# DDL / SQL

Data Description Language /
Data Definition Language
Structured Query Language

# Wikipedia: DDL

- A data definition language or data description language (DDL) is a syntax similar to a computer programming language for defining data structures, especially database schemas.
- The data definition language concept and name was first introduced in relation to the Codasyl database model, where the schema of the database was written in a language syntax describing the records, fields, and sets of the user data model.
- Later it was used to refer to a subset of Structured Query Language (SQL) for creating tables and constraints.
- These information tables were specified as SQL/Schemata in SQL:2003.
- The term DDL is also used in a generic sense to refer to any formal language for describing data or information structures.

# Tools

- Open Source DBMS

  - mySQL
    - for windows→
      - ✓ http://www.mysql.com/why-mysql/windows/
    - For Unix →
      - ✓ http://dev.mysql.com/doc/mysql-linuxunix-excerpt/5.1/en/index.html
    - For Mac →
      - ✓ http://www.mamp.info/en/index.html

- Java DB
  - Oracle's supported distribution of the Apache Derby open source database
  - Integrated in Netbeans

- Tutorial:
  - http://www.w3schools.com/sql/

# Creating a Database Table

Syntax: **CREATE TABLE** <name> (<list of elements>)

An element is an attribute definition of the form

attributeName Type

Type can be: Integer, CHAR(XX), VARCHAR(XX), Date, Real

Do not forget the comma between attribute definitions

Important the difference between CHAR and VARCHAR

KEY (<attribute list>)

FOREIGN KEY (<attribute list>) REFERENCING <Table>

Example: *Studio*(*name*, *address*, *budget, president*), Person(name, tel)

**CREATE TABLE** *Studio* (
*name* **CHAR**(20),
*address* **VARCHAR**(20),
*budget* **REAL,**
*president* **CHAR(20),**
**PRIMARY KEY (name),**
**FOREIGN KEY (***president***) REFERENCES** Person(name) )

**CREATE TABLE** *Person* (
*name* **CHAR**(20),
*tel* **INTEGER,**
**PRIMARY KEY (name)** )

# Deleting a Database Table

Syntax: **DROP TABLE** <name>

Example:

**DROP TABLE** *Studio*

# Types

1. **INT** or **INTEGER**.
2. **BOOLEAN**
3. **REAL** or **FLOAT**.
4. **CHAR**(*n*) = fixed length character string, padded with "pad characters."
5. **VARCHAR**(*n*) = variable-length strings up to *n* characters.
6. **DATE** & **TIME**

# Declaring Keys

Use **PRIMARY KEY** or **UNIQUE**.

- only one **PRIMARY KEY**
  - The primary key of a relational table uniquely identifies each record in the table. It can either be a normal attribute that is guaranteed to be unique (such as Social Security Number in a table with no more than one record per person) or it can be generated by the DBMS
- many **UNIQUE**s allowed.
- SQL permits implementations to create an *index* (data structure to speed access given a key value) in response to **PRIMARY KEY** only.
- SQL does not allow nulls in primary key, but allows them in “unique” columns (which may have two or more nulls, but not repeated non-null values).
- Each table should have a primary key, and each table can have only ONE primary key.

# Example

Find the differences…

**CREATE TABLE** *Studio-version1* (

*name* **CHAR**(20),

*address* **VARCHAR**(20) **UNIQUE**,

*president* **REAL UNIQUE**,

**PRIMARY KEY**(*name*))

**CREATE TABLE** *Studio-version2* (

*name* **CHAR**(20),

*address* **VARCHAR**(20),

*president* **REAL**,

**UNIQUE** (*address*, *president*),

**PRIMARY KEY**(*name*))

# Other Properties You Can Give to Attributes

1. **NOT NULL** = every tuple must have a real value for this attribute.
2. **DEFAULT** value = a value to use whenever no other value of this attribute is known.

Example:

**CREATE TABLE** *Studio* (

name **CHAR**(30) **PRIMARY KEY**,

addr **CHAR**(50) **DEFAULT** '123 Sesame St',

*president_id* **CHAR(50) NOT NULL**)

# Example

Consider executing the following statement

**INSERT INTO** *Studio*(*name, president_id* )
**VALUES**('Fox', 90943)

The result is the following tuple:

| name | addr | *president_id* |
|---|---|---|
| Fox | 123 Sesame St. | 90943 |

Primary key is by default not NULL.
This insertion is legal. It is OK to list a subset of the attributes and values for only this subset. But if we had forgot to specify a value for *president_id* then the insertion could not be made.

