SQL
“Structured Query Language”

• Standard for relational db systems
• History:
  Developed at IBM in late 70s
  First standard: SQL-86
  Second standard: SQL-92
  Third standard: SQL-99 or SQL3, well over 1000 pages!
  “The nice things about standards is that you have so many to choose from” -- Andres S. Tannenbaum
SQL: Data Definition Language

Create table

- Syntax:
  create table <name> (<att_1>  type_1 , <att_2>  type_2 , ..., <att_n>  type_n)

- Example
  CREATE TABLE movies (title char(20), director char(10), actor char(10))
  CREATE TABLE schedule (theater char(10), title char(20))

Delete table

- Syntax
  DROP TABLE <NAME>

- Example
  DROP TABLE schedule
Other DDL commands

• Add a new attribute to an existing table
possible to initialize with default value: otherwise null

    ALTER TABLE schedule ADD COLUMN  time int  DEFAULT 0

• Drop attribute from a table
tuples are truncated: “projection”

• Define constraints on tables: keys, foreign keys,…
will see later
### SQL Queries: The Basic From

<table>
<thead>
<tr>
<th>Basic form</th>
<th>WHERE clause is optional</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SELECT</strong> ( a_1, \ldots, a_N )** FROM** ( R_1, \ldots, R_M )** WHERE** condition</td>
<td>When more than one relation of the FROM has an attribute named ( A ), we refer to a specific ( A ) attribute as (&lt;\text{RelationName}&gt;.A)</td>
</tr>
</tbody>
</table>

#### Examples

- **Find titles of currently playing movies**
  
  ```sql
  SELECT Title
  FROM Schedule
  ```

- **Find the titles of all movies by “Berto”**
  
  ```sql
  SELECT Title
  FROM Movie
  WHERE Director="Berto"
  ```

- **Find the titles and the directors of all currently playing movies**
  
  ```sql
  SELECT Movie.Title, Director
  FROM Movie, Schedule
  WHERE Movie.Title=Schedule.Title
  ```
SQL Queries: Tuple variables

• Use the same relation more than once in the FROM clause
• Example: find actors who are also directors

SELECT t.Actor
FROM Movie t, Movie s
WHERE t.Actor = s.Director
SQL Queries: Nesting

- The WHERE clause can contain predicates of the form
  - attr/value IN <SQL query>
  - attr/value NOT IN <SQL query>
- The IN predicate is satisfied if the attr or value appears in the result of the nested <SQL query>

**Examples:**

*find directors of current movies*

SELECT director FROM Movie
WHERE title IN
  (SELECT title
   FROM schedule)

The nested query finds currently playing movies
More examples

Find actors playing in some movie by Bertolucci

```
SELECT actor FROM Movie
WHERE title IN
    (SELECT title
     FROM Movie
     WHERE director = "Bertolucci")
```

The nested query finds the titles of movies by Bertolucci

Queries involving nesting but no negation can always be un-nested, unlike queries with nesting and negation
Typical use: “find objects that always satisfy property X”, e.g.,
find actors playing in every movie by “Berto”

SQL’s way of saying this:

find the actors for which there is no movie by Bertolucci in which they do not act

OR equivalently:

find the actors not among the actors for which there is some movie by Bertolucci in which they do not act

SELECT Actor FROM Movie
WHERE Actor NOT IN
(SELECT m2.Actor
FROM Movie m1, Movie m2,
WHERE m1.Director="Berto"
AND m2.Actor NOT IN
(SELECT Actor
FROM Movie
WHERE Title=m1.Title))

The shaded query finds actors for which there is some movie by “Berto” in which they do not act

The top lines complement the shaded part
SQL: Union, Intersection, Difference

- **Union**
  - \(<SQL \text{ query } 1>\) UNION \(<SQL \text{ query } 2>\)

- **Intersection**
  - \(<SQL \text{ query } 1>\) INTERSECT \(<SQL \text{ query } 2>\)

- **Difference**
  - \(<SQL \text{ query } 1>\) MINUS \(<SQL \text{ query } 2>\)

---

**Find all actors or directors**

(SELECT Actor
FROM Movie)
UNION
(SELECT Director
FROM Movie)

---

**Find all actors who are not directors**

(SELECT Actor
FROM Movie)
MINUS
(SELECT Director
FROM Movie)
Nested Queries: Existential and Universal Quantification

- $A \text{ op } \text{ANY} <\text{nested query}>$ is satisfied if **there is** a value $X$ in the result of the $<\text{nested query}>$ and the condition $A \text{ op } X$ is satisfied
  - ANY aka SOME

- $A \text{ op } \text{ALL} <\text{nested query}>$ is satisfied if **for every** value $X$ in the result of the $<\text{nested query}>$ the condition $A \text{ op } X$ is satisfied

**Find directors of currently playing movies**
SELECT Director
FROM Movie
WHERE Title = ANY
  SELECT Title
  FROM Schedule

