Northeastern University

DS 5110 – Lecture 6
SQL Part II

Roi Yehoshua

Agenda

Northeastern University

- Subqueries
- Common table expressions (CTEs)
- Recursive queries
- Query execution plans
- Views
- Stored procedures
- Transactions
- Indexes
- Working with SQL from Python

Subqueries

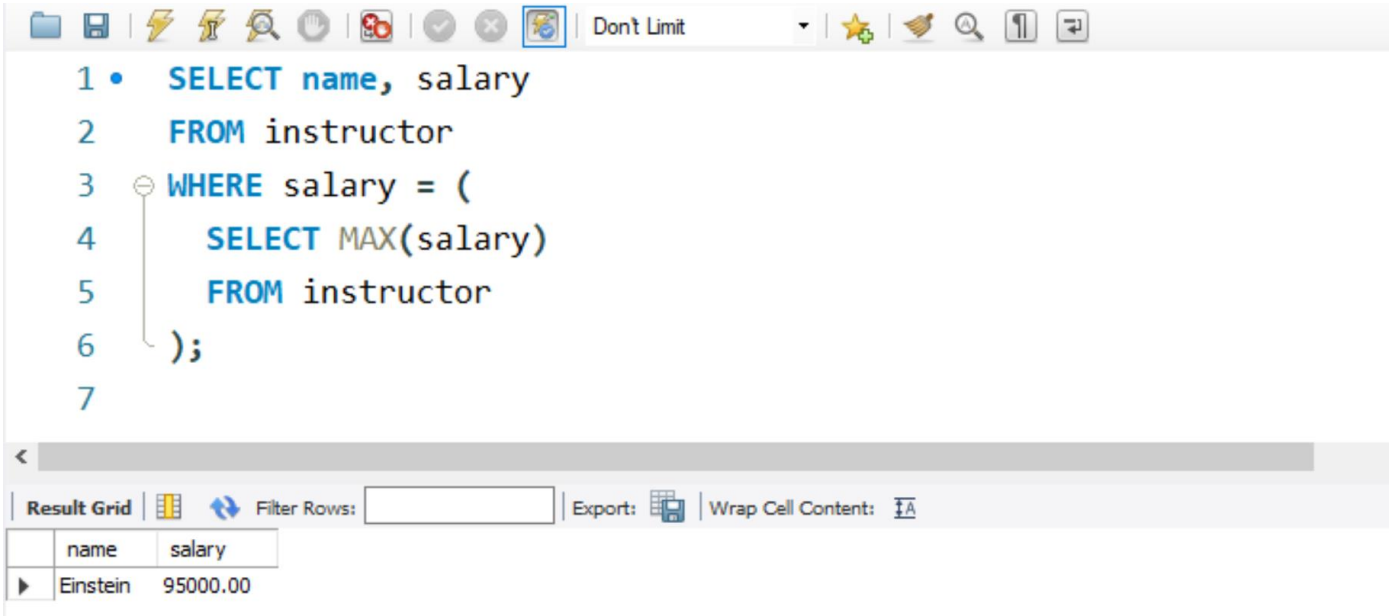
Northeastern University

- A subquery is a SELECT query nested inside another query
- Subqueries allow you to perform SQL operations in multiple steps
  - The subquery is executed before its parent query and its results are passed to the outer query
- A subquery may be nested inside
  - The SELECT clause
  - The FROM clause
  - The WHERE clause
  - The HAVING clause
- Subqueries must be enclosed inside brackets ()
- Subqueries can return an individual value or a list of records

Subqueries

Northeastern University

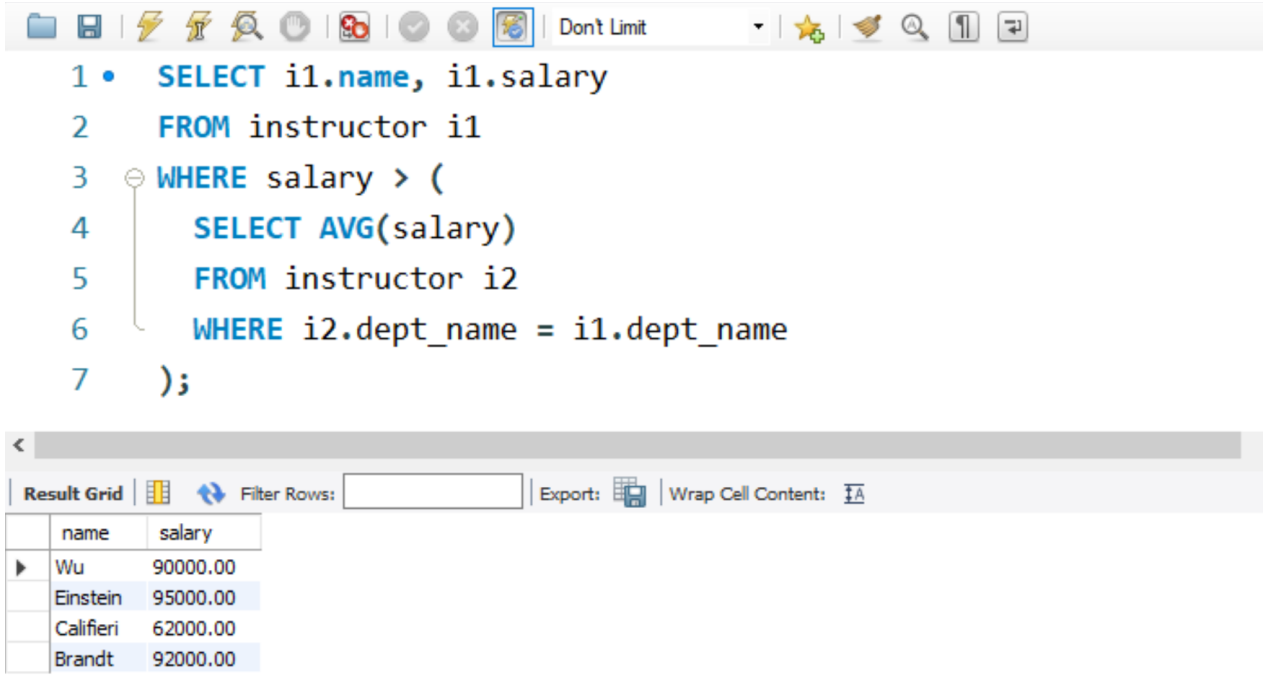
- Example: Find the name of the instructor with the highest salary
- This query cannot be written without sorting the whole table
- The way to write it with a subquery:



Correlated Subqueries

Northeastern University

- A correlated subquery references one or more columns from the outer query
- A correlated subquery is executed once for each candidate row in the outer query
- Example: Find all instructors whose salary is higher than the average salary **in their own department**



