SQL Data Manipulation Language

- primarily declarative query language
- starting point: relational calculus
  aka first-order predicate logic
- many additions, bells and whistles…
- corresponding procedural language: relational algebra

will discuss relational calculus and algebra later

Running example: Movie database

<table>
<thead>
<tr>
<th>Movie</th>
<th>Title</th>
<th>Director</th>
<th>Actor</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></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></td>
</tr>
</tbody>
</table>
## Running example: Movie database

<table>
<thead>
<tr>
<th>Movie</th>
<th>Title</th>
<th>Director</th>
<th>Actor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Star Wars</td>
<td>Lucas</td>
<td>Ford</td>
<td></td>
</tr>
<tr>
<td>Star Wars</td>
<td>Lucas</td>
<td>Fischer</td>
<td></td>
</tr>
<tr>
<td>Mad Max</td>
<td>Miller</td>
<td>Hardy</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Schedule</th>
<th>Theater</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hillcrest</td>
<td>Star Wars</td>
<td></td>
</tr>
<tr>
<td>Hillcrest</td>
<td>Mad Max</td>
<td></td>
</tr>
<tr>
<td>Paloma</td>
<td>Rocky Horror</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Find titles of currently playing movies**

```
SELECT Title
FROM Schedule
```

**Find the titles of all movies by “Lucas”**

```
SELECT Title
FROM Movie
WHERE Director=“Lucas”
```

**Find the titles and the directors of all currently playing movies**

```
SELECT Movie.Title, Director
FROM Movie, Schedule
WHERE Movie.Title=Schedule.Title
```
Basic form

```
SELECT a_1, ..., a_n
FROM R_1, ..., R_m
WHERE condition
```

WHERE clause is optional

Informal semantics of basic form

```
for each tuple t_1 in R_1
for each tuple t_2 in R_2
......
for each tuple t_m in R_m
if condition(t_1, t_2, ..., t_m) then output in answer
attributes a_1, ..., a_n of t_1, ..., t_m
```
Examples revisited

SELECT Title
FROM Movie
WHERE Director= “Lucas”

Semantics:

for each tuple m in Movie
if m(Director) = “Lucas” then output m(Title)

Examples revisited

SELECT Movie.Title, Director
FROM Movie, Schedule
WHERE Movie.Title=Schedule.Title

Semantics:

for each tuple m in Movie
for each tuple s in Schedule
if m(title) = s(title) then output <m(Title),m(Director)>
SQL Queries: Tuple variables

- Needed when using the same relation more than once in the FROM clause
- Example: find actors who are also directors

```sql
SELECT t.Actor
FROM Movie t, Movie s
WHERE t.Actor = s.Director
```

Semantics:
for each t in Movie
for each s in Movie
if \( t(\text{Actor}) = s(\text{Director}) \) then output \( t(\text{Actor}) \)

Previous examples using tuple variables

```sql
SELECT Title
FROM Movie
WHERE Director = "Lucas"
```

```sql
SELECT m.Title
FROM Movie m
WHERE m.Director = "Lucas"
```
Previous examples using tuple variables

SELECT Movie.Title, Director
FROM Movie, Schedule
WHERE Movie.Title=Schedule.Title

SELECT m.Title, m.Director
FROM Movie m, Schedule s
WHERE m.Title = s.Title

SQL Queries: * and LIKE

• 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_”
SQL Queries: duplicate elimination

- Default: answers to queries contain duplicates
  SELECT Title
  FROM Movie
  SELECT DISTINCT Title
  FROM Movie

- Duplicate elimination must be explicitly requested
  - SELECT DISTINCT …
    FROM … WHERE …

- Test for uniqueness:
  UNIQUE <SQL query>
  NOT UNIQUE <SQL query>

Ordering the Display of Tuples

- List all titles and actors of movies by Fellini, in alphabetical order of titles
  ```sql
  select Title, Actor
  from Movie
  where Director = ‘Fellini’
  ORDER BY Title
  ```

- We may specify desc for descending order or asc for ascending order, for each attribute;
  - ascending order is the default.
  - Example: order by Title desc
Renaming attributes in result

Done using the `as` construct:

Find titles of movies by Bertolucci, under attribute Berto-titles:

```sql
select title as Berto-title
from movie
where director = 'Bertolucci'
```

---

Aggregate Functions

These functions operate on the multiset of values of a column of a relation, and return a value:

- `avg`: average value
- `min`: minimum value
- `max`: maximum value
- `sum`: sum of values
- `count`: number of values
Aggregate Functions (Cont.)