# Changing Columns

Add an attribute to relation *R* with

**ALTER TABLE** *R* **ADD** <column declaration>

Example:

**CREATE TABLE** *MovieStar* (

*name* **CHAR**(30),

*address* **VARCHAR**(255),

*gender* **CHAR(1)**,

*birthdate* **DATE,**

**PRIMARY KEY**(*name*))

**ALTER TABLE** *MovieStar* **ADD** *phone* **CHAR**(16) **DEFAULT** 'unlisted'

Columns may also be dropped…

Example"

**ALTER TABLE** *MovieStars* **DROP** *birthdate*

# Database Modifications

CRUD operations: Create (insert), Read (select), Update(update), Delete (delete)

e.g. to insert a tuple we use the statement

**INSERT INTO** <relation> **VALUES** (<list of values>)

- In the above statement the values as listed in the same order with which the attributes were declared
- If we want to ignore this order then we list the attributes as arguments of the relation

Example:

Consider *StarsIn*(*movieTitle*, *movieYear*, *starName*)

Insert the fact that Sydney Green stars in The Maltese Falcon

**INSERT INTO** *StarsIn*(*movieTitle, starName, movieYear)*
**VALUES**('The Maltese Falcon', 'Sydney Green', 1942);

# Insertion of the Result of a Query

Syntax: **INSERT INTO** <relation> (<subquery>).

(not so important for the Intro Web Prog. course … )

Example:

Consider

*Studio*(*name, address, presC#*)

*Movie*(*title*, *year*, *length*, *inColor*, *studio-Name*, *prodC#*)

Add to relation *Studio* all the studios that are mentioned in relation *Movie*

```
INSERT INTO Studio(name)
      SELECT DISTINCT studio-Name
      FROM Movie
      WHERE studio-Name NOT IN (SELECT name
                                FROM Studio)
```

# Deletion

Syntax: **DELETE FROM** <relation> **WHERE** <condition>

Semantic: Deletes all tuples satisfying the condition from

the named relation

Example:

Consider *StarsIn*(*movieTitle*, *movieYear*, *starName*)

Sydney Green was not a star in The Maltese Falcon…

**DELETE FROM** *StarsIn*
**WHERE** *movieTitle* = 'The Maltese Falcon' **AND**
*movieYear* = 1942 **AND** *starName* = 'Sydney Green'

As another example, to make the *StarsIn* relation empty execute

**DELETE FROM** *StarsIn*

# Updates

Syntax: **UPDATE** <relation>

**SET** <new-value assignments>

**WHERE** <condition>

Example:

Consider

*Customers( CustID, CustName, ContactName, Address, City, PostalCose, Country)*

**UPDATE** *Customers*
**SET** *ContactName* = 'Pippo ', *City* ='Trento'
**WHERE** *CustName* = 'Pluto'

# SQL

# SQL Queries

Principal form:

**SELECT** desired attributes
**FROM** tuple variables — range over relations
**WHERE** condition about tuple variables

Running example relation schema:

*Movie*(*title*, *year*, *length*, *inColor*, *studio-Name*, *producerC#*)
*StarsIn*(*movieTitle*, *movieYear*, *starName*)
*MovieStar*(*name*, *address*, *gender*, *birthdate*)
*MovieExec*(*name*, *address*, *cert#*, *netWorth*)
*Studio*(*name*, *address*, *presC#*)

# Example

*Consider relation:*
*Movie*(*title*, *year*, *length*, *inColor*, *studio-Name*, *producerC#*)

Find all the movies produced by Disney Studios in 1990…

```
SELECT *
FROM Movie
WHERE studio-Name='Disney' AND year=1990
```

Find all the movies made by Fox that are at least 100 min. long

```
SELECT *
FROM Movie
WHERE studio-Name='Fox' AND length >= 100
```

# Star as List of All Attributes

*Movie*:

| *title* | *year* | *length* | *in-Color* | *studio-Name* | *produceC#* |
|---|---|---|---|---|---|
| Star Wars | 1977 | 124 | true | Fox | 12345 |
| Mighty Ducks | 1991 | 104 | true | Disney | 67890 |
| Wayne's World | 1992 | 95 | true | Paramaount | 99999 |
| Spider-Man | 2002 | 121 | true | Columbia | 12345 |
| Episode I | 1999 | 133 | true | Fox | 45634 |
| Episode II | 2002 | 142 | true | Fox | 23456 |

**SELECT** *
**FROM** *Movie*
**WHERE** *studio-Name*='Disney' **AND** *year*>=1990

| *title* | *year* | *length* | *in-Color* | *studio-Name* | *produceC#* |
|---|---|---|---|---|---|
| Mighty Ducks | 1991 | 104 | true | Disney | 67890 |

# Projection in SQL

*Movie*:

| *title* | *year* | *length* | *in-Color* | *studio-Name* | *produceC#* |
|---|---|---|---|---|---|
| Star Wars | 1977 | 124 | true | Fox | 12345 |
| Mighty Ducks | 1991 | 104 | true | Disney | 67890 |
| Wayne's World | 1992 | 95 | true | Paramaount | 99999 |
| Spider-Man | 2002 | 121 | true | Columbia | 12345 |
| Episode I | 1999 | 133 | true | Fox | 45634 |
| Episode II | 2002 | 142 | true | Fox | 23456 |