Multiple-Row Subqueries

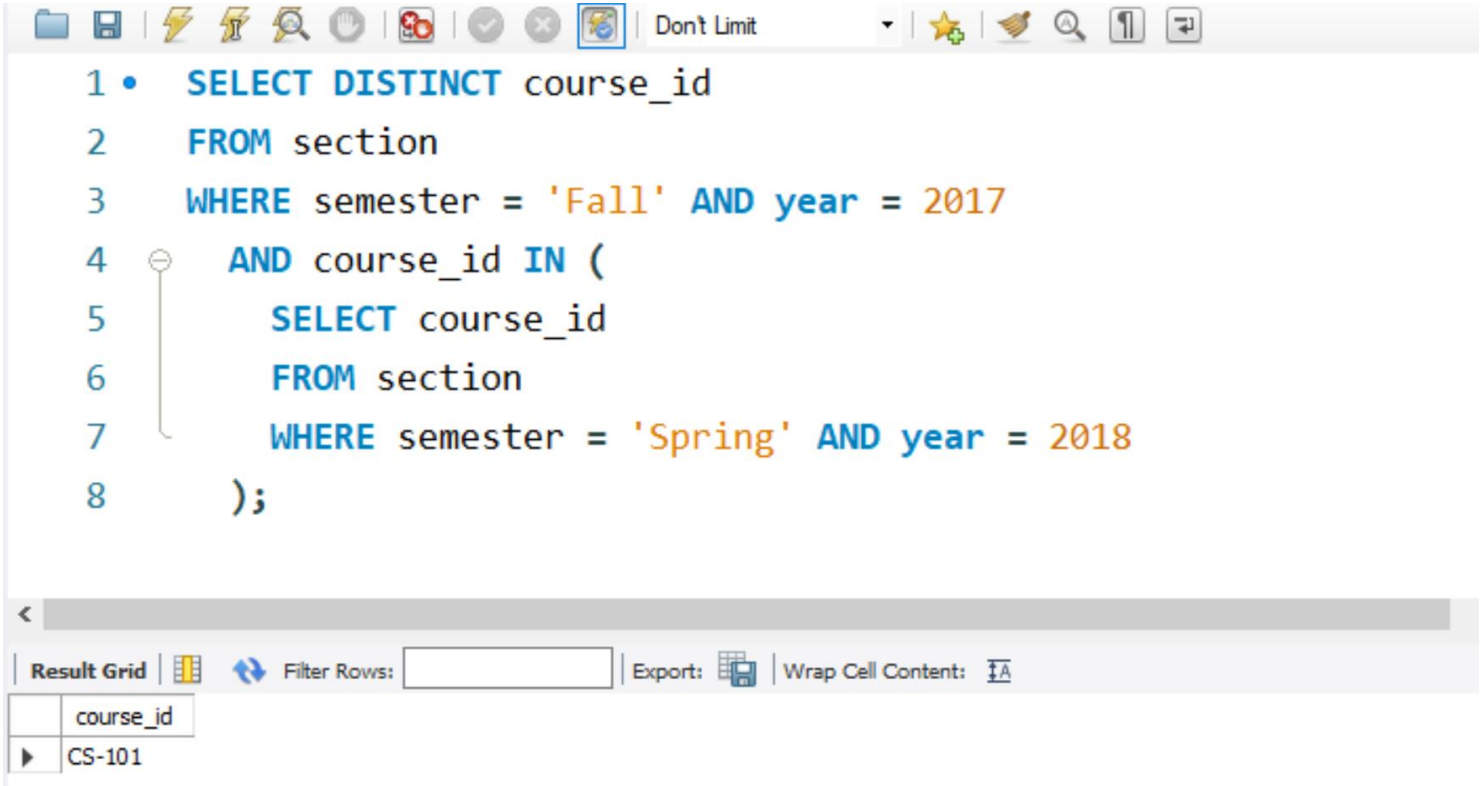
Northeastern University

- If the subquery returns more than one row, you cannot use it with the regular comparison operators like <, >, =
- There are 4 additional operators that allow to use the results of these subqueries:
  - IN / NOT IN
  - ALL
  - ANY
  - EXISTS / NOT EXISTS

The IN Operator

Northeastern University

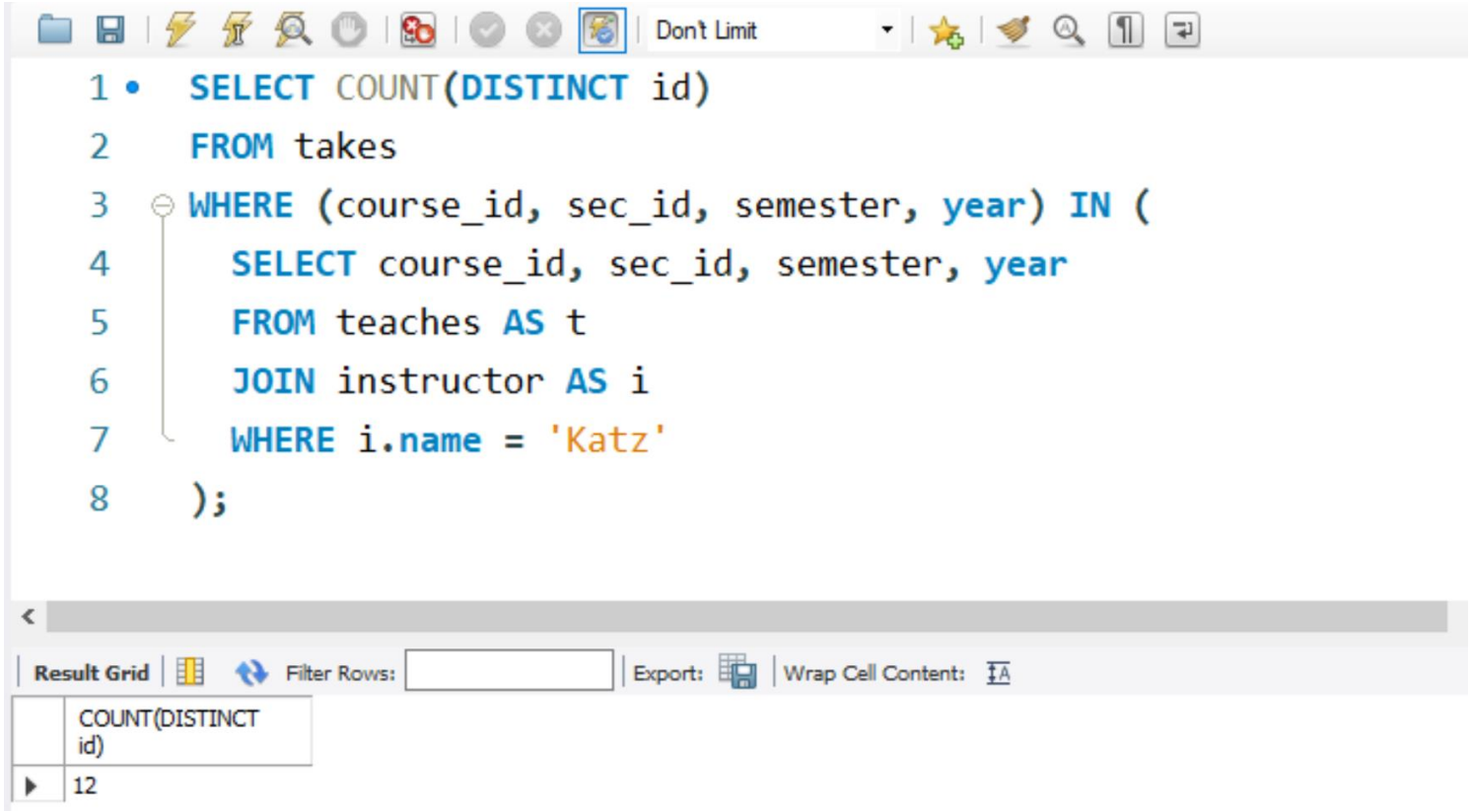
- Checks whether a column contains a value that is within the results of the subquery
- Example: Find all the courses that were offered both in Fall 2017 and Spring 2018



The IN Operator

Northeastern University

- You can compare more than one column with the results of the subquery
- Example: Find the total number of (distinct) students who have been taught by Katz



The NOT IN Operator

- Checks if the column doesn't contain any value within the results of the subquery
- Example: Find all the courses that were offered in Fall 2017 but not in Spring 2018

```
1 •  SELECT DISTINCT course_id
2    FROM section
3    WHERE semester = 'Fall' AND year = 2017
4      AND course_id NOT IN (
5        SELECT course_id
6        FROM section
7        WHERE semester = 'Spring' AND year = 2018
8      );
```