-- Find the average account balance at the La Jolla branch.

```
select avg (balance)
from account
where branch_name = 'La Jolla'
```

-- Find the number of depositors in the `customer` relation.

```
select count (*)
from customer
```

-- Find the number of depositors in the bank.

```
select count (distinct customer_name)
from depositor
```

Aggregate Functions (Cont.)

- Find the maximum salary, the minimum salary, and the average salary among all employees for the Company database

```
SELECT MAX(SALARY),
    MIN(SALARY), AVG(SALARY)
FROM EMPLOYEE
```

Obs. Some SQL implementations may not allow more than one aggregate function in the SELECT-clause!
Aggregate Functions (Cont.)

- Find the maximum salary, the minimum salary, and the average salary among employees who work for the 'Research' department.

SELECT MAX(SALARY), MIN(SALARY), AVG(SALARY)
FROM EMPLOYEE, DEPARTMENT
WHERE DNO=DNUMBER AND
  DNAME='Research'

Note: the aggregate functions are applied to the relation consisting of all pairs of tuples from Employee and Department satisfying the condition in the WHERE clause.
### Grouping (example)

<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**

```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>

**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>
Grouping

• Allows to apply the aggregate functions to subgroups of tuples in a relation
• Each subgroup of tuples consists of the set of tuples that have the same value for the grouping attribute(s)
• The function is applied to each subgroup independently
• SQL has a GROUP BY-clause for specifying the grouping attributes, which must also appear in the SELECT-clause

Grouping (cont.)

• For each department, retrieve the department number, the number of employees in the department, and their average salary.

```
SELECT DNO, COUNT(*) AS NUMEMP, AVG(SALARY) AS AVGSAL
FROM EMPLOYEE
GROUP BY DNO
```

– The EMPLOYEE tuples are divided into groups--each group having the same value for the grouping attribute DNO
– The COUNT and AVG functions are applied to each such group of tuples separately
– The SELECT-clause includes only the grouping attribute and the aggregate functions to be applied on each group of tuples
GROUPING Example

- For each project, retrieve the project number, project name, and the number of employees who work on that project.

```
SELECT PNUMBER, PNAME, COUNT (*)
FROM PROJECT, WORKS_ON
WHERE PNUMBER=PNO
GROUP BY PNUMBER, PNAME
```

- Note: the grouping and functions are applied on pairs of tuples from PROJECT, WORKS_ON
SELECT PNUMBER, PNAME, COUNT (DISTINCT ESSN) FROM PROJECT, WORKS_ON WHERE PNUMBER=PNO GROUP BY PNUMBER, PNAME

Subtlety: suppose PNO and ESSN do not form a key for WORKS_ON
Problem: will get duplicate employees

<table>
<thead>
<tr>
<th>Works_on</th>
<th>ESSN</th>
<th>PNO</th>
<th>HOURS</th>
<th>PROJECT</th>
<th>PNAME, PNUMBER</th>
</tr>
</thead>
<tbody>
<tr>
<td>111-11-1111</td>
<td>001</td>
<td>20</td>
<td></td>
<td>Wiki</td>
<td>001</td>
</tr>
<tr>
<td>111-11-1111</td>
<td>001</td>
<td>10</td>
<td></td>
<td>Geo</td>
<td>002</td>
</tr>
<tr>
<td>22-22-2222</td>
<td>002</td>
<td>25</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Fix:

THE HAVING-CLAUSE

- Sometimes we want to retrieve the values of aggregate functions for only those groups that satisfy certain conditions
- The HAVING-clause is used for specifying a selection condition on groups (rather than on individual tuples!)
Aggregate Functions – Having Clause

• Find the names of all branches where the average account balance is more than $1,200.

```sql
select branch_name, avg(balance)
from account
group by branch_name
HAVING avg(balance) > 1200
```

Condition in HAVING clause use values of attributes in group-by clause and aggregate functions on the other attributes

THE HAVING-CLAUSE (cont.)

• For each project on which more than two employees work, retrieve the project number, project name, and the number of employees who work on that project.

```sql
SELECT PNUMBER, PNAME, COUNT(*)
FROM PROJECT, WORKS_ON
WHERE PNUMBER=PNO
GROUP BY PNUMBER, PNAME
HAVING COUNT(*) > 2
```

Note: predicates in the having clause are applied after the formation of groups whereas predicates in the where clause are applied before forming groups
Another example

*For each movie having more than 100 actors, find the number of theaters showing the movie*

```sql
SELECT m.Title, COUNT(DISTINCT 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

