Structured Query Language I

Instructor: Lei Jiang

Slides provided by Ramona Truta

SQL - A little history

- Structured Query Language
- Developed originally at IBM in the late 70s
  - First standard: SQL-86
  - SQL-89 (minor changes)
  - Second standard: SQL-92 (SQL2)
  - Third standard: SQL:1999 (SQL3)
    - Added regular expression matching, recursive queries, triggers, non-scalar types and some object-oriented features.
  - Latest standard: SQL:2003
    - Introduced XML-related features, window functions, standardized sequences and columns with auto-generated values

Data Definition Language (DDL)

- Allows the specification of not only a set of relations but also information about each relation, including:
  - The schema for each relation.
  - The domain of values associated with each attribute.
  - Integrity constraints
  - The set of indices to be maintained for each relations.
  - Security and authorization information for each relation.
  - The physical storage structure of each relation on disk.

Schema Definition

- A *schema* is a collection of objects: domains, tables, indexes, assertions, views, privileges.
- A schema has a name and an owner (`authorization`).
- **Syntax:**

```
CREATE SCHEMA <Name> [AUTORIZATION <username>]
```

- **Example:**

```
CREATE SCHEMA my343;
```

Domain types in SQL

- `char(n)` – fixed length string of exactly `n` characters.
  Example: `'Polanski'`
- `varchar(n)` – variable length string of up to `n` characters.
  Example: `'Polanski'`.
- `bit(n)` – fixed length bit string of exactly `n` bits.
  Example: `B'0101'`, `X'C1'`
- `varying(n)` – variable length bit string of up to `n` bits.

Domain types in SQL cont'd

- Exact numeric domains:
  - `int`: signed integer (4 bytes)
  - `smallint`: signed integer (2 bytes)
  - `numeric(p,n)`: user-specified precision of p digits, with n digits to the right of decimal point
- Approximate numeric domains
  - based on floating point represention
  - `real, double`: floating point and double-precision floating point numbers, with machine-dependent precision
  - `float(n)`: floating point number, with user-specified precision of at least n digits.
- Null values are allowed in all the domain types.
- Casting between data types: `CAST (<attribute> AS <type>)`

Types cont'd: Dates and Times

- `date type`: keyword `DATE` followed by a date in an appropriate form, e.g. `DATE '2007-05-31'`
- `time` type: keyword `TIME` followed by a string representing time; SQL uses a 24-hour clock.
- `timestamp` type: combines `date` and `time`.
- Operations on these types:

  – they can be compared for equality, and for order. If for two dates d_1 and d_2 we have $d_1 < d_2$, then d_1 is earlier than d_2.

  – `YEAR/MONTH/DAY (d)`, where `d` is of type `DATE` returns the year/month/day corresponding to a certain date.

  – `CURRENT_DATE` returns the current date.

Table definition

- An SQL *table* consists of an ordered set of attributes, and (possibly empty) set of constraints.
- The `CREATE TABLE` statement defines a relation schema, creating an empty instance.
- **Syntax:**

```
CREATE TABLE <Name> (<attr1> <type> [DEFAULT <value>], ...,
                     <attrN> <type>[DEFAULT <value>])
```

- **Example**:

```
CREATE TABLE Movies (mID int,
                     title char(20),
                     director char(10),
                     year int default 0,
                     length real)
```

Schema Updates

- `DROP <SCHEMA | TABLE | VIEW| ASSERTION <name>`

  removes `<name>` from the database.

- `ALTER TABLE <name> ADD COLUMN <newcolumn> <type>`

  adds `<newcolumn>` to the table `<name>`

  – **Example**:

```
ALTER TABLE Movies ADD COLUMN budget real
```

- `ALTER TABLE <name> DROP COLUMN <columnName>`

  removes the `<columnName>` from the table `<name>`

  – **Example**:

```
ALTER TABLE Movies DROP COLUMN budget
```

Modification of the database: Insertion

- Inserting a new tuple into a table:

  `INSERT INTO <name> VALUES (...)`

- **Example**:

  `INSERT INTO Movies VALUES (123, 'Chinatown', 'Polanski', 1974, 131)`

- **Bulk insertions**:

  `INSERT INTO <name> (<query>)`

  – the attributes in the result of `<query>` have to be the same as those of `<name>`.

  – `<query>` is fully evaluated before any of its resulted tuples are inserted into `<name>`

Modification of the database: Insertion cont'd

- One can insert values to some attributes only:

  ```
  INSERT INTO <name>(<attributes>) (<query>)
  ```

- **Example**:

  ```
  INSERT INTO Roles(mID)
     (SELECT mID FROM Movies)
  ```

- When only values for some attributes are inserted, what are the values of the rest of the attributes ?
  - Answer: default values

Database modification: deletions

- General form:

```
DELETE FROM <relation name>
WHERE <condition>
```

- Conditions apply to individual tuples; all tuples satisfying the condition are deleted.
- Suppose we want to delete movies which currently have no assigned actors:

```
DELETE FROM Movies
WHERE mID NOT IN (SELECT mID
                  FROM Roles)
```

Database modifications: updates

- **General syntax**:

```
UPDATE <table-name>
SET <new-value-assignments>
WHERE <conditions>;
<new-value-assignments> ::= <new-value-assignment>,
                            <new-value-assignment>
<new-value-assignment>  ::= attribute = <expression> |
                            attribute = constant
```

  – Conditions apply to individual tuples; all tuples satisfying the conditions are updated.

  – Tables are updated one tuple at a time.