**SELECT** *title, length*
**FROM** *Movie*
**WHERE** *studio-Name*='Disney' **AND** *year*>=1990

| *title* | *length* |
|---|---|
| Mighty Ducks | 104 |

# Expressions as Values in Columns

*Movie*:

| *title* | *year* | *length* | *in-Color* | *studio-Name* | *produceC#* |
|---|---|---|---|---|---|
| Star Wars | 1977 | 124 | true | Fox | 12345 |
| Mighty Ducks | 1991 | 104 | true | Disney | 67890 |
| Wayne's World | 1992 | 95 | true | Paramaount | 99999 |
| Spider-Man | 2002 | 121 | true | Columbia | 12345 |
| Episode I | 1999 | 133 | true | Fox | 45634 |
| Episode II | 2002 | 142 | true | Fox | 23456 |

**SELECT** *title* **AS** *name, length * 0.016667* **AS** *h-duration*
**FROM** *Movie*
**WHERE** *studio-Name*='Disney' **AND** *year*>=1990

| *name* | h-*duration* |
|---|---|
| Mighty Ducks | 1 |

# Patterns

- % stands for any string.
- _ stands for any one character.
- "Attribute **LIKE** pattern" is a condition that is true if the string value of the attribute matches the pattern. Also **NOT LIKE** for negation.

Example:
Given relation *Movie*(*title*, *year*, *length*, *inColor*, *studio-Name*, *producerC#*)

find movies with title "Star *something*", where *something* has 4 letters…

**SELECT** *title*
**FROM** *Movie*
**WHERE** *title* **LIKE** 'Star _ _ _ _'

Note: patterns must be quoted, like strings.

# Nulls

Used in place of a value in a tuple's component.

- Interpretation is not exactly “missing value.”
- There could be many reasons why no value is present, *e.g.*, “value inappropriate.”

Operations with expressions that evaluate to **NULL**

- The result is always **NULL**

Comparing expressions that evaluate to **NULL** with values

- 3rd truth value **UNKNOWN**.
- A query only produces tuples if the **WHERE**-condition evaluates to **TRUE** (**UNKNOWN** is not sufficient).

# Example

| *title* | *year* | *length* | *in-Color* | *studio-Name* | *produceC#* |
|---|---|---|---|---|---|
| Mighty Ducks | 1991 | NULL | true | Disney | 67890 |

**SELECT** *title*, *year*
**FROM** *Movie*
**WHERE** *length* < 90 **OR** *length* >= 90

**UNKNOWN** **UNKNOWN**

**UNKNOWN**

Mighty Ducks is not *selected*, even though the **WHERE** condition is a tautology.

# Multi-relation Queries

- List of relations in **FROM** clause.
- Relation-dot-attribute disambiguates attributes from several relations

Example:

Consider

*Movie*(*title*, *year*, *length*, *inColor*, *studio-Name*, *producerC#*)

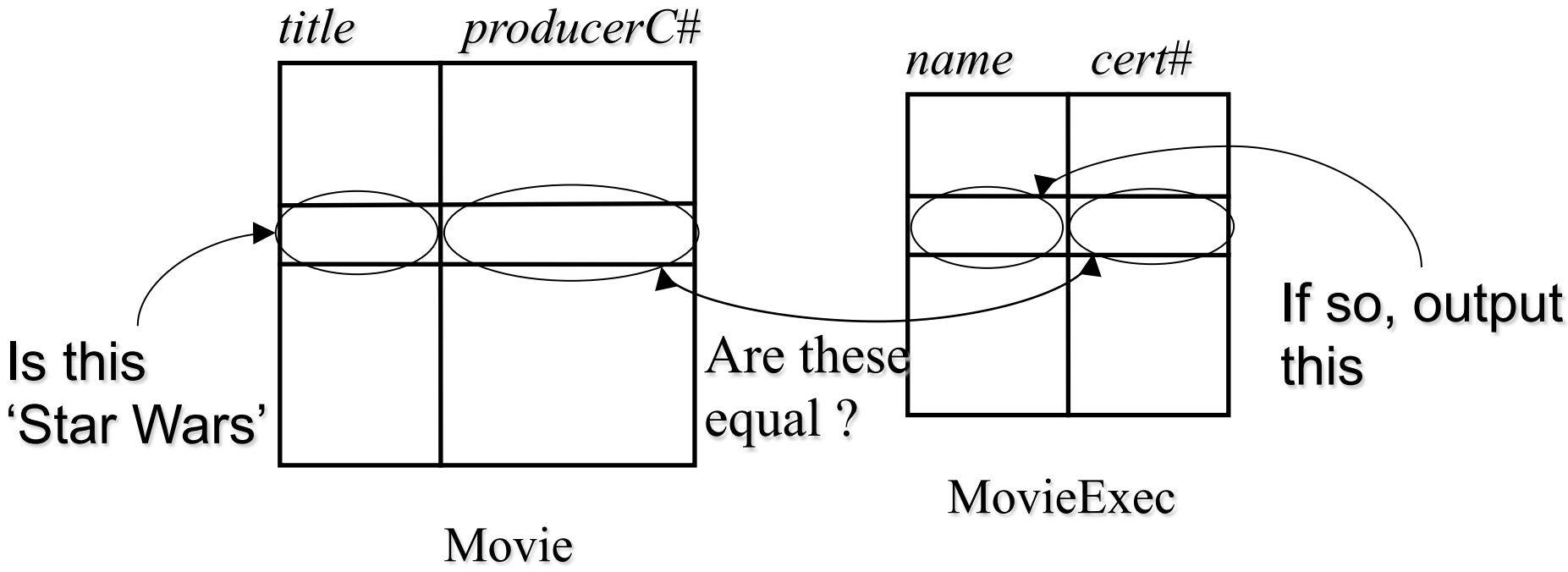
*MovieExec*(*name*, *address*, *cert#*, *netWorth*)