<table>
<thead>
<tr>
<th>Schedule</th>
<th>Theater</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hillcrest</td>
<td>Star Wars</td>
<td></td>
</tr>
<tr>
<td>Paloma</td>
<td>Star Wars</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Movie</th>
<th>Title</th>
<th>Director</th>
<th>Actor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Star Wars</td>
<td>Lucas</td>
<td>Ford</td>
<td></td>
</tr>
<tr>
<td>Star Wars</td>
<td>Lucas</td>
<td>Ford</td>
<td></td>
</tr>
</tbody>
</table>

FROM Schedule s, Movie m
WHERE s.Title = m.Title

<table>
<thead>
<tr>
<th>Theater</th>
<th>Title</th>
<th>Director</th>
<th>Actor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hillcrest</td>
<td>Star Wars</td>
<td>Lucas</td>
<td>Ford</td>
</tr>
<tr>
<td>Paloma</td>
<td>Star Wars</td>
<td>Lucas</td>
<td>Ford</td>
</tr>
<tr>
<td>Hillcrest</td>
<td>Star Wars</td>
<td>Lucas</td>
<td>Fischer</td>
</tr>
<tr>
<td>Paloma</td>
<td>Star Wars</td>
<td>Lucas</td>
<td>Fischer</td>
</tr>
</tbody>
</table>

GROUP BY m.Title

<table>
<thead>
<tr>
<th>Title</th>
<th>Theater</th>
<th>Director</th>
<th>Actor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Star Wars</td>
<td>Hillcrest</td>
<td>Lucas</td>
<td>Ford</td>
</tr>
<tr>
<td>Paloma</td>
<td>Lucas</td>
<td>Ford</td>
<td></td>
</tr>
<tr>
<td>Hillcrest</td>
<td>Lucas</td>
<td>Fischer</td>
<td></td>
</tr>
<tr>
<td>Paloma</td>
<td>Lucas</td>
<td>Fischer</td>
<td></td>
</tr>
</tbody>
</table>
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**

```sql
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*

```sql
SELECT actor FROM Movie
WHERE title IN (SELECT title FROM Movie
WHERE director = “Bertolucci”)
```

The nested query finds the titles of movies by Bertolucci
In this case, can eliminate nesting:

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

```
SELECT m1.actor
FROM Movie m1, Movie m2
WHERE m1.title = m2.title
  and m2.director = "Bertolucci"
```

Question: is nesting syntactic sugar? Can it always be eliminated?

A: yes   B: no
Question: is nesting syntactic sugar? Can it always be eliminated?
A: yes  B: no

Queries involving nesting but no negation can always be un-nested, unlike queries with nesting and negation

Correlated nested queries
• If a condition in the WHERE-clause of a nested query references an attribute of a relation declared in the outer query, the two queries are said to be correlated
• The result of a correlated nested query may be different for each tuple (or combination of tuples) of the relation(s) the outer query
• E.g. DB Company: Retrieve the name of each employee who has a dependent with the same first name as the employee.

```sql
SELECT E.FNAME, E.LNAME
FROM EMPLOYEE E
WHERE E.SSN IN
  (SELECT ESSN
   FROM DEPENDENT
   WHERE ESSN=E.SSN
   AND E.FNAME=DEPENDENT_NAME)
```
Correlated nested queries (cont.)

- Correlated queries using just the IN comparison operators can still be unnested
- For example, the previous query could be un-nested as follows:

```sql
SELECT E.FNAME, E.LNAME
FROM EMPLOYEE E, DEPENDENT D
WHERE E.SSN=D.ESSN AND
E.FNAME=D.DEPENDENT_NAME
```

Use of NOT IN tests increases expressive power
Simple use of NOT IN

*Find all movies in which Hitchcock does not act:*

```sql
SELECT title FROM Movie
WHERE title NOT IN
  (SELECT title FROM Movie
   WHERE actor = 'Hitchcock')
```

Simple use of NOT IN

*Find all movies that are not currently playing:*

```sql
SELECT title FROM Movie
WHERE title NOT IN
  (SELECT title FROM Schedule)
```
Why can’t these be flattened?

Hand-waving “proof”:
1. Basic queries with no nesting are monotonic
   
   the answer never decreases when the database increases
   
   DB1 ⊆ DB2 implies Query(DB1) ⊆ Query(DB2)

2. But queries using NOT IN are usually not monotonic

   SELECT title FROM Movie
   WHERE title NOT IN
   (SELECT title FROM Schedule)

   If Schedule increases, the answer may decrease

---

Recall semantics of basic queries:

SELECT a₁, ..., aₙ
FROM R₁, ..., Rₘ
WHERE condition

for each tuple t₁ in R₁
for each tuple t₂ in R₂
......
for each tuple tₘ in Rₘ
if condition(t₁,t₂,...,tₘ) then output in answer
attributes a₁,...,aₙ of t₁,...,tₘ

This is monotonic if condition has no nested queries
Monotonic (A) or non-monotonic (B) ?

1. Find the theaters showing some movie by Fellini
2. Find the theaters showing only movies by Fellini
3. Find the actors who are also directors
4. Find the actors playing in some movie showing at Paloma
5. Find the actors playing in every movie by Bertolucci

<table>
<thead>
<tr>
<th>Schedule</th>
<th>Theater</th>
<th>Title</th>
<th>Movie</th>
<th>Title</th>
<th>Director</th>
<th>Actor</th>
</tr>
</thead>
</table>

More complex use of NOT IN

*Find the names of employees with the maximum salary*