- **Example**:

```
UPDATE Movies
SET Length = 134, Year = 1975
WHERE title='Chinatown'
```

SQL and constraints

- Constraints are conditions that must be satisfied by every database instance.
- The constraints should be declared in `CREATE TABLE`
- SQL checks if each modification preserves constraints
- **Table constraints**
- **Attribute constraints**

SQL and constraints

-
  ```
  CREATE TABLE Movies (mID int not null primary key,
                       title char(20),
                       director char(10),
                       year int default 0,
                       length real)
  ```

  ```
  INSERT INTO Movies VALUES (123, 'Chinatown', 'Polanski', 1974, 131)

  INSERT INTO Movies VALUES (123, 'Shining', 'Kubrick', 1980, 131)
  ```

SQL and constraints cont'd

- Two equivalent ways to declare primary keys:

```
CREATE TABLE Movies (
    mID int not null primary key,
    title char(20),
    director char(10),
    year int default 0,
    length real)
```

```
CREATE TABLE Roles (
    mID int not null,
    aID int not null,
    character char(15) not null,
    primary key (mID, aID, character))
```

- What if we have another key, e.g., `(title, director)`?
- We cannot declare it as another primary key.
- But we can declare it as `unique`.

`UNIQUE` in SQL

- Revised example:

```
CREATE TABLE Movies (
    mID int not null,
    title char(20) not null,
    director char(10) not null,
    year int default 0,
    length real,
    primary key (mID),
    unique (title, director))
```

- Unique specifications are verified in the same way as `primary key`.

```
INSERT INTO Movies VALUES (123, 'Chinatown', 'Polanski', 1974)

INSERT INTO Movies VALUES (124, 'Chinatown', 'Polanski', 1974)
```

Inclusion constraints: reminder

- **Referential** integrity constraints: attributes of one relation that refer to attributes of another relation.
- There is an inclusion dependency $R[A_1, \ldots, A_n] \subseteq S[B_1, \ldots, B_n]$ when

$$\pi_{A_1,\ldots,A_n}(R) \quad \subseteq \quad \pi_{B_1,\ldots,B_n}(S)$$

- Most often inclusion constraints occur as a part of a **foreign key**

Foreign keys in SQL

- **Syntax**:

```
CREATE TABLE T1 (...
    foreign key <attr1, ..., attrN>
    references T2 <attr1-1, ...,attrN-1>)
```

In T2, `<attr1-1, ..., attrN-1>` must be present and form a primary key.

- **Example**:

```
CREATE TABLE Roles
  (mID int not null references Movies (mID),
   aID int not null,
   character char(15) not null,
   primary key (mID, aID, character),
   foreign key (aID) references Actors (aID))
```

More on referential integrity constraints

- It is possible to associate reaction policies to violations of referential integrity constraints.
- Violations arise from
  - updates on referred attribute, or
  - tuple deletions
- Reactions operate on the referring table (i.e., the table with the foreign key), after changes to the referred table. They are:
  - `NO ACTION`: reject the change on the referred table (the default action);
  - `CASCADE`: propagate the change from the referred table to the referring table;
  - `SET NULL`: nullify the referring attribute;
  - `SET DEFAULT`: assign default value to the referring attribute.

CHECK Constraints

- The most generic constraint type.
- It allows you to specify that the value of an attribute must satisfy some condition.

```
CREATE TABLE R (A int,
                B int CHECK (B > 0),
                C int CHECK (C > 0),
                CHECK (B > C))
```

- A CHECK constraint is satisfied if the check expression evaluates to true or unknown.
  - i.e., it is violated if the check expression evaluates to false.

Naming Constraints

- It allows you to specify a name for a constraint.

```
CREATE TABLE R (A int,
                B int CONSTRAINT positive_B CHECK (B > 0),
                C int CONSTRAINT positive_C CHECK (C > 0),
                CHECK (B > C))
```

- It allows you to refer to the constraint when you need to update it.

```
ALTER TABLE R DROP CONSTRAINT ''positive_B'';
```

Data retrieval: Basic structure

- A typical SQL query has the form:

```
SELECT A1, A2, ..., An
FROM R1, R2, ..., Rm
WHERE C
```

- `A1, A2, ..., An` represent attributes
- `R1, R2, ..., Rm` represent relations
- `C` is a condition;
  - Simple conditions can be combined using the logical connectives **AND, OR, NOT**.
- This query is equivalent to the Relational Algebra expression:

$$\pi_{A_1,A_2,\cdots A_n}(\sigma_C(R_1 \times R_2 \times \cdots \times R_m))$$

- The result of an SQL query is a new relation.

Examples of SQL queries

Query: Find titles of movies directed by Lucas

```
SELECT M.title
FROM Movies M
WHERE M.director = 'Lucas'
```

- *Tuple variable* `M` ranges over tuples of `Movies`

Evaluation strategy:

- `FROM` clause produces Cartesian product of (input) listed relations;
- `WHERE` clause assigns tuples to `M` in sequence and produces a relation containing only the tuples satisfying the condition;
- `SELECT` clause retains only the listed (output) attributes.

Result:

Examples of SQL queries

Query: Find titles of movies directed by Lucas

```
SELECT M.title
FROM Movies M
WHERE M.director = 'Lucas'
```

- *Tuple variable* `M` ranges over tuples of `Movies`

Evaluation strategy:

- `FROM` clause produces Cartesian product of (input) listed relations;
- `WHERE` clause assigns tuples to `M` in sequence and produces a relation containing only the tuples satisfying the condition;
- `SELECT` clause retains only the listed (output) attributes.

Result:

title
Star Wars IV
American Graffiti

More on tuple variables

- What is the result of the following queries?

1.
   ```
   SELECT *
   FROM Movies M1, Movies M2
   ```

2.
   ```
   SELECT M1.mID
   FROM Movies M1, Movies M2
   ```

3.
   ```
   SELECT M.mID
   FROM Movies M , Roles R
   WHERE M.mID = R.mID
   ```

4.
   ```
   SELECT mID
   FROM Movies, Roles
   WHERE Movies.mID = Roles.mID
   ```

5.
   ```
   SELECT *
   FROM Movies, Movies
   ```

More on the SELECT clause

- An asterisk (*) in the SELECT clause denotes "all attributes"

```
SELECT *
FROM Movies
```

Result:

More on the SELECT clause

- An asterisk (*) in the SELECT clause denotes "all attributes"

```
SELECT *
FROM Movies
```

Result:

mID	title	director	year	length
1	Shining	Kubrick	1980	146
2	Player	Altman	1992	146
3	Chinatown	Polanski	1974	131
4	Repulsion	Polanski	1965	143
5	Star Wars IV	Lucas	1977	126
6	American Graffiti	Lucas	1973	110
7	Full Metal Jacket	Kubrick	1987	156

More on the SELECT clause cont'd

- The SELECT clause can contain arithmetic expressions involving the operators +, , *, and /, and operating on constants or attributes of tuples.
- `SELECT M.mID, M.title, M.director, 2009 - M.year`
  `FROM Movies M`

  Result:

- **Note:** SQL does not permit the "-" character in names.
- **Note:** SQL names are case insensitive, i.e. you can use capital or small letters.

