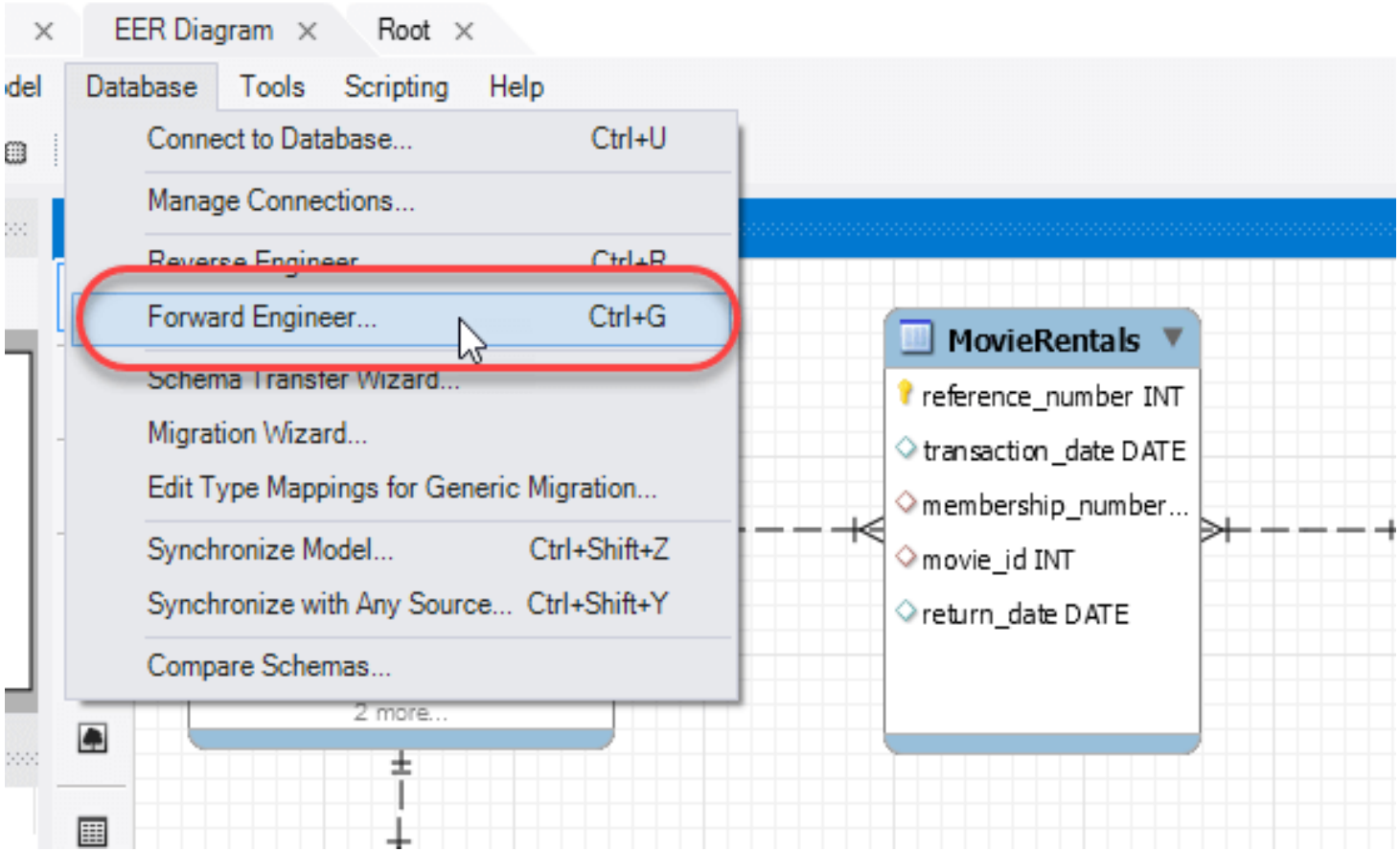
SQL Queries

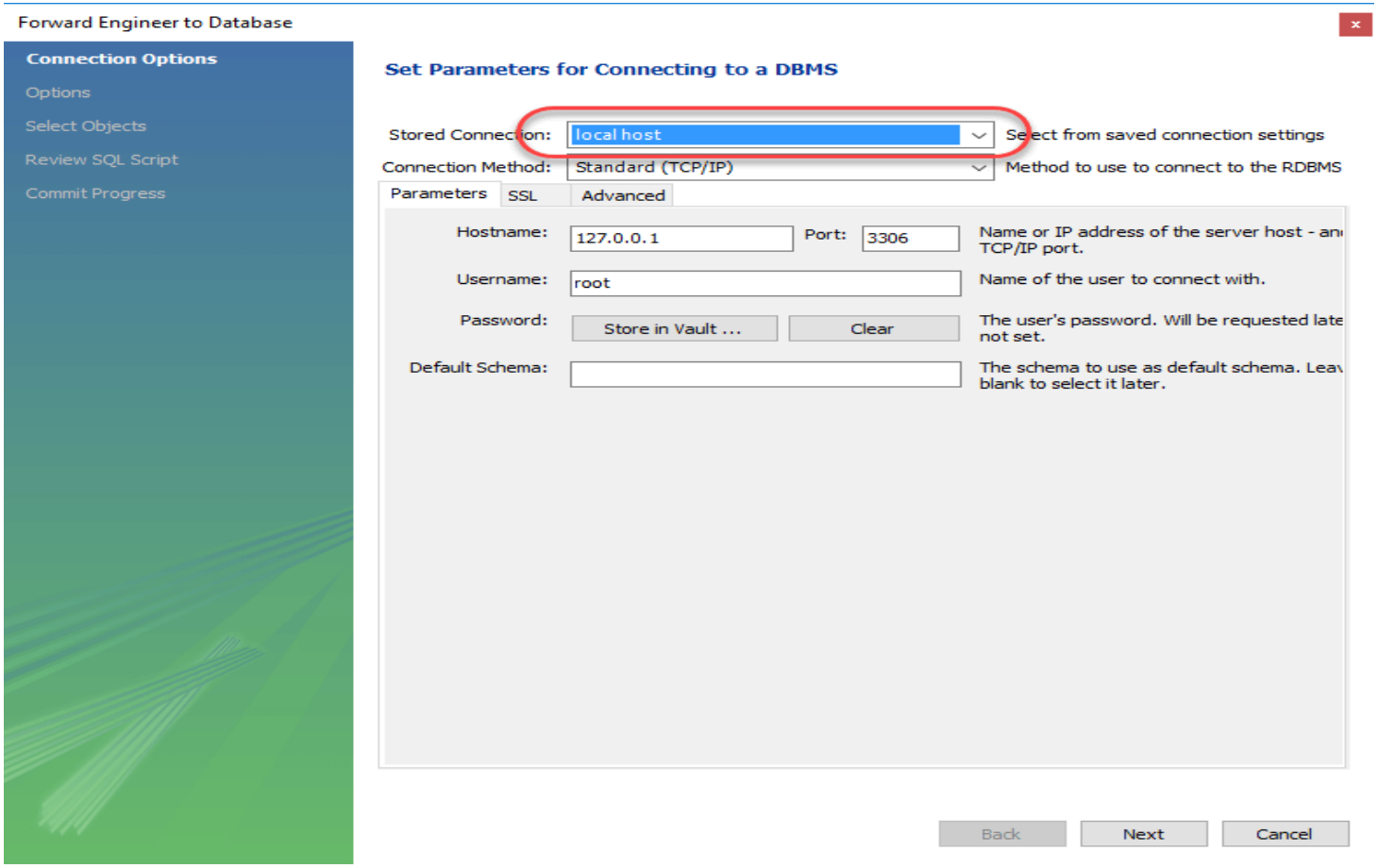
SQL Queries

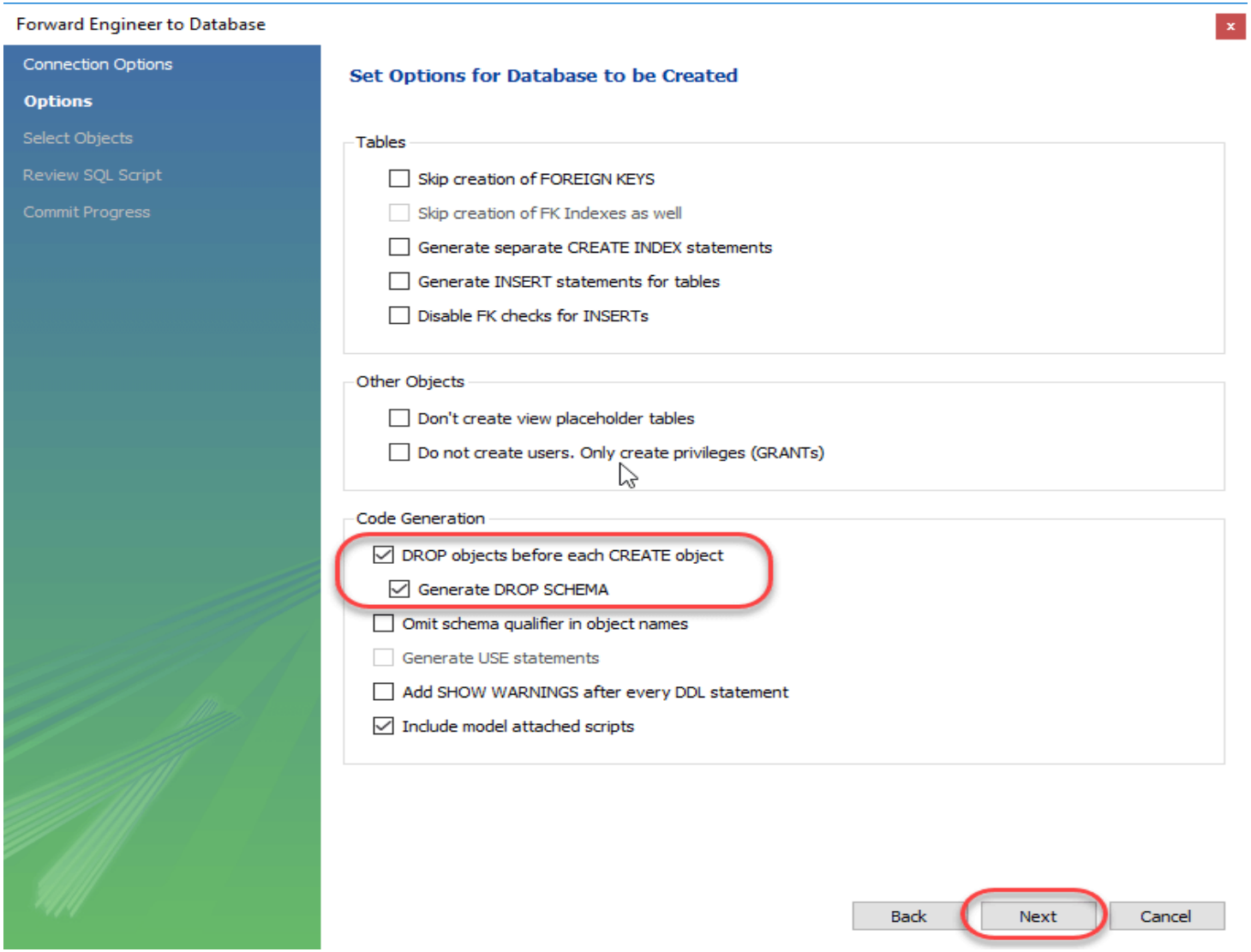
- **Create Database**
- **Creating Tables [MySQL]**
- **Data types**
- **Use [MySQL] workbench ER diagram forward Engineering**

ER tables – Forward Engineering



ER tables – Forward Engineering




Forward Engineer to Database
Connection Options
Options
Select Objects
Review SQL Script
Commit Progress
Set Options for Database to be Created
Tables
Skip creation of FOREIGN KEYS
Skip creation of FK Indexes as well
Generate separate CREATE INDEX statements
Generate INSERT statements for tables
Disable FK checks for INSERTs
Other Objects
Don't create view placeholder tables
Do not create users. Only create privileges (GRANTs)
Code Generation
DROP objects before each CREATE object
Generate DROP SCHEMA
Omit schema qualifier in object names
Generate USE statements
Add SHOW WARNINGS after every DDL statement
Include model attached scripts
Back
Next
Cancel

CREATE DATABASE

CREATE DATABASE is the SQL command for creating a database

- ***CREATE DATABASE Movies;***
- ***CREATE SCHEMA Movies;***

CREATE Database

Check Multiple databases Condition:

- ***CREATE DATABASE IF NOT EXISTS movies;***
- ***CREATE DATABASE IF EXISTS;***
- ***SHOW DATABASES;***

CREATE database

Collation and Character Set :

- Data is stored in RDBMS using a specific character set.
- The character set is defined at different levels viz, server , database , table and columns.
- select the rules of collation and character set for the data base

CREATE DATABASE IF NOT EXISTS movies CHARACTER SET latin1 COLLATE latin1_swedish_ci

CREATE TABLES

- ***CREATE TABLE [IF NOT EXISTS] `TableName` (`fieldname` dataType [optional parameters]) ENGINE = storage Engine;***

- ***CREATE TABLE [IF NOT EXISTS] `TableName` (`fieldname` dataType [AUTO_INCREMENT , NOT NULL]) ENGINE = storage Engine;***

