL: More Bells and Whistles ...

- Select all attributes using *
- Pattern matching conditions
  - `<attr> LIKE <pattern>`

Retrieve all student attributes of currently enrolled students

Retrieve all students whose name contains “Sm”

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

- **Duplicate elimination** must be explicitly requested
  
  ```sql
  SELECT DISTINCT ...
  FROM ...
  WHERE ...
  ```

- **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>
<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>
### SQL as a Data Manipulation Language: Insertions

- Inserting tuples
  
  ```sql
  INSERT INTO R(A₁,…,Aₖ)
  VALUES (v₁,…,vₖ);
  ```

- 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 A₁=<exp₁>,…,Aₖ=<expₖ>
  WHERE <cond>
  ```

+ Delete “John” “Smith”

+ Update the registered credits of all CSE135 students to 5