More on the SELECT clause cont'd

- The SELECT clause can contain arithmetic expressions involving the operators +, , *, and /, and operating on constants or attributes of tuples.

```
SELECT M.mID, M.title, M.director, 2009 - M.year
FROM Movies M
```

Result:

mID	title	director	?column?
1	Shining	Kubrick	30
2	Player	Altman	28
3	Chinatown	Polanski	36
4	Repulsion	Polanski	45
5	Star Wars IV	Lucas	33
6	American Graffiti	Lucas	37
7	Full Metal Jacket	Kubrick	23

Renaming in SQL

- New attribute names can be introduced in `SELECT` using keyword `AS`.
- The query

```
SELECT M.mID, M.title, M.director, 2009 - M.year AS numOfYears
FROM Movies M
```

Result:

mID	title	director	numOfYears
1	Shining	Kubrick	30
2	Player	Altman	28
3	Chinatown	Polanski	36
4	Repulsion	Polanski	45
5	Star Wars IV	Lucas	33
6	American Graffiti	Lucas	37
7	Full Metal Jacket	Kubrick	23

More on the SELECT clause cont'd

- So far, in relational algebra and calculus, we operated with sets. SQL, on the other hand, deals with bags, that is, sets with duplicates.
- To force the elimination of duplicates, use the keyword `DISTINCT` after `SELECT`.
- **Query**: Find the names of directors who directed at least one movie.

```
SELECT DISTINCT director
FROM Movies
```

– Since a `director` could have directed more than one movie, `DISTINCT` is necessary not to include his name more than once.

Result:

More on the SELECT clause cont'd

- So far, in relational algebra and calculus, we operated with sets. SQL, on the other hand, deals with bags, that is, sets with duplicates.
- To force the elimination of duplicates, use the keyword `DISTINCT` after `SELECT`.
- **Query**: Find the names of directors who directed at least one movie.

```
SELECT DISTINCT director
FROM Movies
```

– Since a `director` could have directed more than one movie, `DISTINCT` is necessary not to include his name more than once.

Result:

director
Kubrick
Altman
Polanski
Lucas

Join queries

Query: Find actors playing in movies directed by Lucas:

```
SELECT A.aName
FROM Artists A, Roles R, Movies M
WHERE M.director = 'Lucas' AND
      R.mID = M.mID AND
      A.aID = R.aID
```

Evaluation strategy:

- `FROM` clause produces Cartesian product of (input) listed relations;
- `WHERE` clause:
  - **Selection condition** `M.director = 'Lucas'` eliminates irrelevant tuples;
  - **Join conditions** `R.mID = M.mID AND A.aID = R.aID` relates facts to each other;
- `SELECT` clause retains only the listed (output) attributes.

Nested subqueries

- SQL provides a mechanism for the nesting of subqueries.
- A *subquery* is a select-from-where expression that is nested within another query.

  – In general, a `WHERE` clause could contain *another query*, and test some relationship between an attribute and the result of that query.
- A common use of subqueries is to *perform tests* for
  - set membership,
  - set comparisons, and
  - set cardinality.
- `R IN S` tests for set membership

Nested subqueries cont'd

- **Example:** Find actors playing in movies directed by Lucas:

```
SELECT A.aName
FROM Artists A
WHERE A.aID IN (SELECT DISTINCT R.aID
                FROM Roles R, Movies M            -- subquery
                WHERE M.director = 'Lucas' AND
                      R.mID = M.mID)
```

- **Evaluation strategy:**

  – the subquery is evaluated once to produce the set of aID's of actors who played in movies of Lucas

  Result:

aID

  – each tuple (as A) is tested against this set

  Final Result:

aName

Nested subqueries cont'd

- **Example:** Find actors playing in movies directed by Lucas:

```
SELECT A.aName
FROM Artists A
WHERE A.aID IN (SELECT DISTINCT R.aID
                FROM Roles R, Movies M            -- subquery
                WHERE M.director = 'Lucas' AND
                      R.mID = M.mID)
```

- **Evaluation strategy:**

  – the subquery is evaluated once to produce the set of aID's of actors who played in movies of Lucas

Result:

aID
2
4

  – each tuple (as A) is tested against this set

Final Result:

aName
Harrison Ford
Carrie Fisher

String operations

- SQL includes a string-matching operator for comparisons on character strings. Patterns are described using two special characters:
  - underscore `_` – matches any character
  - percent `%` – matches any substring, including the empty one.
  - `attribute LIKE pattern`
- **Examples**:

  pattern `'_a_b_'` matches `cacbc, aabba, etc`

  pattern `'%a%b_'` matches `ccaccbc, aaaabcbcbbd, aba, etc`
- SQL supports a variety of string operations such as
  - concatenation (using ||)
  - converting from upper to lower case (and vice versa)
  - finding string length, extracting substrings, etc.

String operations cont'd

- Is Kubrick spelled with a "k" or "ck" at the end?

```
SELECT Title, Director
FROM Movies
WHERE director LIKE 'Kubri%'
```

- Is Polanski spelled with a "y" or with an "i"?

```
SELECT Title, Director
FROM Movies
WHERE director LIKE 'Polansk%'
```

- What happens if we change it to `WHERE director LIKE 'Polansk_'`?

Set comparison

- `<value>  <condition>  ALL ( <query> )`
  is true if either:
  - `<query>` evaluates to the empty set, or
  - for every `<value1>` in the result of `<query>`,
    `<value>  <condition>  <value1>` is true.
- where `<condition>` can be $<, \leq, >, \geq, \neq, =$
- For example,

  $5 > \texttt{ALL}(\emptyset)$ is true;

  $5 > \texttt{ALL}(\{1, 2, 3\})$ is true;

  $5 > \texttt{ALL}(\{1, 2, 3, 4, 5, 6\})$ is false.