- ***CREATE TABLE [IF EXISTS] `TableName'***

CREATE TABLES

```
CREATE  TABLE IF NOT EXISTS `MyFlixDB`.`Members`
 ( `membership_number` INT  AUTOINCREMENT ,
   `full_names` VARCHAR(150) NOT NULL ,
  `gender` VARCHAR(6) ,
   `date_of_birth` DATE ,
    `physical_address` VARCHAR(255) ,
    `postal_address` VARCHAR(255) ,
     `contact_number` VARCHAR(75) ,
      PRIMARY KEY (`membership_number`) )
    ENGINE = InnoDB;
```

SQL Statement : Select

SELECT [DISTINCT|ALL] { * | [fieldExpression [AS newName]}
FROM tableName [alias]
[WHERE condition]
[GROUP BY fieldName(s)]
[HAVING condition] ORDER BY fieldName(s)

- ***SELECT*** is the SQL keyword that lets the database know that user is retrieving the data.
- ***[DISTINCT | ALL]*** are optional keywords that can be used to fine tune the results returned from the SQL SELECT statement. If nothing is specified then ALL is assumed as the default.
- ***{ * | [fieldExpression [AS newName] }*** at least one part must be specified, "*" selected all the fields from the specified table name, fieldExpression performs some computations on the specified fields
- ***FROM tableName*** is mandatory and must contain at least one table, multiple tables must be separated using commas or joined using the JOIN keyword.
- ***WHERE*** condition is optional, it can be used to specify criteria in the result set returned from the query.
- ***GROUP BY*** is used to put together records that have the same field values.
- ***HAVING*** condition is used to specify criteria when working using the GROUP BY keyword.
- ***ORDER BY*** is used to specify the sort order of the result set.

Select : Examples

membership_ number	full_names	gender	date_of_ birth	physical_ address	postal_ address	contct_ number
1	Janet Jones	Female	21-07-1980	First Street Plot No 4	Private Bag	0759 253 542
2	Janet Smith Jones	Female	23-06-1980	Melrose 123	NULL	NULL
3	Robert Phil	Male	12-07-1989	3rd Street 34	NULL	12345
4	Gloria Williams	Female	14-02-1984	2nd Street 23	NULL	NULL

R1: **members** table
R2: **movies** table

movie_id	title	director	year_released	category_id
1	Pirates of the Caribean 4	Rob Marshall	2011	1
2	Forgetting Sarah Marshal	Nicholas Stoller	2008	2
3	X-Men	NULL	2008	NULL
4	Code Name Black	Edgar Jimz	2010	NULL
5	Daddy's Little Girls	NULL	2007	8
6	Angels and Demons	NULL	2007	6
7	Davinci Code	NULL	2007	6
9	Honey mooners	John Schultz	2005	8
16	67% Guilty	NULL	2012	NULL

Select Query Examples

1. *Get a list of all the registered library members from our database:*

   SELECT * FROM `members`;

2. *Get a list of all the registered library members from our database*

   SELECT `full_names`,`gender`,`physical_address` FROM `members`;

3. *Get a list of all the registered library members from our database*

   SELECT `full_names`,`gender`,`physical_address` FROM `members`

   WHERE members.gender = 'Female'

Select Examples

membership_ number	full_names	gender	date_of_ birth	physical_ address	postal_ address	contct_ number
1	Janet Jones	Female	21-07-1980	First Street Plot No 4	Private Bag	0759 253 542
2	Janet Smith Jones	Female	23-06-1980	Melrose 123	NULL	NULL
3	Robert Phil	Male	12-07-1989	3rd Street 34	NULL	12345
4	Gloria Williams	Female	14-02-1984	2nd Street 23	NULL	NULL

full_names	gender	physical_ address
Janet Jones	Female	First Street Plot No 4
Janet Smith Jones	Female	Melrose 123
Robert Phil	Male	3rd Street 34
Gloria Williams	Female	2nd Street 23

full_names	gender	physical_ address
Janet Jones	Female	First Street Plot No 4
Janet Smith Jones	Female	Melrose 123
Gloria Williams	Female	2nd Street 23

Select Query Examples

4. Get a list of movie title and director with movie title and the name of the movie director in one field.

- ***SELECT Concat(`title`, `director`) FROM `movies`;***
- ***SELECT Concat(`title`, ' (', `director`, ')') , `year_released` FROM `movies`;***

5. Apply Alias name for concat filed ;

- ***SELECT Concat(`title`, ' (', `director`, ')') AS 'Concat', `year_released` FROM `movies`;***
- ***SELECT `column_name|value|expression` [AS] `alias_name`;***

Select Examples

Concat(`title`, ' (', `director`, ')')	year_released
Pirates of the Caribean 4 (Rob Marshall)	2011
Forgetting Sarah Marshal (Nicholas Stoller)	2008
NULL	2008
Code Name Black (Edgar Jimz)	2010
NULL	2007
NULL	2007
NULL	2007
Honey mooners (John Schultz)	2005
NULL	2012

Concat	year_released
Pirates of the Caribean 4 (Rob Marshall)	2011
Forgetting Sarah Marshal (Nicholas Stoller)	2008
NULL	2008
Code Name Black (Edgar Jimz)	2010
NULL	2007
NULL	2007
NULL	2007
Honey mooners (John Schultz)	2005
NULL	2012

Select Query Examples

6. Get a list of all the members showing the membership number, full names and year of birth. use the LEFT string function to extract the year of birth from the date of birth field

```
SELECT
`membership_number`,`full_names`,LEFT(`date_of_birth`,4) AS
`year_of_birth` FROM members;
```

7. Get a list of all the registered female directors list from members whose dob is more than 1980 from our database where

```
SELECT `full_names`,`gender`,`physical_address`
FROM `members`
WHERE members.gender = 'Female' AND members.
Date of birth >= '1980'
```

Select Examples

membership_number	full_names	year_of_birth
1	Janet Jones	1980
2	Janet Smith Jones	1980
3	Robert Phil	1989
4	Gloria Williams	1984

Select Query Examples

***WHERE clause* Syntax:**

SELECT statement...FROM table name [WHERE condition | GROUP BY `field_name(s)`

HAVING condition] ORDER BY `field_name(s)` [ASC | DESC] ;

```
SELECT * FROM tableName WHERE condition;
```

HERE

- **"SELECT * FROM tableName"** is the standard SELECT statement
- **"WHERE"** is the keyword that restricts select query result set and **"condition"** is the filter to be applied on the results. The filter could be a range, single value or sub query.