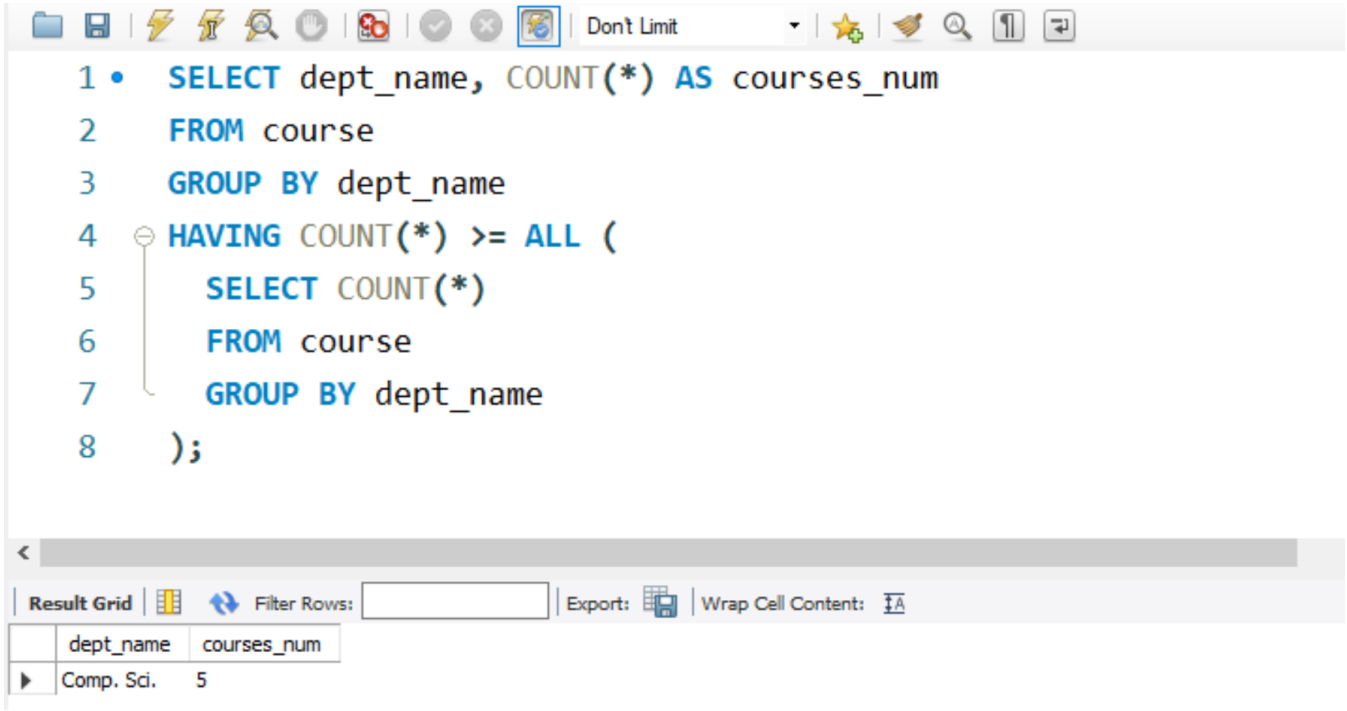
Result Grid | Filter Rows: | Export: | Wrap Cell Content:

course_id
CS-347
PHY-101

The ALL Operator

Northeastern University

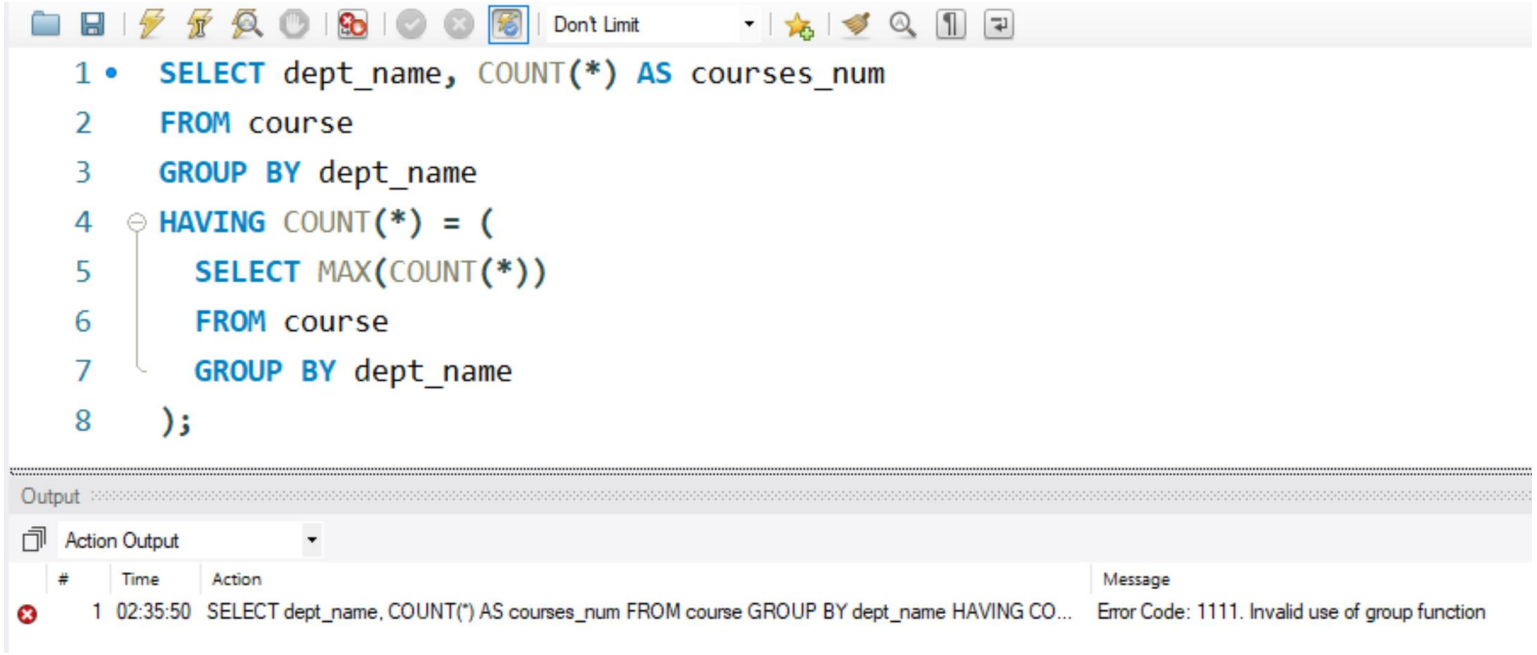
- The ALL operator returns true if **all** of the subquery results meet the condition
- Used in conjunction with one of the comparison operators (=, <>, >, >=, <, or <=)
- Example: Find the department with the highest number of courses



The ALL Operator

Northeastern University

- Note that the following attempt to write the same query won't work



- You cannot layer aggregate functions on top of each other in the same SELECT

The ANY Operator

Northeastern University

- The ANY operator returns true if **any** of the subquery results meets the condition
- Used in conjunction with one of the comparison operators (=, <>, >, >=, <, or <=)
- Example: Find all the instructors whose salary is higher than any of the instructors in the Biology department

```
1 • SELECT name, salary, dept_name
2   FROM instructor
3   WHERE salary > ANY (
4     SELECT salary
5     FROM instructor
6     WHERE dept_name = 'Biology'
7   );
```