  $5 \neq \texttt{ALL}(\{1, 2, 3, 4\})$ is true.

  $5 = \texttt{ALL}(\{4, 5\})$ is false.
- $\neq$ ALL is equivalent to `NOT IN`
- But, $=$ ALL is not equivalent to `IN`

Set comparison cont'd

- Find directors whose all movies have been completed before 1980.

```
SELECT DISTINCT M.director
FROM Movies M
WHERE 1980 > ALL (SELECT M1.year
                  FROM Movies M1
                  WHERE M1.director = M.director)
```

- **Result:**

director

Set comparison cont'd

- Find directors whose all movies have been completed before 1980.

```
SELECT DISTINCT M.director
FROM Movies M
WHERE 1980 > ALL (SELECT M1.year
                  FROM Movies M1
                  WHERE M1.director = M.director)
```

- **Result:**

director
Polanski
Lucas

Set comparison cont'd

- `<value> <condition> ANY ( <query> )` is true if for some `<value1>` in the result of `<query>`, `<value> <condition> <value1>` is true.
- where `<condition>` can be $<, \leq, >, \geq, \neq$
- For example,

  $5 < \texttt{ANY}(\emptyset)$ is false;

  $5 < \texttt{ANY}(\{1, 2, 3, 4\}$ is false;

  $5 < \texttt{ANY}(\{1, 2, 3, 4, 5, 6\}$ is true.

  $5 \neq \texttt{ANY}(\{1, 2, 3, 4, 5\})$ is true.

  $5 = \texttt{ANY}(\{4, 5\})$ is true.
- $=$ ANY is equivalent to `IN`
- But, $\neq$ ANY is not equivalent to `NOT IN`

Set comparison cont'd

- Find directors who completed some movies before 1980.

```
SELECT DISTINCT M.director
FROM Movies M
WHERE 1980 > ANY (SELECT M1.year
                  FROM Movies M1
                  WHERE M1.director = M.director)
```

- **Result:**

director

Set comparison cont'd

- Find directors who completed some movies before 1980.

```
SELECT DISTINCT M.director
FROM Movies M
WHERE 1980 > ANY (SELECT M1.year
                  FROM Movies M1
                  WHERE M1.director = M.director)
```

- **Result:**

director
Kubrick
Lucas

Set Operations

- The set operations UNION, INTERSECT, and EXCEPT operate on relations and correspond to the relational algebra operations $\cup, \cap, -$.
- Each of the above operations automatically eliminates duplicates; to retain all duplicates use UNION ALL, INTERSECT ALL, and EXCEPT ALL.
- Suppose a tuple occurs `m` times in `R` and `n` times in `S`; then, it occurs:
  - `m + n` times in `R UNION ALL S`
  - `min(m,n)` times in `R INTERSECT ALL S`
  - `max(0, m - n)` times in `R EXCEPT ALL S`
- For the relations below, what is the result of all the above operations?

R:

A	B
1	2
3	4
1	2
3	4

S:

A	B
1	2
5	6
1	2

Empty set traps

- Consider R:

A
1
2

S:

A

- What is the result of the following query?

```
SELECT R.A
FROM R, S
```

Test for empty relation

- `EXISTS(<query>)` returns `true` if the result of `<query>` contains at least one tuple.
- `NOT EXISTS(<query>)` returns `true` if the result of `<query>` contains no tuples.
- **Note:** $\mathrm{R} - \mathrm{S} = \emptyset \Leftrightarrow \mathrm{R} \subseteq \mathrm{S}$

  i.e., `relation S contains relation R` can be written as `NOT EXISTS (R EXCEPT S)`

A more complicated example

- Find actors who played in all the movies directed by “Lucas”
- **Strategy**:
  - Let R contain all the movies directed by “Lucas” and
  - Let S_A contain all the movies in which an actor `A` played.
  - `For each actor A`
    If $R \subseteq S_A$ then output A.

```
SELECT A.aName
FROM Artists A
WHERE NOT EXISTS ((SELECT M.mID
                   FROM Movies M
                   WHERE M.director = 'Lucas')
                  EXCEPT
                  (SELECT R.mID
                   FROM Roles R
                   WHERE R.aID = A.aID))
```

A more complicated example cont'd

Movies:

mID	title	director	year	length
1	Shining	Kubrick	1980	146
2	Player	Altman	1992	146
3	Chinatown	Polanski	1974	131
4	Repulsion	Polanski	1965	143
5	Star Wars IV	Lucas	1977	126
6	American Graffiti	Lucas	1973	110
7	Full Metal Jacket	Kubrick	1987	156

Artists:

aID	aName	nat
1	Jack Nicholson	American
2	Harrison Ford	American
3	Philip Stone	British
4	Carrie Fisher	American

Roles:

mID	**aID**	**character**
1	1	Jack Torrance
1	3	Delbert Grady
3	1	Jake 'J.J.' Gittes
5	2	Han Solo
5	4	Princess Leia Organa
6	2	Bob Falfa

A more complicated example cont'd

Evaluation:

- R:

mID
5
6

S$_1$:

mID

S$_2$:

mID

S$_3$:

mID

S$_4$:

mID

- The subquery in the `WHERE` clause has to be evaluated for every tuple in `Artists`
  - Evaluation for the first tuple `(1, 'Jack Nicholson')`:

`WHERE NOT EXISTS(`

mID
5
6

`-`

mID

`)` is `WHERE NOT EXISTS(`

mID

`)`

which evaluates to

  - Evaluation for the second tuple `(2, 'Harrison Ford')`:

`WHERE NOT EXISTS(`

mID
5
6

`-`

mID

`)` is `WHERE NOT EXISTS(`

mID

`)`

which evaluates to

A more complicated example cont'd

Evaluation:

- R:

mID
5
6

S$_1$:

mID
1
3

S$_2$:

mID
5
6

S$_3$:

mID
1

S$_4$:

mID
5

- The subquery in the `WHERE` clause has to be evaluated for every tuple in `Artists`
  - Evaluation for the first tuple `(1, 'Jack Nicholson')`:

`WHERE NOT EXISTS(`

mID
5
6

`-`

mID
1
3

`)` is `WHERE NOT EXISTS(`

mID
5
6

`)`

which evaluates to `false`

  - Evaluation for the second tuple `(2, 'Harrison Ford')`:

`WHERE NOT EXISTS(`

mID
5
6

`-`

mID
5
6

`)` is `WHERE NOT EXISTS(`

mID

`)`

which evaluates to `true`

A more complicated example cont'd

– Evaluation for the third tuple `(3, 'Philip Stone')`:

`WHERE NOT EXISTS(` $\begin{array}{|c|}\hline \text{mID} \\ \hline 5 \\ \hline 6 \\ \hline \end{array}$ `-` $\begin{array}{|c|}\hline \text{mID} \\ \hline \\ \\ \end{array}$ $)$ is `WHERE NOT EXISTS(` $\begin{array}{|c|}\hline \text{mID} \\ \hline \\ \\ \end{array}$ $)$ which

evaluates to

– Evaluation for the fourth tuple `(4, 'Carrie Fisher')`:

`WHERE NOT EXISTS(` $\begin{array}{|c|}\hline \text{mID} \\ \hline 5 \\ \hline 6 \\ \hline \end{array}$ `-` $\begin{array}{|c|}\hline \text{mID} \\ \hline \\ \\ \end{array}$ $)$ is `WHERE NOT EXISTS(` $\begin{array}{|c|}\hline \text{mID} \\ \hline \end{array}$ $)$ which

evaluates to

Therefore, the **final result** of the query is $\begin{array}{|c|}\hline \text{aName} \\ \hline \end{array}$

A more complicated example cont’d

– Evaluation for the third tuple `(3, 'Philip Stone')`:

`WHERE NOT EXISTS(`

mID
5
6

`-`

mID
1

`)` is `WHERE NOT EXISTS(`

mID
5
6

`)` which evaluates to `false`

– Evaluation for the fourth tuple `(4, 'Carrie Fisher')`:

`WHERE NOT EXISTS(`

mID
5
6

`-`

mID
5

`)` is `WHERE NOT EXISTS(`

mID
6

`)` which evaluates to `false`

Therefore, the **final result** of the query is

aName
Harrison Ford

Structured Query Language II

Null values

- Movies:

mID	title	director	year	length
1	Shining	Kubrick	1980	146
2	NULL	Altman	1992	146
3	Chinatown	NULL	1974	131
4	Repulsion	Polanski	1965	143
5	Star Wars IV	Lucas	1977	126
6	American Graffiti	Lucas	1973	110
7	Full Metal Jacket	Kubrick	1987	156
8	Kubrick	A Clockwork Orange	NULL	NULL

- What could null possibly mean? There are three possibilities:
  - Value exists, but is unknown at the moment.
  - Value does not exist.
  - There is no information.
- SQL approach: there is a single general purpose NULL for all cases of missing/inapplicable information

Null values cont'd

- To test for null values:

  ```
  <attribute> IS [NOT] NULL
  ```

- **Example:**
  - For T:

A	B
1	2
NULL	3

  - the result of the query

```
SELECT B
FROM T
WHERE A IS NULL
```

is:

B

```
SELECT B
FROM T
WHERE A IS NOT NULL
```

is:

B

Null values cont'd

- To test for null values:

  ```
  <attribute> IS [NOT] NULL
  ```

- **Example:**
  - For T:

A	B
1	2
NULL	3

  - the result of the query

```
SELECT B
FROM T
WHERE A IS NULL
```

is:

B
3

```
SELECT B
FROM T
WHERE A IS NOT NULL
```

is:

B
2

Nulls cont'd

- Consider S:

A
1
2
1
2
null
null

- Are the following queries equivalent?

1.
```
SELECT A FROM S WHERE A <> 1
UNION ALL
SELECT A FROM S WHERE A = 1
```
2.
```
SELECT A FROM S
```

Nulls cont'd

- Consider R:

A
1
2
1
2
null
null

- Are the following queries equivalent?

1.
```
SELECT A FROM R WHEN A <> 1
UNION ALL
SELECT A FROM R WHEN A = 1
UNION ALL

SELECT A FROM R WHEN A IS NULL
```

2.
```
SELECT A FROM R
```

Nulls and other operations

- **Rule:** For any arithmetic, string, etc. operation, if one argument is NULL, then the result is NULL.
- **Example:**
  - For R:

A
1
NULL

and S:

B
2

  - the result of the query

```
SELECT R.A + S.B AS C
FROM R, S
```

  - is:

Nulls and other operations

- **Rule:** For any arithmetic, string, etc. operation, if one argument is NULL, then the result is NULL.
- **Example:**
  - For R:

A
1
NULL

  and S:

B
2

  - the result of the query

```
SELECT R.A + S.B AS C
FROM R, S
```

  - is:

C
3
NULL

Nulls and other operations cont'd

- A comparison with NULL value returns UNKNOWN
- How does *unknown* interact with Boolean connectives?

Logical value	value
true	1
false	0
unknown	1/2

x logical operator y	value
x AND y	min (x, y)
x OR y	max (x, y)
NOT x	1 - x

Nulls in subqueries

- For R:

A
1
2

and S:

A
1
2
3
4

the result of the query

```
SELECT S.A
FROM S
WHERE S.A NOT IN (SELECT R.A
                  FROM R)
```

- is:

A

Nulls in subqueries

- For R:

A
1
2

and S:

A
1
2
3
4

the result of the query

```
SELECT S.A
FROM S
WHERE S.A NOT IN (SELECT R.A
                  FROM R)
```

- is:

A
3
4

Nulls in subqueries cont'd

- For R:

A
1
2
NULL

and S:

A
1
2
3
4

the result of the query

```
SELECT S.A
FROM S
WHERE S.A NOT IN (SELECT R.A
                  FROM R)
```

- is:

A

Nulls in subqueries cont'd

- Although this result is counterintuitive, it is correct.
- `1 NOT IN {1,2,null}` evaluates to `false`,
- `2 NOT IN {1,2,null}` evaluates to `false`,
- How to evaluate `3 NOT IN (SELECT R1.A FROM R1)?`

  ```
  3 NOT IN {1,2,null}
  =
  ```

- Similarly, `4 NOT IN {1,2,null}` evaluates to
- Thus, the query returns

A

Nulls in subqueries cont'd

- Although this result is counterintuitive, it is correct.
- `1 NOT IN {1,2,null}` evaluates to `false`,
- `2 NOT IN {1,2,null}` evaluates to `false`,
- How to evaluate `3 NOT IN (SELECT R1.A FROM R1)`?