```
SELECT name FROM Employee
WHERE salary NOT IN
  (SELECT e.salary
   FROM Employee e, Employee f
   WHERE e.salary < f.salary)
```

Intuition: salary is maximum if it is not among salaries e.salary lower than some f.salary
More complex use of \texttt{NOT IN}:

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

\textit{Everybody likes}
Another construct used with nesting: **EXISTS**

```sql
SELECT …
FROM…
WHERE … EXISTS (<query>)
```

Meaning of EXISTS:

**EXISTS** (<query>) is true iff the result of <query> is not empty.
**NOT EXISTS** (<query>) is true iff the result of <query> is empty.

Examples:

<table>
<thead>
<tr>
<th>R</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>7</td>
</tr>
</tbody>
</table>

**EXISTS** (select A from R where A < 4)

**NOT EXISTS** (select A from R where A > 10)

**EXISTS** (select * from R where A < B)

**NOT EXISTS** (select * from R where A < B)
Example: Find titles of currently playing movies directed by Berto:

```
SELECT s.title
FROM schedule s
WHERE EXISTS (SELECT * FROM movie
               WHERE movie.title = s.title AND
               movie.director = 'Berto')
```

<table>
<thead>
<tr>
<th>Schedule</th>
<th>Theater</th>
<th>Title</th>
<th>Movie</th>
<th>Title</th>
<th>Director</th>
<th>Actor</th>
</tr>
</thead>
</table>

Example (Boolean predicate): Everybody likes Sara Lee

```
NOT EXISTS
(SELECT * FROM PERSON
WHERE NOT EXISTS
(SELECT * FROM LIKES
WHERE PERSON.name = LIKES.name
AND brand = 'Sara Lee')
```
Example: Find the actors playing in every movie by Berto

```
SELECT a.actor FROM movie a
WHERE NOT EXISTS
  (SELECT * FROM movie m
   WHERE m.director = 'Berto' AND
     NOT EXISTS
     (SELECT *
      FROM movie t
      WHERE m.title = t.title
        AND t.actor = a.actor))
```

<table>
<thead>
<tr>
<th>Movie</th>
<th>Title</th>
<th>Director</th>
<th>Actor</th>
</tr>
</thead>
</table>

Nested Queries: ANY and ALL

- \( A \ op \ \text{ANY} <\text{nested query}> \) is satisfied if there is a value \( X \) in the result of the \(<\text{nested query}>\) and the condition \( A \ op X \) is satisfied
  - ANY aka SOME
- \( A \ op \ \text{ALL} <\text{nested query}> \) is satisfied if for every value \( X \) in the result of the \(<\text{nested query}>\) the condition \( A \ op X \) 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
```
Nested Queries: Set Comparison

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

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”)
```

The original SQL as specified for SYSTEM R had a CONTAINS operator. This was dropped from the language, possibly because of the difficulty in implementing it efficiently.

Nested queries in FROM clause

SQL allows nested queries in the FROM clause

Find directors of movies showing in Hillcrest:

```
SELECT m.director
FROM movie m,
     (select title from schedule
      where theater = ‘Hillcrest’) t
WHERE m.title = t.title
```

Note: this is syntactic sugar and can be eliminated
SQL: Union, Intersection, Difference

- **Union**
  - `<SQL query 1> UNION <SQL query 2>`

- **Intersection**
  - `<SQL query 1> INTERSECT <SQL query 2>`

- **Difference**
  - `<SQL query 1> EXCEPT <SQL query 2>`

- **Find all actors or directors**
  (SELECT Actor as Name FROM Movie)
  UNION
  (SELECT Director as Name FROM Movie)

- **Find all actors who are not directors**
  (SELECT Actor as Name FROM Movie)
  EXCEPT
  (SELECT Director as Name FROM Movie)

**Example (union):**
for each title in movie, find the number of theaters showing that title

<table>
<thead>
<tr>
<th>schedule</th>
<th>theater title</th>
<th>movie</th>
<th>title</th>
<th>director</th>
<th>actor</th>
</tr>
</thead>
</table>

57