Select Query Examples

8. *Get a member's personal details from members table given the membership number 1*

SELECT * FROM `members` WHERE `membership_number` = 1;

membership_number	full_names	gender	date_of_birth	physical_address	postal_address	contct_number
1	Janet Jones	Female	21-07-1980	First Street Plot No 4	Private Bag	0759 253 542

Select Query Examples

Order by clause

- ***SELECT statement FROM table name [WHERE condition | GROUP BY `field_name(s)`***

  HAVING condition] ORDER BY `field_name(s)` [ASC | DESC] ;

"SELECT statement..." is the regular select query
" | " represents alternatives
"[WHERE condition | GROUP BY `field_name(s)` HAVING condition" is the optional condition used to filter the query result sets.
"ORDER BY" performs the query result set sorting
"[ASC | DESC]" is the keyword used to sort result sets in either ascending or descending order. Note ***ASC*** is used as the default.

Select Query Examples

WHERE clause combined with - *AND* LOGICAL Operator

9. Get a list of all the movies in category 2 that were released in 2008.

- ***SELECT * FROM `movies` WHERE `category_id` = 2 AND `year_released` = 2008;***
- ***SELECT * FROM `movies` WHERE `category_id` = 1 OR `category_id` = 2;***

movie_id	title	director	year_released	category_id
2	Forgetting Sarah Marshal	Nicholas Stoller	2008	2

movie_id	title	director	year_released	category_id
1	Pirates of the Caribean 4	Rob Marshall	2011	1
2	Forgetting Sarah Marshal	Nicholas Stoller	2008	2

Select Query Examples

WHERE clause combined with - IN Keyword

WHERE clause when used together with the IN keyword only affects the rows whose values matches the list of values provided in the IN keyword

- ***SELECT * FROM `members` WHERE `membership_number` IN (1,2,3);***
- ***SELECT * FROM `members` WHERE `membership_number` NOT IN (1,2,3);***

membership_number	full_names	gender	date_of_birth	physical_address	postal_address	contct_number
1	Janet Jones	Female	21-07-1980	First Street Plot No 4	Private Bag	0759 253 542
2	Janet Smith Jones	Female	23-06-1980	Melrose 123	NULL	NULL
3	Robert Phil	Male	12-07-1989	3rd Street 34	NULL	12345

Select Query Examples

- ***SELECT * FROM `members` WHERE `membership_number` NOT IN (1,2,3);***

membership_number	full_names	gender	date_of_birth	physical_address	postal_address	contct_number	email
4	Gloria Williams	Female	14-02-1984	2nd Street 23	NULL	NULL	NULL

- ***SELECT * FROM `members` WHERE `gender` = 'Female';***
- ***SELECT * FROM `payments` WHERE `amount_paid` > 2000;***
- ***SELECT * FROM `movies` WHERE `category_id`<> 1;***

Select Query Examples

membership_number	full_names	gender	date_of_birth	physical_address	postal_address	contct_number
1	Janet Jones	Female	21-07-1980	First Street Plot No 4	Private Bag	0759 253 542
2	Janet Smith Jones	Female	23-06-1980	Melrose 123	NULL	NULL
4	Gloria Williams	Female	14-02-1984	2nd Street 23	NULL	NULL

payment_id	membership_number	payment_date	description	amount_paid	external_reference_number
1	1	23-07-2012	Movie rental payment	2500	11
3	3	30-07-2012	Movie rental payment	6000	NULL

movie_id	title	Director	year_released	category_id
2	Forgetting Sarah Marshal	Nicholas Stoller	2008	2
5	Daddy's Little Girls	NULL	2007	8
6	Angels and Demons	NULL	2007	6
7	Davinci Code	NULL	2007	6
9	Honey mooners	John Schultz	2005	8

Select Query Examples

10. get a list of rented movies that have not been returned on time 25/06/2012.

SELECT * FROM `movierentals` WHERE `return_date` < '2012-06-25' AND movie_returned = 0;

reference_number	transaction_date	return_date	membership_number	movie_id	movie_returned
14	21-06-2012	24-06-2012	2	2	0

Select Query Examples

Grouping and aggregate functions

SELECT `gender`,COUNT(`membership_number`) FROM `members`
GROUP BY `gender`;

gender	COUNT('membership_number')
Female	3
Male	5

Select Query Examples

Restricting query results using the HAVING clause

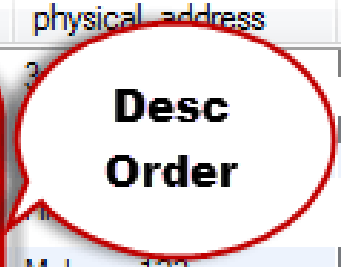
SELECT * FROM `movies` GROUP BY `category_id`,`year_released` HAVING `category_id` = 8;

movie_id	title	director	year_released	category_id
9	Honey mooners	John Schultz	2005	8
5	Daddy's Little Girls	NULL	2007	8

Select Query Examples

```
SELECT * FROM members ORDER BY date_of_birth DESC;
```

membership_number	full_names	gender	date_of_birth	physical_address
3	Robert Phil	Male	1989-07-12	3
4	Gloria Williams	Female	1984-02-14	
1	Janet Jones	Female	1980-07-21	
2	Janet Smith Jones	Female	1980-06-23	Melrose 123
5	Leonard Hofstadter	Male	NULL	Woodcrest
6	Sheldon Cooper	Male	NULL	Woodcrest
7	Rajesh Koothrappali	Male	NULL	Woodcrest
8	Leslie Winkle	Male	NULL	Woodcrest



Select Query Examples

- ***SELECT * FROM `members` ORDER BY `gender`;***
- ***SELECT * FROM `members` ORDER BY `gender`, `date_of_birth` DESC;***

Select Statement

Summary

- *The SQL SELECT keyword is used to query data from the database and it's the most commonly used command.*
- *The simplest form has the syntax "SELECT * FROM tableName;"*
- *Expressions can also be used in the select statement .*

  Example : "SELECT quantity + price FROM Sales"
- *The SQL SELECT command can also have other optional parameters such as WHERE, GROUP BY, HAVING, ORDER BY.*
- *SQL workbench will help develop SQL statements, execute them and produce the output result in the same window.*

Select with Regular Expression

- ***SELECT * FROM `movies` WHERE `title` REGEXP 'code';***
- ***SELECT * FROM `movies` WHERE `title` REGEXP '[^abcd]';***
- ***SELECT * FROM `movies` WHERE `title` REGEXP '^[^abcd]';***

movie_id	title	director	year_released	category_id
4	Code Name Black	Edgar Jimz	2010	NULL
5	Daddy's Little Girls	NULL	2007	8
6	Angels and Demons	NULL	2007	6
7	Davinci Code	NULL	2007	6

movie_id	title	director	year_released	category_id
1	Pirates of the Caribean 4	Rob Marshall	2011	1
2	Forgetting Sarah Marshal	Nicholas Stoller	2008	2
3	X-Men		2008	
9	Honey mooners	John Schultz	2005	8
16	67% Guilty		2012	
17	The Great Dictator	Chalie Chaplie	1920	7
18	sample movie	Anonymous		8
19	movie 3	John Brown	1920	8