  `3 NOT IN {1,2,null}`

  = $(3 \neq 1)$ AND $(3 \neq 1)$ AND $(3 \neq \texttt{null})$

  = `true AND true AND unknown`

  = `unknown`
- Similarly, `4 NOT IN {1,2,null}` evaluates to `unknown`.
- Thus, the query returns

A

Join relations

Query: Find actors playing in movies directed by Lucas:

- Recall:

```
SELECT A.aName
FROM Movies M, Roles R, Artists A
WHERE M.director = 'Lucas' AND
      R.mID = M.mID AND
      A.aID = R.aID
```

- We will see other ways to write this query.

Join relations cont'd

- Join operations take two relations and return as result another relation.
- These additional operations are typically used as subquery expressions in the `FROM` clause:

```
FROM Table [JoinType] JOIN Table ON <condition>
```

- *Join type* defines how tuples in each relation that do not match any tuple in the other relation (based on the join condition) are treated.
  - `LEFT OUTER`
  - `RIGHT OUTER`
  - `FULL OUTER`

Join relations cont'd

Query: Find actors playing in movies directed by Lucas:

- One solution:

```
SELECT A.aName
FROM Movies M, Roles R, Artists A
WHERE M.director = 'Lucas' AND
      R.mID = M.mID AND
      A.aID = R.aID
```

- Another solution:

```
SELECT A.aName
FROM (Movies M JOIN Roles R ON ((M.director = 'Lucas') AND
                                (R.mID = M.mID)))
      JOIN Artists A ON  A.aID = R.aID
```

Join relations cont'd

- **Studio**:

Name	mID
United	1
United	2
Dreamworks	3

Finance:

mID	Gross
1	156
2	412
4	25

- **Query:** for each studio, find the total gross of its movies:

```
SELECT S.Name, SUM(F.gross) AS Total
FROM Studio S JOIN Finance F ON S.mID = F.mID
GROUP BY S.Name
```

- **Result:**

Name	Total
United	568

- Dreamworks is lost because movie 3 doesn't match anything in `Finance`.

Outer Joins

- **Idea**: perform a join, and keep tuples that do not match too, padding them with nulls.
- The result of the query

```
SELECT *
FROM Studio LEFT OUTER JOIN Film ON S.mID = F.mID
```

is:

Name	mID	mID1	Gross

- The result of the query

```
SELECT S.Name, SUM(F.gross) AS Total
FROM Studio S LEFT OUTER JOIN Finance F ON S.mID = F.mID
GROUP BY S.Name
```

is:

Name	Total

Outer Joins

- The result of the query

```
SELECT *
FROM Studio LEFT OUTER JOIN Film ON S.mID = F.mID
```

is:

Name	mID	mID1	Gross
United	1	1	156
United	2	2	412
Dreamworks	3	NULL	NULL

- The result of the query

```
SELECT S.Name, SUM(F.gross) AS Total
FROM Studio S LEFT OUTER JOIN Finance F ON S.mID = F.mID
GROUP BY S.Name
```

is:

Name	Total
United	568
Dreamworks	NULL

Other Outer Joins

- The result of the query

```
SELECT *
FROM Studio RIGHT OUTER JOIN Finance ON S.mID = F.mID
```

is:

Name	mID	mID1	Gross

- The result of the query

```
SELECT *
FROM Studio FULL OUTER JOIN Finance ON S.mID = F.mID
```

is:

Name	mID	mID1	Gross

Other Outer Joins

- The result of the query

```
SELECT *
FROM Studio RIGHT OUTER JOIN Finance ON S.mID = F.mID
```

is:

Name	mID	mID1	Gross
United	1	1	156
United	2	2	412
NULL	NULL	4	25

- The result of the query

```
SELECT *
FROM Studio FULL OUTER JOIN Finance ON S.mID = F.mID
```

is:

Name	mID	mID1	Gross
United	1	1	156
United	2	2	412
Dreamworks	3	NULL	NULL
NULL	NULL	4	25

Derived relations

- Find directors, provided that they have directed at least 2 movies.
- Solution with `HAVING`:

```
SELECT director
FROM Movies
GROUP BY director
HAVING COUNT(mID) >= 2
```

- Solution with a derived relation, `Temp`:

```
SELECT director
FROM (SELECT director, COUNT(mID) AS CTN
      FROM Movies
      GROUP BY director) AS Temp
WHERE CTN > 2
```

Views

- Provide a mechanism to hide certain data from the view of certain users.
- Provide a way to save intermediate results, for future reference.
- Usually it is done when the result of a certain query is needed often
- Syntax:
  `CREATE VIEW <name> (<attributes>) AS <query>`
- Example: suppose we need directors and actors playing in movies directed by them.

```
CREATE VIEW DirAct (dir, act) AS
   SELECT M.director, R.aID
   FROM Movies M, Roles R
   WHERE M.mID = R.mID
```

Using views

- Once a view is created, it can be used in queries.
- Find the actors who played in movies directed by “Lucas”

```
SELECT act
FROM DirAct
WHERE dir = 'Lucas'
```

- When the view is no longer needed, it has to be dropped.

```
DROP VIEW <name>
```

WITH Clause

- Allows views to be defined locally to a query, rather than globally. Analogous to procedures in a programming language.

```
WITH DirAct (dir, act) AS
   (SELECT M.director, A.aID
   FROM Movies M, Roles R
   WHERE M.mID = R.mID)
SELECT act
FROM DirAct
WHERE dir = 'Lucas'
```