**Find the employees with the highest salary**
SELECT Name
FROM Employee
WHERE Salary >= ALL
  SELECT Salary
  FROM Employee
Find actors playing in every movie by “Bertolucci”

SELECT m1.Actor
FROM Movie m1
WHERE
    (SELECT Title
     FROM Movie
     WHERE Actor = m1.Actor)
CONTAINS
    (SELECT Title
     FROM Movie
     WHERE Director = "Berto")
CONTAINS
    (SELECT Title
     FROM Movie
     WHERE Director = "Berto")

- <nested query 1> CONTAINS
  <nested query 2>
Views

- Create permanent or temporary tables holding result of a query

- **Syntax:**
  
  ```sql
  CREATE VIEW <TABLE> AS <query>
  ```

- Once defined, views can be used in queries like any other relation

- Their content is automatically updated when database changes

  ```sql
  CREATE VIEW Berto-movies (movie, actor) AS
  SELECT title, actor
  FROM movie
  WHERE director = "Bertolucci"
  ```

  ```sql
  SELECT movie
  FROM Berto-movies
  WHERE actor = "Winger"
  ```
Views can simplify nested queries

Example

*find actors playing in every movie by “Berto”:*

```sql
SELECT Actor FROM Movie
WHERE Actor NOT IN
  (SELECT m2.Actor
   FROM Movie m1, Movie m2,
   WHERE m1.Director="Berto"
   AND m2.Actor NOT IN
   (SELECT Actor
    FROM Movie
    WHERE Title=m1.Title))
```

The shaded query finds actors NOT playing in some movie by “Berto”
Same query using views

CREATE VIEW Berto-Movies AS
SELECT title FROM Movie WHERE director = “Bertoucci” ;

CREATE VIEW Not-All-Berto AS
SELECT m.actor FROM Movies m, Berto-Movies
WHERE Berto-Movies.title NOT IN
  (SELECT title FROM Movies
   WHERE actor = m.actor);

CREATE VIEW Answer AS
SELECT actor FROM Movies WHERE actor NOT IN
  (SELECT * FROM Not-All-Berto)
SQL Queries: Aggregation and Grouping

- Aggregate functions: AVG, COUNT, MIN, MAX, SUM, ...
  (user defined functions)
- Group-by

**Employee**

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

Find 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>

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** clause

Find the average salary of for each department that has more than 1 employee

```
SELECT Dept, AvgSal= Avg(Salary)
FROM Employee
GROUP-BY Dept
HAVING COUNT(Name)>1
```

For each movie having at least 100 actors, find the number of theaters showing the movie

```
SELECT m.Title,
    COUNT(s.Theater) AS number
FROM Schedule s, Movie m
WHERE s.Title = m.Title
GROUP BY m.Title
HAVING COUNT(DISTINCT m.Actor) > 100
```

Aggregate is taken over pairs <s,m> with same Title
SQL: More Bells and Whistles ...

- Select all attributes using *

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

Retrieve all movie attributes of currently playing movies
SELECT Movie.*
FROM Movie, Schedule
WHERE Movie.Title=Schedule.Title

Retrieve all movies where the title starts with “Ta”
SELECT *
FROM Movie
WHERE Title LIKE “Ta%”

Forgot if “Polanski” is spelled with “i” or “y”:
SELECT *
FROM Movie
WHERE Director LIKE “Polansk_”
…and a Few “Dirty” Points

- *Duplicate elimination* must be explicitly requested
  - `SELECT DISTINCT … FROM … WHERE …`
- *Null values*
  - all comparisons involving NULL are *unknown* by definition
  - all aggregation operations, except *count*, ignore NULL values

```
SELECT Title
FROM Movie
```

```
<table>
<thead>
<tr>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tango</td>
</tr>
<tr>
<td>Tango</td>
</tr>
<tr>
<td>Tango</td>
</tr>
</tbody>
</table>
```

```
SELECT DISTINCT Title
FROM Movie
```

```
<table>
<thead>
<tr>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tango</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>Title</th>
<th>Director</th>
<th>Actor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wild</td>
<td>Lynch</td>
<td>Winger</td>
</tr>
<tr>
<td>Sky</td>
<td>Berto</td>
<td>Winger</td>
</tr>
<tr>
<td>Reds</td>
<td>NULL</td>
<td>Beatty</td>
</tr>
<tr>
<td>Tango</td>
<td>Berto</td>
<td>Brando</td>
</tr>
<tr>
<td>Tango</td>
<td>Berto</td>
<td>Winger</td>
</tr>
<tr>
<td>Tango</td>
<td>Berto</td>
<td>NULL</td>
</tr>
</tbody>
</table>
```
SQL as a Data Manipulation Language: Insertions