Query Sublanguage of SQL

SELECT C.CrsName
FROM Course C
WHERE C.DeptId = 'CS'

OR

$\pi_{CrsName}\sigma_{DeptId='CS'}$**(Course)**

- *Tuple variable* C ranges over rows of Course.
- Evaluation strategy:
  - FROM clause produces Cartesian product of listed tables
  - WHERE clause assigns rows to C in sequence and produces table containing only rows satisfying condition
  - SELECT clause retains listed columns

Join Queries

```
SELECT  C.CrsName
FROM  Course C, Teaching T
WHERE  C.CrsCode=T.CrsCode AND T.Sem='S2000'
```

1. List all CS courses taught in S2000

- Join condition "C.*CrsCode*=T.*CrsCode*"
  - eliminates garbage
- Selection condition " T.*Sem*='S2000' "
  - eliminates irrelevant rows

- Equivalent (using natural join) to:

$$\pi_{CrsName}(\text{Course} \bowtie \sigma_{Sem='S2000'}(\text{Teaching}))$$

$$\pi_{CrsName}(\sigma_{Sem='S2000'}(\text{Course} \bowtie \text{Teaching}))$$

Self-join Queries

2. Find Ids of all professors who taught at least two courses in the same semester:

```
SELECT  T1.ProfId
FROM  Teaching T1, Teaching T2
WHERE  T1.ProfId = T2.ProfId
    AND  T1.Semester = T2.Semester
    AND  T1.CrsCode <> T2.CrsCode
```

Tuple variables essential in this query!

Equivalent to:

$$\pi_{ProfId}(\sigma_{T1.CrsCode \neq T2.CrsCode}(\text{Teaching}[ProfId, T1.CrsCode, Sem] \bowtie \text{Teaching}[ProfId, T2.CrsCode, Sem]))$$

Duplicates

- Duplicate rows not allowed in a relation
- Duplicate elimination from query result is costly and not automatically done; it must be explicitly requested:

```
SELECT DISTINCT .....
FROM .....
```

Set Operators

3. Example: Find all professors in the CS Department and all professors who taught CS courses

- SQL provides UNION, EXCEPT (set difference), and INTERSECT for union compatible tables

```
(SELECT   P.Name
FROM   Professor P, Teaching T
WHERE  P.Id=T.ProfId AND T.CrsCode LIKE 'CS%')
UNION
(SELECT  P.Name
FROM  Professor P
WHERE  P.DeptId = 'CS')
```

Nested Queries

4. List all courses that were not taught in S2000

```
SELECT C.CrsName
FROM Course C
WHERE C.CrsCode NOT IN
      (SELECT T.CrsCode      --subquery
       FROM Teaching T
       WHERE T.Sem = 'S2000')
```

Evaluation strategy: subquery evaluated once to produces set of courses taught in S2000. Each row (as C) tested against this set.

Correlated Nested Queries

5. Output a row <*prof*, *dept*> if *prof* has taught a course in *dept*.

```
SELECT  P.Name, D.Name                    --outer query
  FROM Professor P, Department D
  WHERE  P.Id  IN
          -- set of  all ProfId's who have taught a course in D.DeptId
          (SELECT T.ProfId                    --subquery
           FROM Teaching T, Course C
           WHERE T.CrsCode=C.CrsCode  AND
                 C.DeptId=D.DeptId       --correlation
          )
```

Correlated Nested Queries (con't)

- Tuple variables T and C are *local* to subquery
- Tuple variables P and D are *global* to subquery
- *Correlation*: subquery uses a global variable, D
- The value of D.*DeptId* parameterizes an evaluation of the subquery
- Subquery must (at least) be re-evaluated for each distinct value of D.*DeptId*

- *Correlated queries can be expensive to evaluate*

Aggregates

- Functions that operate on sets:
  - COUNT, SUM, AVG, MAX, MIN
- Produce numbers (not tables)
- Not part of relational algebra

SELECT COUNT(*) FROM Professor P

SELECT MAX (*Salary*) FROM Employee E

Aggregates

6. Count the number of courses taught in S2000

```
SELECT COUNT (T.CrsCode)
FROM Teaching T
WHERE  T.Semester = 'S2000'
```

But if multiple sections of same course are taught, use:

```
SELECT COUNT (DISTINCT T.CrsCode)
FROM Teaching T
WHERE  T.Semester = 'S2000'
```