- **Note**: the WITH clause it's not supported by PostgreSQL

Structured Query Language III

Aggregate functions

These functions operate on the 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'd

- Movies:

mID	title	director	year	length
1	Shining	Kubrick	1980	146
2	Player	Altman	1992	146
3	Chinatown	Polanski	1974	131
4	Repulsion	Polanski	1965	143
5	Star Wars IV	Lucas	1977	126
6	American Graffiti	Lucas	1973	110
7	Full Metal Jacket	Kubrick	1987	156

- Count the number of tuples in Movies

```
SELECT COUNT(*) AS noTuples
FROM Movies
```

- **Result:**

noTuples

Aggregate functions cont'd

- Movies:

mID	title	director	year	length
1	Shining	Kubrick	1980	146
2	Player	Altman	1992	146
3	Chinatown	Polanski	1974	131
4	Repulsion	Polanski	1965	143
5	Star Wars IV	Lucas	1977	126
6	American Graffiti	Lucas	1973	110
7	Full Metal Jacket	Kubrick	1987	156

- Count the number of tuples in Movies

```
SELECT COUNT(*) AS noTuples
FROM Movies
```

- **Result:**

noTuples
7

Aggregate functions cont'd

- Movies:

mID	title	director	year	length
1	Shining	Kubrick	1980	146
2	Player	Altman	1992	146
3	Chinatown	Polanski	1974	131
4	Repulsion	Polanski	1965	143
5	Star Wars IV	Lucas	1977	126
6	American Graffiti	Lucas	1973	110
7	Full Metal Jacket	Kubrick	1987	156

- Find the number of directors.

```
SELECT COUNT(DISTINCT director) AS noDirectors
FROM   Movies
```

- **Result:**

noDirectors

Aggregate functions cont'd

- Movies:

mID	title	director	year	length
1	Shining	Kubrick	1980	146
2	Player	Altman	1992	146
3	Chinatown	Polanski	1974	131
4	Repulsion	Polanski	1965	143
5	Star Wars IV	Lucas	1977	126
6	American Graffiti	Lucas	1973	110
7	Full Metal Jacket	Kubrick	1987	156

- Find the number of directors.

```
SELECT COUNT(DISTINCT director) AS noDirectors
FROM   Movies
```

- **Result:**

noDirectors
4

Aggregate functions cont'd

- Movies:

mID	title	director	year	length
1	Shining	Kubrick	1980	146
2	Player	Altman	1992	146
3	Chinatown	Polanski	1974	131
4	Repulsion	Polanski	1965	143
5	Star Wars IV	Lucas	1977	126
6	American Graffiti	Lucas	1973	110
7	Full Metal Jacket	Kubrick	1987	156

- Find the average length of Lucas's movies.

```
SELECT AVG(length) AS AVGL
FROM   Movies
WHERE director = 'Lucas'
```

- **Result:**

AVGL

Aggregate functions cont’d

- Movies:

mID	title	director	year	length
1	Shining	Kubrick	1980	146
2	Player	Altman	1992	146
3	Chinatown	Polanski	1974	131
4	Repulsion	Polanski	1965	143
5	Star Wars IV	Lucas	1977	126
6	American Graffiti	Lucas	1973	110
7	Full Metal Jacket	Kubrick	1987	156

- Find the average length of Lucas’s movies.

```
SELECT AVG(length) AS AVGL
FROM   Movies
WHERE director = 'Lucas'
```

- **Result:**

AVGL
118

Aggregation and grouping

- For each director, return the average running time of his/her movies.

```
SELECT Director, AVG(Length) AS Avgl
FROM Movies
GROUP BY Director
```

- How does grouping work?

director	…	length
d_1	…	l_1
…	…	…
d_1	…	l_n
d_2	…	…
…	…	…

$\longrightarrow$

director	
d_1	$\{l_1, \ldots, l_n\}$
…	…

$\longrightarrow$

director	avgl
d_1	$(\sum_{i=1}^{n} l_i)/n$
…	…

Aggregation cont'd

- Movies:

mID	title	director	year	length
1	Shining	Kubrick	1980	146
2	Player	Altman	1992	146
3	Chinatown	Polanski	1974	131
4	Repulsion	Polanski	1965	143
5	Star Wars IV	Lucas	1977	126
6	American Graffiti	Lucas	1973	110
7	Full Metal Jacket	Kubrick	1987	156

- For each director, return the average running time of his/her movies.

-
  ```
  SELECT Director, AVG(length) AS AVGL
  FROM Movies
  GROUP BY director
  ```

Aggregation cont'd

Evaluation:

- First, create groups based on the values of the `director` attribute.

"group"	director	mID	title	year	length
1	Kubrick	1	Shining	1980	146
	Kubrick	7	Full Metal Jacket	1987	156
2	Altman	2	Player	1992	146
3	Polanski	3	Chinatown	1974	131
	Polanski	4	Repulsion	1965	143
4	Lucas	5	Star Wars IV	1977	126
	Lucas	6	American Graffiti	1973	110

- Then, for each group, compute the average length of the movies in it.
- **Result:**

director	AVGL

Aggregation cont'd

Evaluation:

- First, create groups based on the values of the `director` attribute.

"group"	director	mID	title	year	length
1	Kubrick	1	Shining	1980	146
	Kubrick	7	Full Metal Jacket	1987	156
2	Altman	2	Player	1992	146
3	Polanski	3	Chinatown	1974	131
	Polanski	4	Repulsion	1965	143
4	Lucas	5	Star Wars IV	1977	126
	Lucas	6	American Graffiti	1973	110

- Then, for each group, compute the average length of the movies in it.

- **Result:**

director	AVGL
Kubrick	151
Altman	146
Polanski	137
Lucas	118

Rules about grouping

```
SELECT director, title,  AVG(length) AS AVGL
FROM Movies
GROUP BY director
```

ERROR: column "movies.title" must appear in the GROUP BY clause or be used in an aggregate function.

Rule: Attributes in SELECT clause outside of aggregate functions must appear in the GROUP BY list.