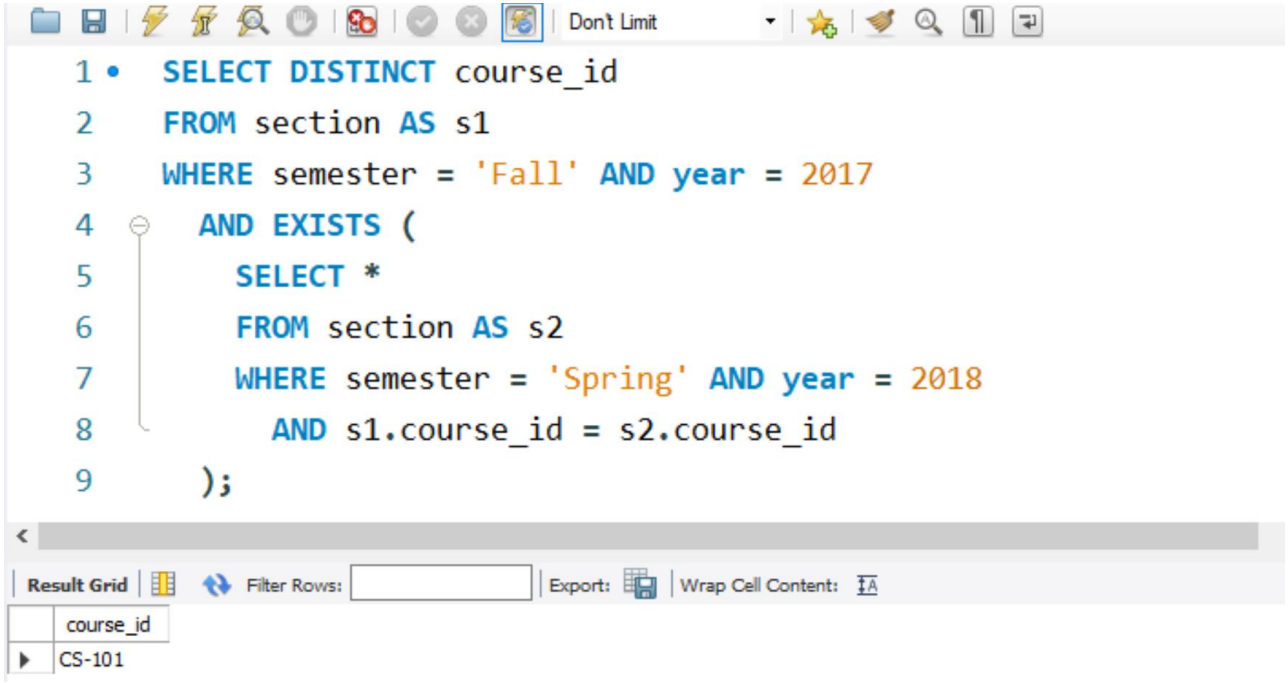
Result Grid | Filter Rows: | Export: | Wrap Cell Content:

name	salary	dept_name
Wu	90000.00	Finance
Einstein	95000.00	Physics
Gold	87000.00	Physics
Katz	75000.00	Comp. Sci.
Singh	80000.00	Finance
Brandt	92000.00	Comp. Sci.
Kim	80000.00	Elec. Eng.

The EXISTS Operator

Northeastern University

- The EXISTS operator tests the existence of any result in the subquery
- It returns true if the subquery returns one or more rows
- For example, another way to write the query “Find all the courses that were offered both in Fall 2017 and Spring 2018”



The NOT EXISTS Operator

Northeastern University

- You can also use NOT EXISTS to check if the subquery didn't return any rows
- Example: Find instructors who haven't taught any course

```
1 •  SELECT id, name
2    FROM instructor AS i
3    WHERE NOT EXISTS (
4      SELECT *
5      FROM teaches AS t
6      WHERE t.id = i.id
7    );
```

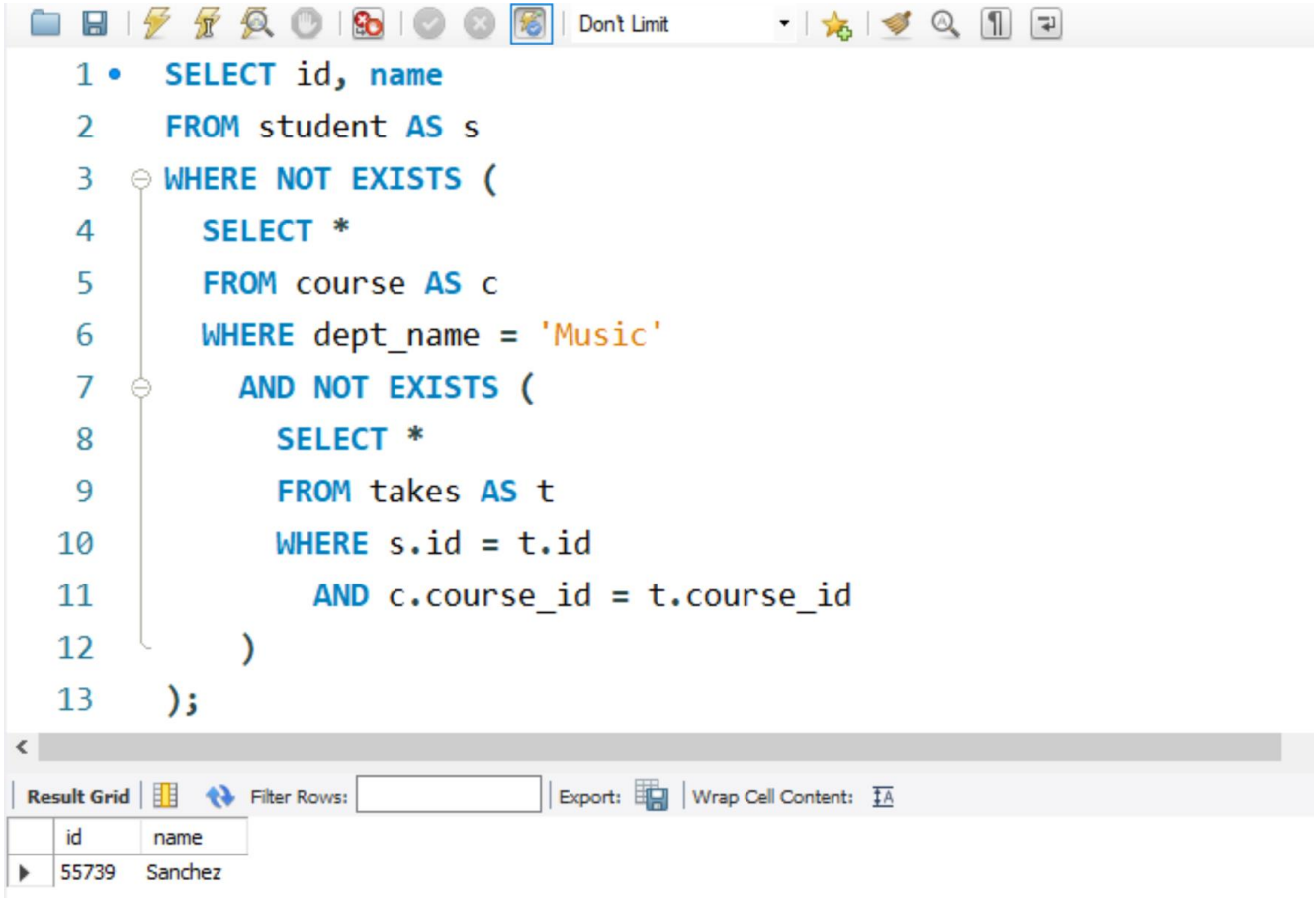
Result Grid | Filter Rows: | Export: | Wrap Cell Content:

id	name
33456	Gold
58583	Califieri
76543	Singh

The NOT EXISTS Operator

Northeastern University

- Example: Find students who have taken all the courses in Music



```
1  SELECT id, name
2  FROM student AS s
3  WHERE NOT EXISTS (
4    SELECT *
5    FROM course AS c
6    WHERE dept_name = 'Music'
7      AND NOT EXISTS (
8        SELECT *
9        FROM takes AS t
10       WHERE s.id = t.id
11         AND c.course_id = t.course_id
12     )
13 );
```