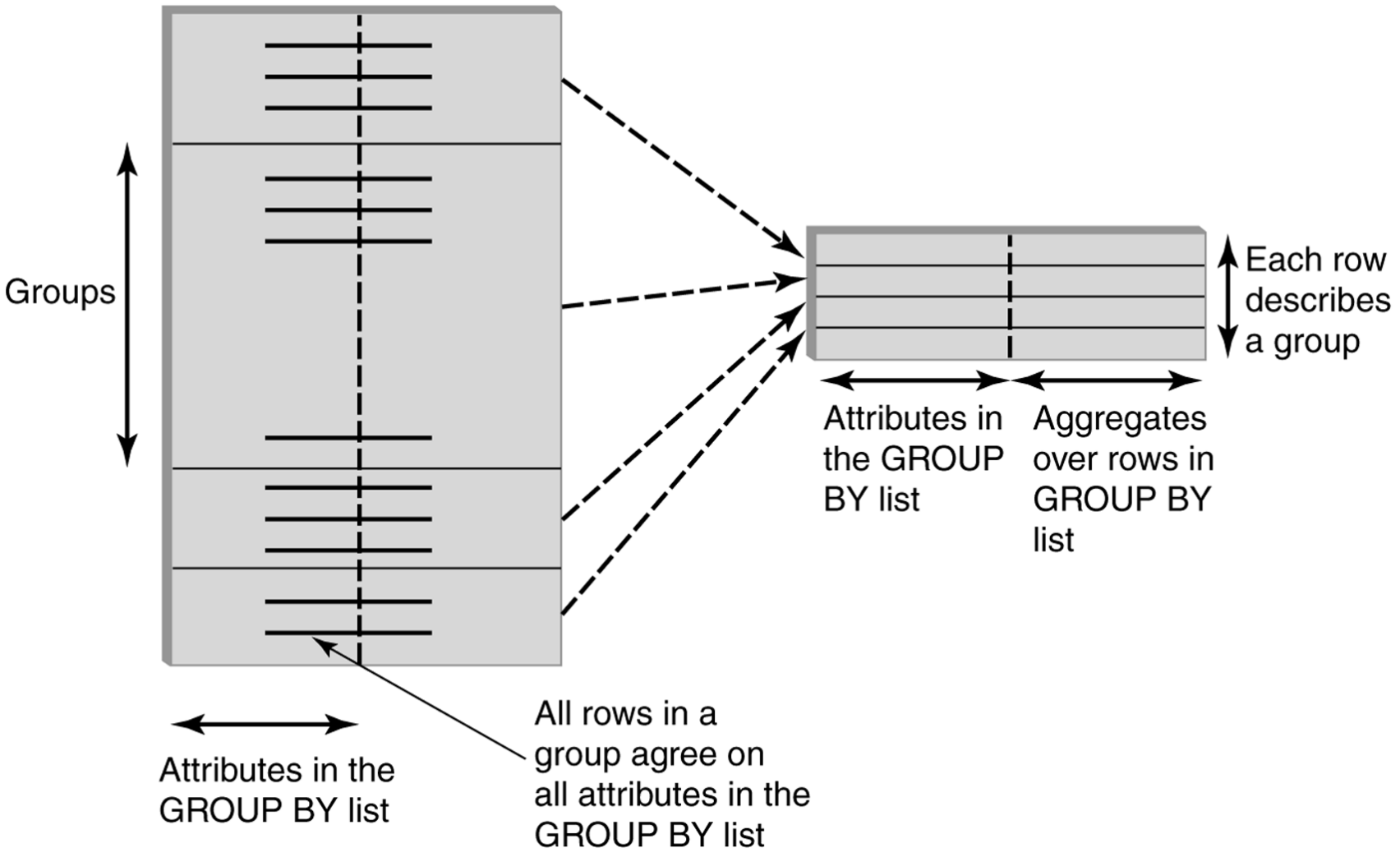
Aggregates: Proper and Improper Usage

SELECT COUNT (T.*CrsCode*), T. *ProfId*
…… --makes no sense (in the absence of GROUP BY clause)

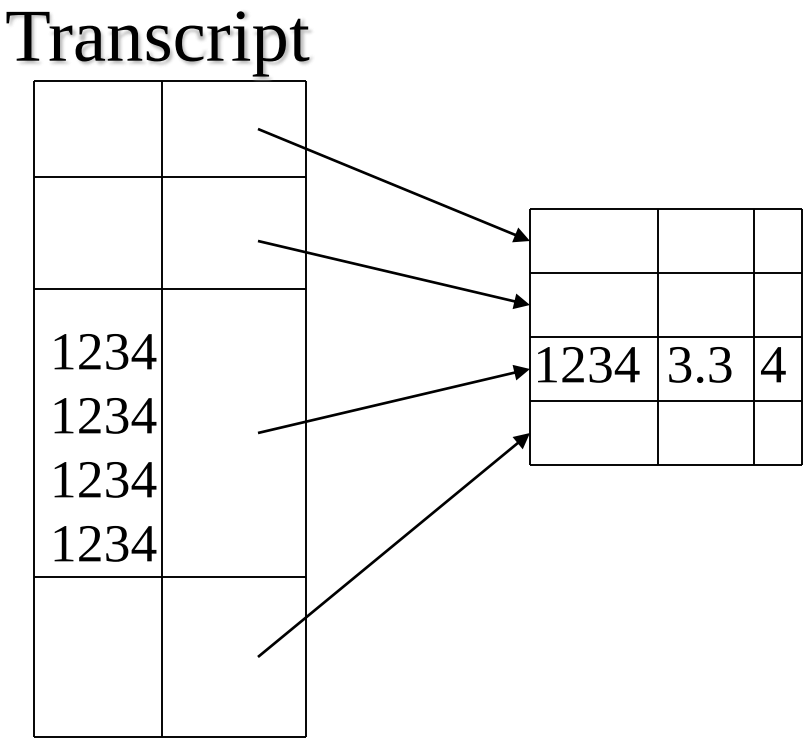
SELECT COUNT (*), AVG (T.*Grade*)
…… --but this is OK

WHERE T.*Grade* > COUNT (SELECT ….
--aggregate *cannot* be applied to result of SELECT statement

GROUP BY


Groups
Each row describes a group
Attributes in the GROUP BY list
Aggregates over rows in GROUP BY list
Attributes in the GROUP BY list
All rows in a group agree on all attributes in the GROUP BY list

GROUP BY - Example



Attributes:

-student's *Id*

-avg grade

-number of courses

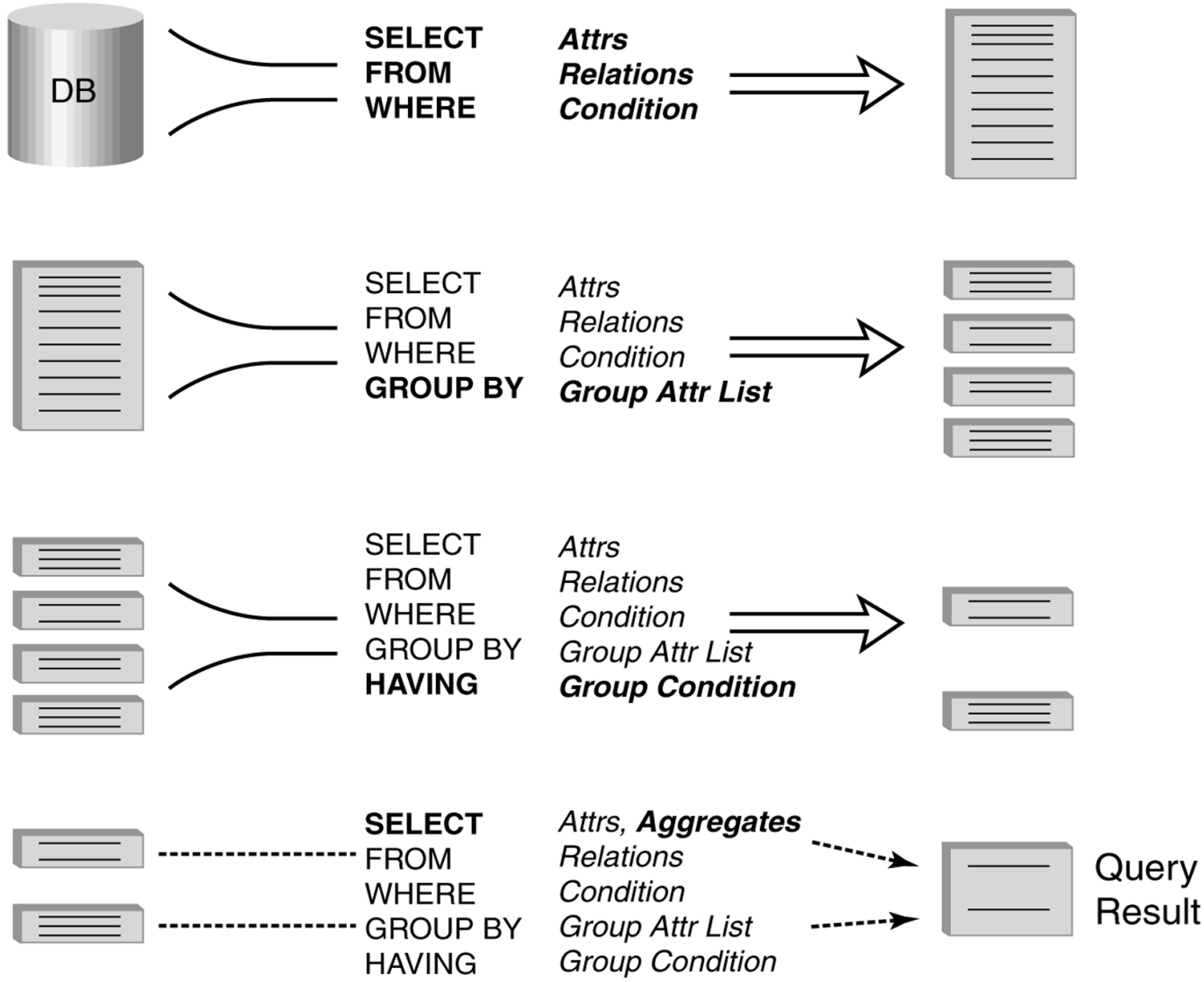
SELECT T.*StudId*, AVG(T.*Grade*), COUNT (*)
FROM Transcript T
GROUP BY T.*StudId*