I wonder who is the producer of Star Wars…

**SELECT** *name*
**FROM** *Movie*, *MovieExec*
**WHERE** *title* = 'Star Wars' **AND** *producerC#* = *cert#*

# Example

**SELECT** *name*
**FROM** *Movie*, *MovieExec*
**WHERE** *title* = 'Star Wars' **AND** *producerC#* = *cert#*



# Explicit Tuple Variables

Sometimes we need to refer to two or more copies of a relation.

To do that we use *tuple variables* as aliases of the relations.

Example:

Consider *MovieStar*(*name*, *address*, *gender*, *birthdate*)

Now, find two stars that share the same address….

**SELECT** $Star_1$.*name*, $Star_2$.*name*
**FROM** *MovieStar* $Star_1$, *MovieStar* $Star_2$
**WHERE** $Star_1$.*address* = $Star_2$.*address* **AND**
$Star_1$.*name* < $Star_2$.*name*

Note that $Star_1$.*name* < $Star_2$.*name* is needed to avoid producing

(Carrie, Carrie) and to avoid producing a pair in both orders.

# Union/Intersection/Difference

Consider

*MovieStar*(*name*, *address*, *gender*, *birthdate*)
*MovieExec*(*name*, *address*, *cert#*, *netWorth*)

Find all the names and adresses of people that are either movie stars or movie executives

(**SELECT** *name*, *address*
**FROM** *MovieStar*)
**UNION**
(**SELECT** *name*, *address*
**FROM** *MovieExec*)

# Union/Intersection/Difference

Consider

*MovieStar*(*name*, *address*, *gender*, *birthdate*)
*MovieExec*(*name*, *address*, *cert#*, *netWorth*)

Find all the female movie stars who are also executives and have a net worth over $10,000,000

(**SELECT** *name*, *address*
**FROM** *MovieStar*
**WHERE** *genter* = 'F')
**INTERSECT**
(**SELECT** *name*, *address*
**FROM** *MovieExec*
**WHERE** *netWorth* > 10000000)

# Union/Intersection/Difference

Consider

*MovieStar*(*name*, *address*, *gender*, *birthdate*)
*MovieExec*(*name*, *address*, *cert#*, *netWorth*)

Find all the movie stars who are not executives

(**SELECT** *name*, *address*
**FROM** *MovieStar*)
**EXCEPT**
(**SELECT** *name*, *address*
**FROM** *MovieExec*)

# Forcing Set/Bag Semantics

- Default for select-from-where is bag semantics
  i.e., duplicate rows are retained.
  - Why? Saves time of not comparing tuples as we generate them.
- Default for union, intersection, and difference is set semantics
  i.e. a set in mathematics is a collection of well defined and distinct object
  - we need to sort anyway when we take intersection or difference. (Union seems to be thrown in for good measure!)
- Force set semantics with **DISTINCT** after **SELECT**.
  - But make sure the extra time is worth it.
- Force bag semantics in Union/Intersection/Difference using **ALL**

# Example

Consider

*MovieStar*(*name*, *address*, *gender*, *birthdate*)
*MovieExec*(*name*, *address*, *cert#*, *netWorth*)

Find all the names of people that are either movie stars or movie executives

```
(SELECT name, address
FROM MovieStar)
    UNION ALL
(SELECT name, address
FROM MovieExec)
```

Find all the unique names of people that are stars

```
SELECT DISTINCT name
FROM MovieStar
```