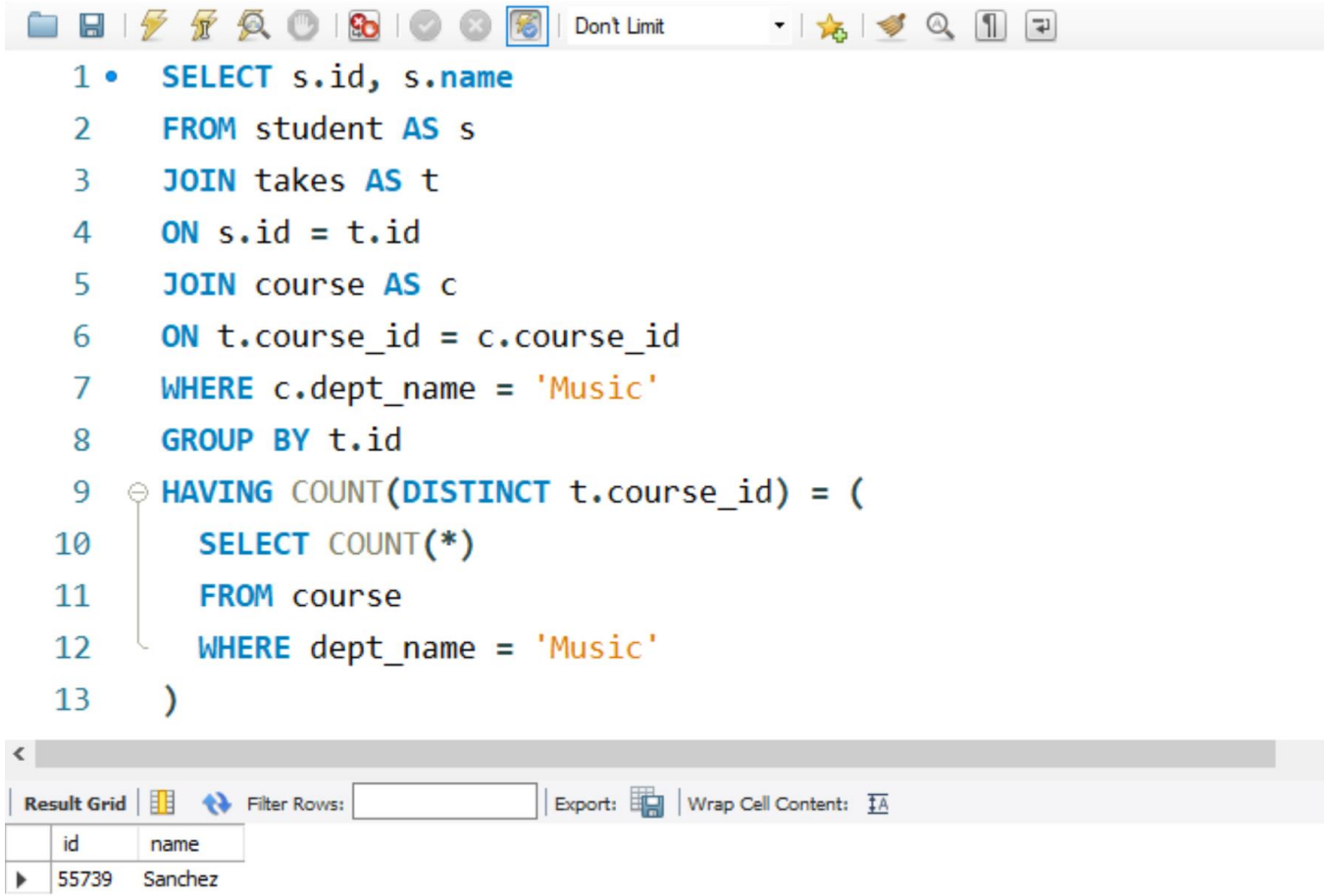
Result Grid | Filter Rows: | Export: | Wrap Cell Content:

id	name
55739	Sanchez

The NOT EXISTS Operator

Northeastern University

- Another way to write the same query:



```
1 •  SELECT s.id, s.name
2    FROM student AS s
3    JOIN takes AS t
4    ON s.id = t.id
5    JOIN course AS c
6    ON t.course_id = c.course_id
7    WHERE c.dept_name = 'Music'
8    GROUP BY t.id
9    HAVING COUNT(DISTINCT t.course_id) = (
10     SELECT COUNT(*)
11     FROM course
12     WHERE dept_name = 'Music'
13   )
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

id	name
55739	Sanchez

Subqueries in the FROM Clause

- SQL allows you to write subqueries in the FROM clause
  - In MySQL, the subquery must be given a name using the AS keyword
- The subquery results can be joined with other tables or other subqueries
- Example: Find the highest number of courses offered in any given department

```
1 •  SELECT MAX(courses_in_dept.courses_num)
2    FROM (
3      SELECT COUNT(*) courses_num
4      FROM course
5      GROUP BY dept_name
6    ) AS courses_in_dept;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

MAX(courses_in_dept.courses_num)
5

Updates with Subqueries

Northeastern University

- You can also use subqueries in UPDATE and DELETE statements
- Example: recompute and update the tot_cred value for each student

```
UPDATE student AS s
SET tot_cred = (
  SELECT SUM(c.credits)
  FROM takes AS t
  JOIN course AS c
  WHERE t.course_id = c.course_id
    AND s.id = t.id
    AND t.grade IS NOT NULL
    AND t.grade <> 'F'
);
```

  - Sets tot_cred to NULL for students who have not taken any course
  - Instead of SUM(credits) can use COALESCE(SUM(credits), 0)
    - This would return SUM(credits) if it is not NULL, and 0 otherwise

Common Table Expressions (CTEs)