HAVING Clause

- Eliminates unwanted groups (analogous to WHERE clause)
- HAVING condition constructed from attributes of GROUP BY list and aggregates of attributes not in list

```
SELECT  T.StudId, AVG(T.Grade) AS CumGpa,
                  COUNT (*) AS NumCrs
FROM    Transcript T
WHERE   T.CrsCode  LIKE  'CS%'
GROUP BY  T.StudId
HAVING  AVG (T.Grade) > 3.5
```

Evaluation of GroupBy with Having



Example

- Output the name and address of all seniors on the Dean's List

```
SELECT  S.Name,  S.Address
FROM  Student S, Transcript T
WHERE  S.StudId = T.StudId AND S.Status = 'senior'

GROUP BY  S.StudId                -- wrong
          S.Name, S.Address       -- right

HAVING AVG (T.Grade) > 3.5  AND  SUM (T.Credit) > 90
```

Use of Expressions

Equality and comparison operators apply to strings (based on lexical ordering)

WHERE S.*Name* < 'P'

Concatenate operator applies to strings

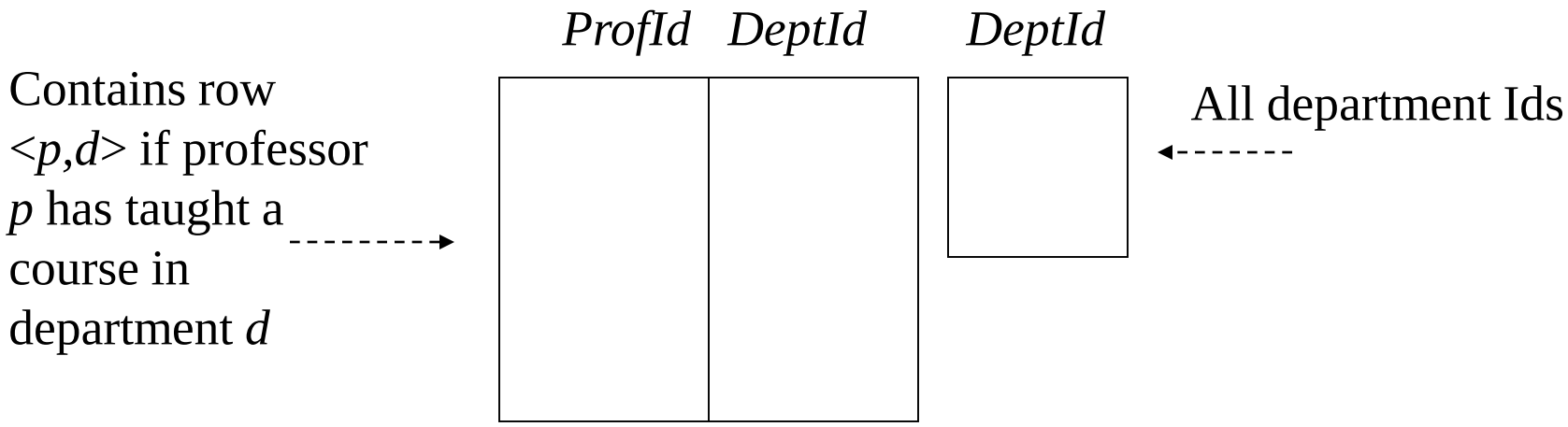
WHERE S.*Name* || '--' || S.*Address* = ….

Expressions can also be used in SELECT clause:

```
SELECT  S.Name || '--' || S.Address AS NmAdd
FROM  Student S
```

Division

- *Query type*: Find the subset of items in one set that are related to *all* items in another set
- *Example*: Find professors who have taught courses in *all* departments
  - Why does this involve division?



Division

- *Strategy for implementing division in SQL*:
  - Find set, A, of all departments in which a particular professor, *p*, has taught a course
  - Find set, B, of all departments
  - Output *p* if A $\supseteq$ B, or, equivalently, if B–A is empty

Division – SQL Solution

```
SELECT P.Id
FROM Professor P
WHERE NOT EXISTS
    (SELECT D.DeptId          -- set B of all dept Ids
     FROM Department D
          EXCEPT
     SELECT C.DeptId          -- set A of dept Ids of depts in
                              -- which P has taught a course
     FROM Teaching T, Course C
     WHERE T.ProfId=P.Id      -- global variable
           AND T.CrsCode=C.CrsCode)
```

ORDER BY Clause

- Causes rows to be output in a specified order

```
SELECT  T.StudId, COUNT (*) AS NumCrs,
        AVG(T.Grade) AS CumGpa
FROM    Transcript T
WHERE   T.CrsCode LIKE 'CS%'
GROUP BY  T.StudId
HAVING  AVG (T.Grade) > 3.5
ORDER BY  DESC  CumGpa,  ASC StudId
```