Rules about grouping cont'd

How do we evaluate the following query?

```
SELECT director, title,  AVG(length) AS AVGL
FROM Movie
GROUP BY director, title
```

- Create groups based on the values of the `director, title` attributes.

"group"	director	title	mID	year	length
1	Kubrick	Shining	1	1980	146
2	Kubrick	Full Metal Jacket	7	1987	156
3	Altman	Player	2	1992	146
4	Polanski	Chinatown	3	1974	131
5	Polanski	Repulsion	4	1965	143
6	Lucas	Star Wars IV	5	1977	126
7	Lucas	American Graffiti	6	1973	110

Rules about grouping cont'd

- Then, for each group, compute the average length of the movies in it.

- **Result:**

director	title	AVGL
Kubrick	Shining	146
Kubrick	Full Metal Jacket	156
Altman	Player	146
Polanski	Chinatown	131
Polanski	Repulsion	143
Lucas	Star Wars IV	126
Lucas	American Graffiti	110

Nulls and aggregation

- **Rule** for nulls and aggregate functions:
  - First, ignore all nulls,
  - and then compute the aggregation value.

  Exception: `COUNT(*)`

- **Example:**
  - For R:

A
1
NULL

  - the result of the query `SELECT COUNT(*) AS CTN FROM R`

is:

CTN

  - the result of the query `SELECT COUNT(R.A) AS CTN FROM R`

is:

CTN

Nulls and aggregation

- **Rule** for nulls and aggregate functions:
  - First, ignore all nulls,
  - and then compute the aggregation value.

  Exception: `COUNT(*)`

- **Example:**
  - For R:

A
1
NULL

  - the result of the query `SELECT COUNT(*) AS CTN FROM R`

  is:

CTN
2

  - the result of the query `SELECT COUNT(R.A) AS CTN FROM R`

  is:

CTN
1

Selection based on aggregation results

- Find directors and average length of their movies, provided they made at least two movies.
- Idea:
  - from all the groups of directors, consider only those for whom `COUNT(mID)` $\geq$ 2;
  - for those directors, compute `AVG(Length)`
- SQL has a special syntax for it: `HAVING`.
-
```
SELECT director, AVG(length)
FROM Movies
GROUP BY director
HAVING COUNT(mID) >= 2
```

Selection based on aggregation results cont'd

Evaluation:

- First, create groups based on the values of the `director` attribute.

"group"	director	mID	title	year	length
1	Kubrick	1	Shining	1980	146
	Kubrick	7	Full Metal Jacket	1987	156
2	Altman	2	Player	1992	146
3	Polanski	3	Chinatown	1974	131
	Polanski	4	Repulsion	1965	143
4	Lucas	5	Star Wars IV	1977	126
	Lucas	6	American Graffiti	1973	110

- Then, among these groups, select only those satisfying the condition in `HAVING`:

"group"	director	mID	title	year	length
1	Kubrick	1	Shining	1980	146
	Kubrick	7	Full Metal Jacket	1987	156
3	Polanski	3	Chinatown	1974	131
	Polanski	4	Repulsion	1965	143
4	Lucas	5	Star Wars IV	1977	126
	Lucas	6	American Graffiti	1973	110

Selection based on aggregation results cont'd

Evaluation cont'd:

- For the remaining groups, compute the aggregation `AVG(length)`

- **Result:**

director	AVGL
Kubrick	151
Polanski	137
Lucas	118

Aggregates in WHERE

- Results of aggregates can be used for comparisons not only in the `HAVING` clause.
- Find the director and the title of the longest movie.

```
SELECT M.director, M.title
FROM Movies M
WHERE M.length = (SELECT MAX(M1.length) AS MAXL
                  FROM Movies M1)
```

- **Result:**

director	title

- **Note:**

```
SELECT MAX(M1.length) AS MAXL
FROM Movies M1
```

returns

MAXL

Aggregates in WHERE

- Results of aggregates can be used for comparisons not only in the `HAVING` clause.
- Find the director and the title of the longest movie.

```
SELECT M.director, M.title
FROM Movies M
WHERE M.length = (SELECT MAX(M1.length) AS MAXL
                  FROM Movies M1)
```

- **Result:**

director	title
Kubrick	Full Metal Jacket

- **Note:**

```
SELECT MAX(M1.length) AS MAXL
FROM Movies M1
```

returns

MAXL
156

Aggregates in WHERE cont'd

- Be careful **not** to write:

```
SELECT M.director, M.title
FROM Movies M
WHERE M.length = MAX(SELECT M1.length
                     FROM Movies M1)
```

which is incorrect.

- Instead, you can write in SQL:

```
SELECT M.director, M.title
FROM Movies M
WHERE M.length >= ALL (SELECT M1.length
                       FROM Movies M1)
```

Ordering the output

- Causes tuples to be outputed in a specified order.
- **Syntax:** `ORDER BY <attribute> ASC | DESC`
- Movies:

mID	title	director	year	length
1	Shining	Kubrick	1980	146
2	Player	Altman	1992	146
3	Chinatown	Polanski	1974	131
4	Repulsion	Polanski	1965	143
5	Star Wars IV	Lucas	1977	126
6	American Graffiti	Lucas	1973	110
7	Full Metal Jacket	Kubrick	1987	156

- Consider the query as before, and, this time, we are ordering the output:

```
SELECT director, AVG(length) AS AVGL
FROM Movies
GROUP BY director
HAVING COUNT(mID) >= 2
ORDER BY AVGL
```

Ordering the output cont'd

- Intermediate result, before ordering the output:

director	AVGL
Kubrick	151
Polanski	137
Lucas	118

- Ordering the output ASCending:

director	AVGL

- Ordering the output DESCending:

director	AVGL

Ordering the output cont'd

- Intermediate result, before ordering the output:

director	AVGL
Kubrick	151
Polanski	137
Lucas	118

- Ordering the output `ASC`ending:

director	AVGL
Lucas	118
Polanski	137
Kubrick	151

- Ordering the output `DESC`ending:

director	AVGL
Kubrick	151
Polanski	137
Lucas	118