Northeastern University

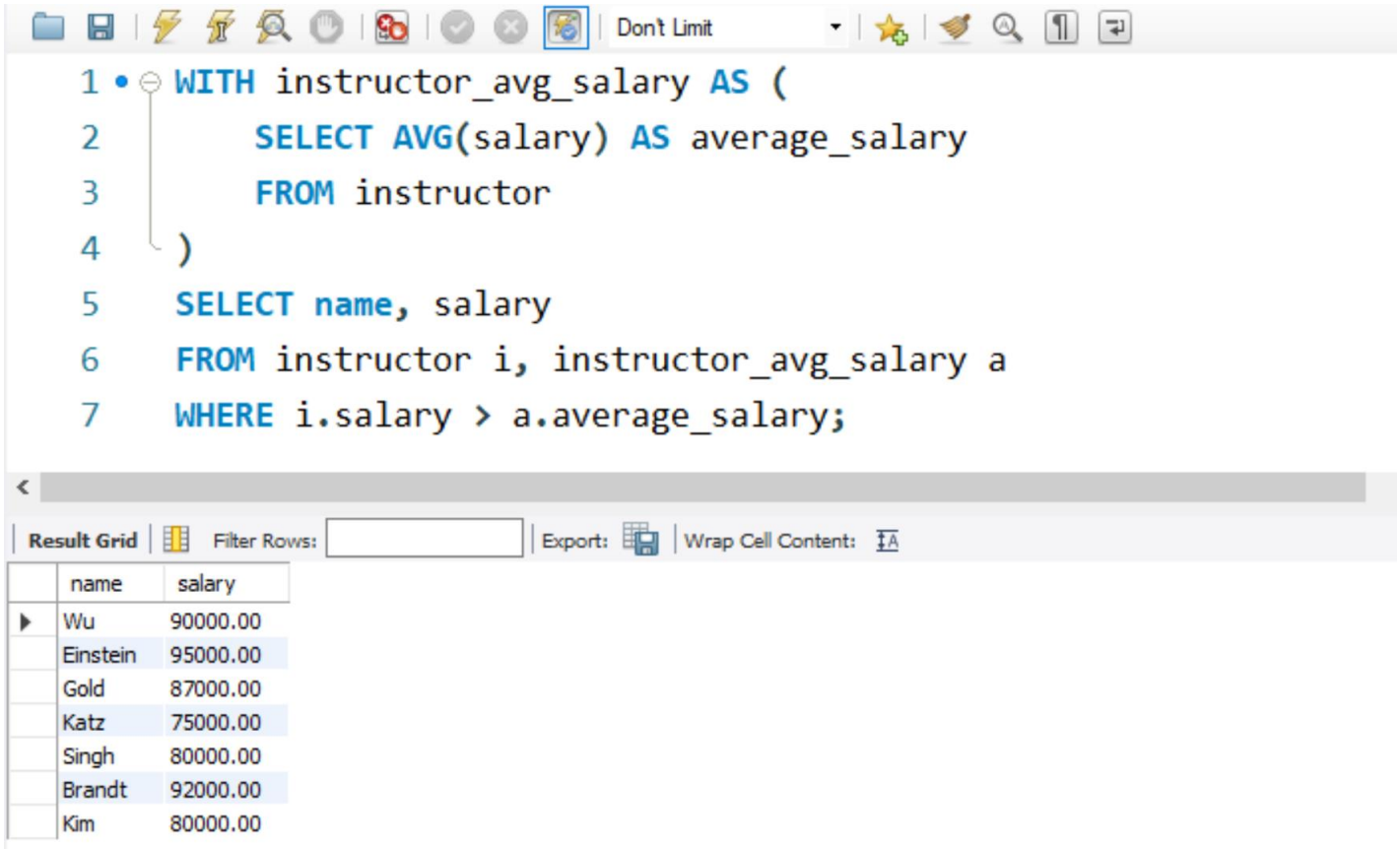
- CTE is a named temporary result set that only exists for the duration of the query
- CTEs are useful for breaking down complex queries into simpler parts
- Basic syntax of CTE:

```
WITH cte_name (column_name1, column_name2, ...)
AS (
    -- Your SQL query here
)
-- Your main SQL statement using the CTE here
```

Common Table Expressions (CTEs)

Northeastern University

- Example: find all the instructors that earn more than the average salary



```
1  WITH instructor_avg_salary AS (
2      SELECT AVG(salary) AS average_salary
3      FROM instructor
4  )
5  SELECT name, salary
6  FROM instructor i, instructor_avg_salary a
7  WHERE i.salary > a.average_salary;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

name	salary
Wu	90000.00
Einstein	95000.00
Gold	87000.00
Katz	75000.00
Singh	80000.00
Brandt	92000.00
Kim	80000.00

Common Table Expressions (CTEs)

Northeastern University

- Show the distribution of number of courses taken by students
  - i.e., how many students took only one course, how many students took two courses, etc.

```sql
WITH courses_per_student AS (
    -- Count the number of courses each student took
    SELECT id, COUNT(DISTINCT course_id) AS num_courses
    FROM takes
    GROUP BY id
)

-- Get the distribution of courses taken by students
SELECT num_courses, COUNT(id) AS num_students
FROM courses_per_student
GROUP BY num_courses
ORDER BY num_courses;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

num_courses	num_students
1	5
2	6
4	1

Recursive Queries

Northeastern University

- A recursive query is a query that refers back to itself
- Used to deal with hierarchical or tree-structured data
- A recursive query is written using a CTE that consists of two subqueries:
  - The base (non-recursive) query provides the starting point of the recursion
  - The recursive query which has a reference to the CTE itself

Recursive Queries

Northeastern University

- Example: Find for each course all its prerequisites (both direct and indirect)

```sql
WITH RECURSIVE rec_prereq(course_id, prereq_id) AS (
    -- Anchor member: The base case, where we select the initial prerequisites
    SELECT course_id, prereq_id
    FROM prereq
    UNION
    -- Recursive member: Select the prerequisites of the previous rows
    SELECT p.course_id, p.prereq_id
    FROM prereq AS p
    JOIN rec_prereq AS rec_p
    ON p.course_id = rec_p.prereq_id -- Join with the CTE itself!
)
-- Now, select the results from the CTE
SELECT * FROM rec_prereq;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

course_id	prereq_id
BIO-301	BIO-101
BIO-399	BIO-101
CS-190	CS-101
CS-315	CS-101
CS-319	CS-101
CS-347	CS-101
EE-181	PHY-101

Summary: SQL Execution Order

Northeastern University

1. FROM
2. ON
3. JOIN
4. WHERE
5. GROUP BY
6. WITH CUBE or WITH ROLLUP
7. HAVING
8. SELECT
9. DISTINCT
10. ORDER BY
11. TOP

Query Execution Plans

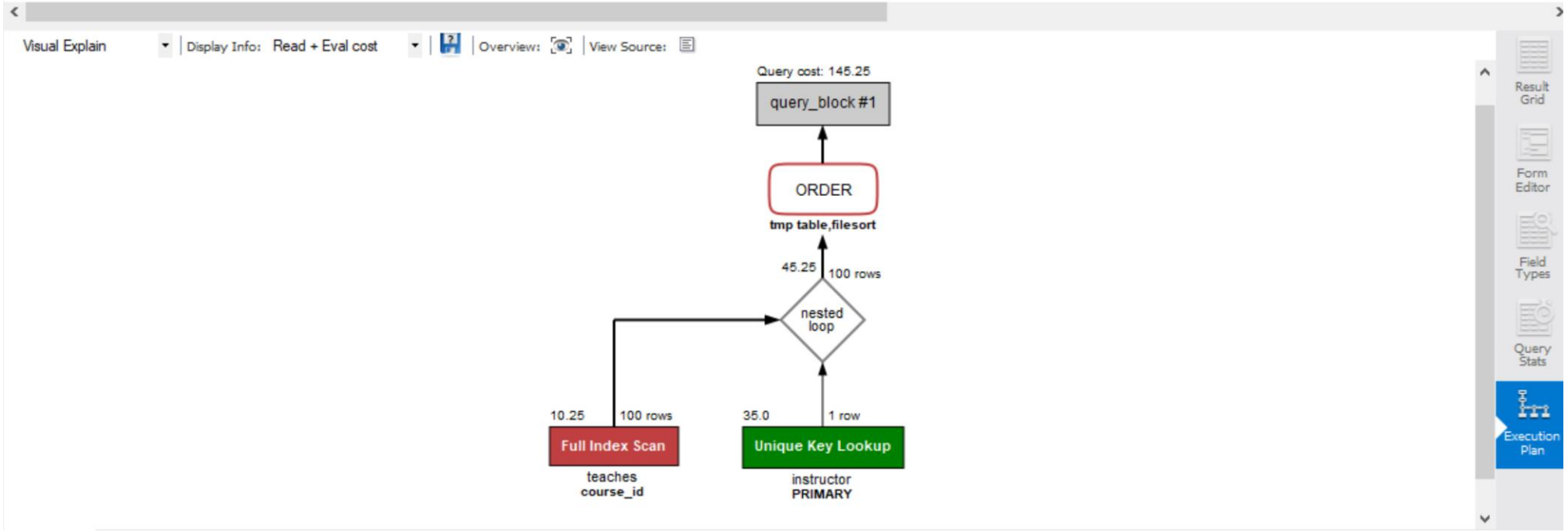
Northeastern University

- A **query execution plan** is an ordered set of steps used to access data in the database
  - Defines which algorithm to use for each operation, and how to coordinate the operations
- When a query is submitted to the database, the **query optimizer** evaluates the different plans for executing the query and returns what it considers the best option
  - Cost difference between execution plans for a query can be enormous
- Estimation of plan cost is based on:
  - Statistical information about tables
    - e.g., number of rows, number of distinct values in a column
  - Cost of different algorithms (e.g., join vs. subquery)
    - Also computed using statistics
- Sometimes we need to manually examine and tune the optimizer plans

Query Execution Plans

- To view the execution plan, click on Execution Plan next to the query results