Query Evaluation Strategy

1 Evaluate FROM: produces Cartesian product, A, of tables in FROM list

2 Evaluate WHERE: produces table, B, consisting of rows of A that satisfy WHERE condition

3 Evaluate GROUP BY: partitions B into groups that agree on attribute values in GROUP BY list

4 Evaluate HAVING: eliminates groups in B that do not satisfy HAVING condition

5 Evaluate SELECT: produces table C containing a row for each group. Attributes in SELECT list limited to those in GROUP BY list and aggregates over group

6 Evaluate ORDER BY: orders rows of C

Views

- Used as a relation, but rows are not physically stored.
  - The contents of a view is *computed* when it is used within an SQL statement
- View is the result of a SELECT statement over other views and base relations
- When used in an SQL statement, the view definition is substituted for the view name in the statement
  - SELECT statement can be nested in FROM clause

View - Example

```
CREATE VIEW  CumGpa (StudId, Cum) AS
  SELECT  T.StudId,  AVG (T.Grade)
  FROM Transcript T
  GROUP BY T.StudId

SELECT  S.Name, C.Cum
FROM  CumGpa C,  Student S
WHERE  C.StudId = S.StudId AND C.Cum > 3.5
```

View Benefits

- *Access Control*: Users not granted access to base tables. Instead they are granted access to the view of the database appropriate to their needs.
  - *External schema* is composed of views.
  - View allows owner to provide SELECT access to a subset of columns (analogous to providing UPDATE and INSERT access to a subset of columns)

Views - Limiting Visibility

```
CREATE VIEW PartOfTranscript (StudId, CrsCode, Semester) AS
    SELECT T. StudId, T.CrsCode, T.Semester      -- limit columns
     FROM Transcript T
     WHERE T.Semester = 'S2000'                  -- limit rows

GRANT SELECT ON PartOfTranscript TO joe
```

This is analogous to:

```
GRANT UPDATE (Grade) ON Transcript TO joe
```

View Benefits (con't)

- *Customization*: Users need not see full complexity of database. View creates the illusion of a simpler database customized to the needs of a particular category of users
- A view is *similar in many ways to a subroutine* in standard programming
  - Can be used in multiple queries

Nulls

- *Conditions*: *x op y* (where *op* is <, >, <>, =, etc.) has value *unknown* (*U*) when either x or y is null
  - WHERE T.*cost* > T.*price*
- *Arithmetic expression*: x *op y* (where *op* is +, -, *, etc.) has value NULL if x or y is null
  - WHERE (T. *price*/T.*cost*) > 2
- *Aggregates*: COUNT counts nulls like any other value; other aggregates ignore nulls

```
SELECT  COUNT (T.CrsCode),  AVG (T.Grade)
FROM Transcript T
WHERE T.StudId = '1234'
```

Nulls (con't)

- WHERE clause uses a *three-valued logic* to filter rows. Portion of truth table:

C1	*C2*	*C1* AND *C2*	*C1* OR *C2*
T	U	U	T
F	U	F	U
U	U	U	U

- Rows are discarded if WHERE condition is *false* or *unknown*
- Ex: WHERE T.*CrsCode* = 'CS305' AND T.*Grade* > 2.5

Modifying Tables - Insert

- Inserting a single row into a table
  - Attribute list can be omitted if it is the same as in CREATE TABLE (but do not omit it)
  - NULL and DEFAULT values can be specified

INSERT INTO Transcript(*StudId*, *CrsCode*, *Semester*, *Grade*)
VALUES (12345, 'CSE305', 'S2000', NULL)

Bulk Insertion

- Insert the rows output by a SELECT

```
CREATE TABLE DeansList (
        StudId      INTEGER,
        Credits     INTEGER,
        CumGpa      FLOAT,
        PRIMARY KEY  StudId )
```

```
INSERT INTO DeansList (StudId, Credits, CumGpa)
SELECT T.StudId, 3 * COUNT (*), AVG(T.Grade)
FROM Transcript T
GROUP BY  T.StudId
HAVING AVG (T.Grade) > 3.5 AND COUNT(*) > 30
```

Modifying Tables - Delete

- Similar to SELECT except:
  - No project list in DELETE clause
  - No Cartesian product in FROM clause (only 1 table name)
  - Rows satisfying WHERE clause (general form, including subqueries, allowed) are deleted instead of output

```
DELETE FROM Transcript T
WHERE T.Grade IS NULL AND
      T.Semester <> 'S2000'
```

Modifying Data - Update

```
UPDATE  Employee E
SET     E.Salary = E.Salary * 1.05
WHERE   E.Department = 'research'
```

- Updates rows in a single table
- All rows satisfying WHERE clause (general form, including subqueries, allowed) are updated

Updating Views

- Question: Since views look like tables to users, can they be updated?
- Answer: Yes – a view update changes the underlying base table to produce the requested change to the view

```
CREATE VIEW   CsReg (StudId, CrsCode, Semester) AS
SELECT     T.StudId, T. CrsCode, T.Semester
FROM      Transcript T
WHERE  T.CrsCode LIKE 'CS%' AND T.Semester='S2000'
```

Updating Views - Problem 1

INSERT INTO CsReg (*StudId*, *CrsCode*, *Semester*)
VALUES (1111, 'CSE305', 'S2000')

- **Question**: What value should be placed in attributes of underlying table that have been projected out (e.g., *Grade*)?
- **Answer**: NULL (assuming null allowed in the missing attribute) or DEFAULT

Updating Views - Problem 2

INSERT INTO CsReg (*StudId*, *CrsCode*, *Semester*)
VALUES (1111, 'ECO105', 'S2000')

- **Problem**: New tuple not in view
- **Solution**: Allow insertion (assuming the WITH CHECK OPTION clause has not been appended to the CREATE VIEW statement)

Updating Views - Problem 3

- Update to the view might not *uniquely* specify the change to the base table(s) that results in the desired modification of the view

```
CREATE VIEW ProfDept (PrName, DeName) AS
SELECT  P.Name, D.Name
FROM  Professor P, Department D
WHERE  P.DeptId = D.DeptId
```

Updating Views - Problem 3 (con't)

- Tuple <Smith, CS> can be deleted from ProfDept by:
  - Deleting row for Smith from Professor (but this is inappropriate if he is still at the University)
  - Deleting row for CS from Department (not what is intended)
  - Updating row for Smith in Professor by setting *DeptId* to null (seems like a good idea)

Updating Views - Restrictions

- Updatable views are restricted to those in which
  - No Cartesian product in FROM clause
  - no aggregates, GROUP BY, HAVING
  - …

For example, if we allowed:

```
CREATE VIEW AvgSalary (DeptId, Avg_Sal ) AS
    SELECT E.DeptId, AVG(E.Salary)
    FROM Employee E
    GROUP BY E.DeptId
```

then how do we handle:

```
UPDATE AvgSalary
    SET Avg_Sal = 1.1 * Avg_Sal
```