- inserting tuples
  - `INSERT INTO R VALUES (v1, ..., vk);`
- some values may be left NULL
- use results of queries for insertion
  - `INSERT INTO R SELECT ... FROM ... WHERE`
SQL as a Data Manipulation Language: Updates and Deletions

• **Deletion** basic form: delete every tuple that satisfies \(<cond>\)
  – DELETE FROM \(R\) WHERE \(<cond>\)

• **Update** basic form: update every tuple that satisfies \(<cond>\) in the way specified by the SET clause
  – UPDATE \(R\)
    SET \(A1=\text{<exp1>},\)
    ...\n    \(Ak=\text{<expk>}\)
    WHERE \(<cond>\)

*Delete the movies that are not currently playing*
DELETE FROM Movie
WHERE Title NOT IN (SELECT Title
FROM Schedule)

*Change all “Berto” entries to “Bertolucci”*
UPDATE Movie
SET Director=“Bertolucci”
WHERE Director=“Berto”

*Increase all salaries in the Toys dept by 10%*
UPDATE Employee
SET Salary = 1.1 * Salary
WHERE Dept = “Toys”

The “rich get richer” exercise:
Increase by 10% the salary of the employee with the highest salary
QBE

• Query-By-Example
  – provides a visual interface for queries and updates

• Examples: movie database queries
  – query 1

<table>
<thead>
<tr>
<th>schedule</th>
<th>theater</th>
<th>title</th>
</tr>
</thead>
<tbody>
<tr>
<td>P.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

  • P. : “print value”

  – query 2

<table>
<thead>
<tr>
<th>movie</th>
<th>title</th>
<th>director</th>
<th>actor</th>
</tr>
</thead>
<tbody>
<tr>
<td>P.</td>
<td>Berto</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
QBE (2)

- query 3

<table>
<thead>
<tr>
<th>movie</th>
<th>title</th>
<th>director</th>
<th>actor</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>_t</td>
<td>_d</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>schedule</th>
<th>theater</th>
<th>title</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>_t</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>answer</th>
<th>title</th>
<th>director</th>
</tr>
</thead>
<tbody>
<tr>
<td>I.</td>
<td>_t</td>
<td>_d</td>
</tr>
</tbody>
</table>

- Note:
  - answer table explicitly specified
  - underscore _x means _x can take any value, like a variable
  - I. means insert
QBE (3)

- “Find all actors playing in every movie by Berto”
  - requires multi-stage query, creating intermediate answers
  - analog of nested queries in SQL
- I stage:

<table>
<thead>
<tr>
<th>schedule</th>
<th>title</th>
<th>director</th>
<th>actor temp</th>
<th>actor</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>_a</td>
<td>I</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>_t Berto</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>_t</td>
<td>_a</td>
</tr>
</tbody>
</table>

- Semantics of \( \neg \)
  - for \(_t\) and \(_a\) fixed, satisfying positive part of pattern, there is no tuple occurring with \(_t\) and \(_a\) as in the negated tuple
QBE (4)

• II stage
  – (complement of temp computed in stage I)

<table>
<thead>
<tr>
<th>movie</th>
<th>title</th>
<th>director</th>
<th>actor</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>_a</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>temp</th>
<th>actor</th>
</tr>
</thead>
<tbody>
<tr>
<td>¬_a</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>answer</th>
<th>actor</th>
</tr>
</thead>
<tbody>
<tr>
<td>I._a</td>
<td></td>
</tr>
</tbody>
</table>
Updates in QBE

• Deletions: similar to inserts
  – D.
  – “Delete all movies by Berto”:
    | movie | title | director | actor |
    |-------|-------|----------|-------|
    | D.    |       | Berto    |       |
  – “Delete all movies by directors who are also actors”:
    | movie | title | director | actor |
    |-------|-------|----------|-------|
    | D.    |       | _d       |       |
    |       |       | _d       | _d   |
Updates in QBE (2)

- Updates: using **key** attributes
  - **key**: set of attributes which uniquely identify the tuple
  - keys are explicitly declared

  - “Sally gets a 5% salary raise”

<table>
<thead>
<tr>
<th>employee</th>
<th>name</th>
<th>salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>U.</td>
<td>Sally</td>
<td>_x * 1.05</td>
</tr>
<tr>
<td>Sally</td>
<td>_x</td>
<td></td>
</tr>
<tr>
<td>Sally</td>
<td>_x</td>
<td></td>
</tr>
</tbody>
</table>
Updates in QBE (3)

• “All employees who make less than 2000 receive a 5% raise”

<table>
<thead>
<tr>
<th>employee</th>
<th>name</th>
<th>salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>U.</td>
<td>_u</td>
<td>_x * 1.05</td>
</tr>
<tr>
<td></td>
<td>_u</td>
<td>_x</td>
</tr>
</tbody>
</table>

Condition box

• Note: QBE allows explicit specification of conditions using condition boxes