```
1 • SELECT *
2   FROM instructor
3   JOIN teaches
4   ON instructor.id = teaches.id
5   ORDER BY name;
```



Views

Northeastern University

- A view is a virtual table based on the results set of an SQL statement
- Reasons to use views:
  - Data security – allows you to hide certain data from view of certain users
    - e.g., some users only need to know the instructor IDs and names, but not their salaries
  - Hiding complexity of the underlying tables
- A view is created with the CREATE VIEW statement:

```
CREATE VIEW view_name AS
SELECT column1, column2, ...
FROM table_name
WHERE condition;
```

- A view always shows up-to-date data
  - The database engine recreates the view whenever it is used in a query

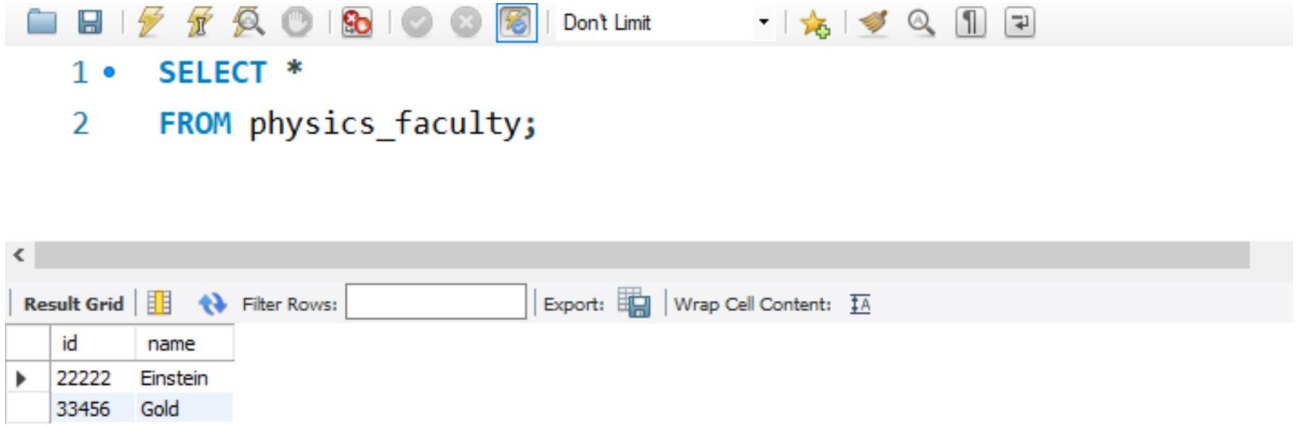
Views

Northeastern University

- Example: A view that shows all Physics instructors without their salary

```
CREATE VIEW physics_faculty AS
SELECT id, name
FROM instructor
WHERE dept_name = 'Physics';
```

- We can query the view above as follows:

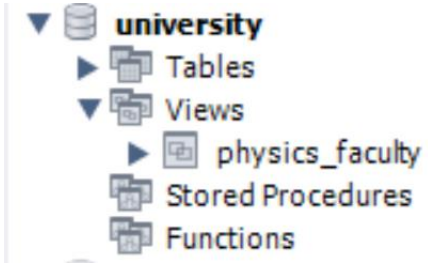


- View names may appear in a query anywhere a table name may appear

Views

Northeastern University

- You can see all the available views in the database under the View folder:



- To update a view use the CREATE OR REPLACE VIEW command:

```
CREATE OR REPLACE VIEW view_name AS
SELECT column1, column2, ...
FROM table_name
WHERE condition;
```

- To delete a view use the DROP VIEW command:

```
DROP VIEW view_name;
```

Updatable Views

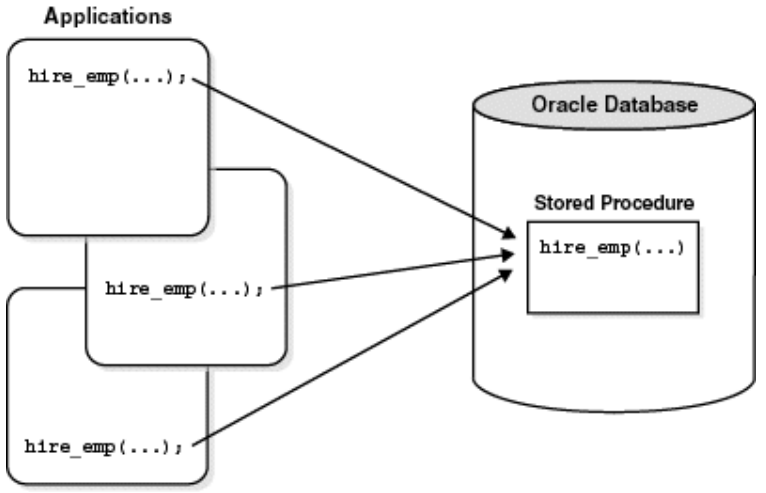
Northeastern University

- Some views are updatable
  - i.e., you can use them in statements like INSERT or UPDATE to update the underlying tables
- For a view to be updatable, there must be a one-to-one relationship between the rows in the view and the rows in the underlying table
- More specifically, the view isn't updatable if it contains any of the following:
  - Aggregate functions like MIN() or AVG()
  - DISTINCT
  - GROUP BY or HAVING
  - Certain joins are not allowed
  - Subquery in SELECT
  - Subquery in the WHERE clause that refers to a table in the FROM clause
  - UNION or UNION ALL

Stored Procedures

Northeastern University

- A **stored procedure** (SP) is a group of SQL statements that is stored in the database
  - You can also pass parameters to a stored procedure
- The benefits of using stored procedures:
  - Reusable code that can be used by different applications
  - Allows a single point of change in case the business rules change
  - Faster execution
  - Reduce the network traffic
  - More secure than ad-hoc queries



Creating a Stored Procedure

Northeastern University

- The syntax for creating a new stored procedure in MySQL:

```
DELIMITER $$

CREATE PROCEDURE procedure_name (
  IN | OUT | INOUT parameter1 datatype,
  IN | OUT | INOUT parameter2 datatype,
  ...
)
BEGIN
  -- SQL statements
END $$
DELIMITER ;
```

Redefine the delimiter temporarily so that ; can be used to separate statements inside the procedure

Use ; in the body of the stored procedure

Use the delimiter $$ to end the stored procedure

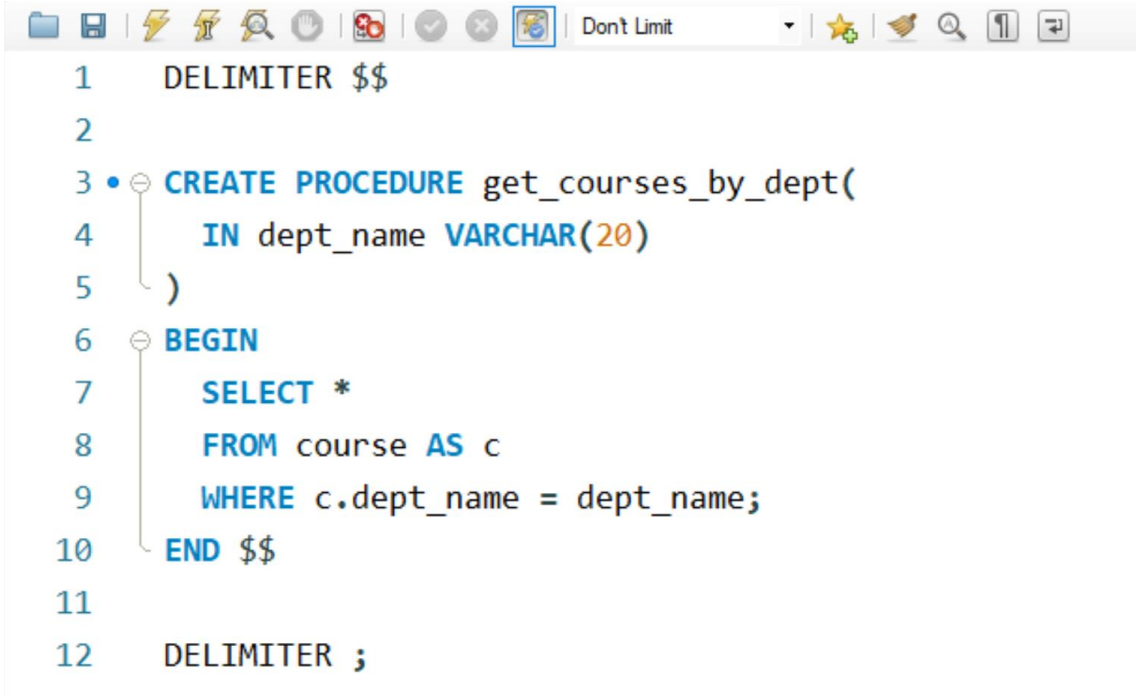
Change the default delimiter back to ;

- Parameter modes:
  - **in** (default) – the parameter's value is passed into the SP and cannot be changed inside the SP
  - **out** – the parameter's value is passed back to the calling program, must be a variable
  - **inout** – the parameter's value is passed to the SP and a new value can be assigned to it

Creating a Stored Procedure

Northeastern University

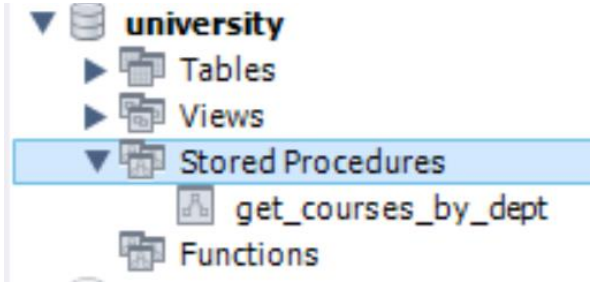
- Example: a stored procedure that returns all courses in a given department



Creating a Stored Procedure

Northeastern University

- You can view your procedure under the Stored Procedures folder of the schema



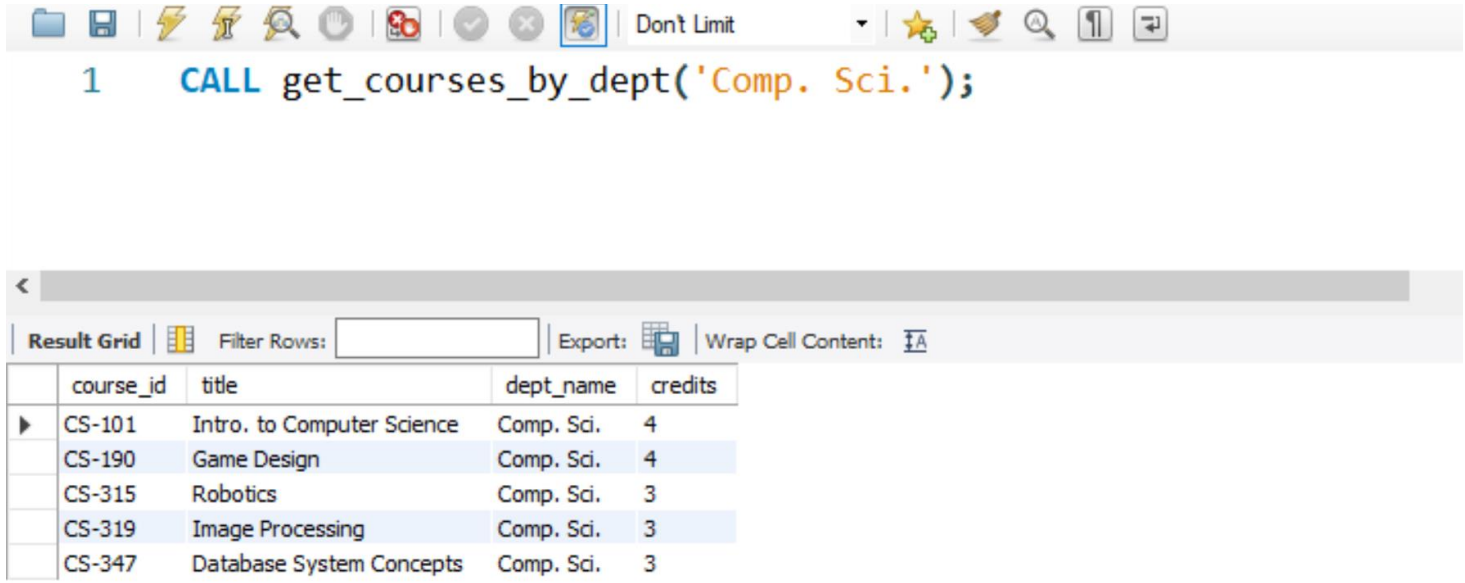
Calling a Stored Procedure

Northeastern University

- To execute the store procedure, you use the CALL keyword:

```
CALL procedure_name(parameter1, parameter2, ...);
```

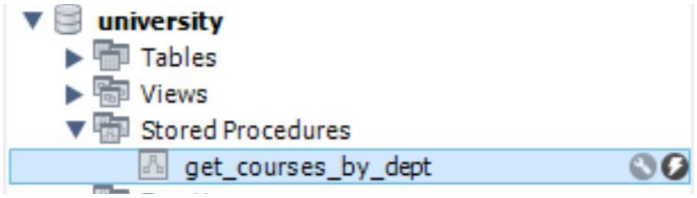
- For example, let's call the get_courses_by_dept with Comp. Sci. as the argument:

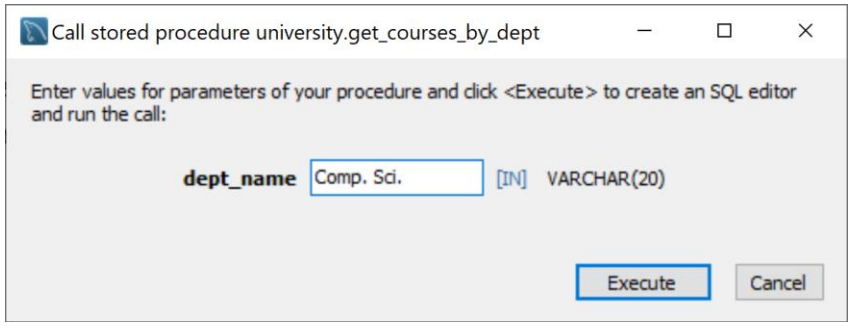


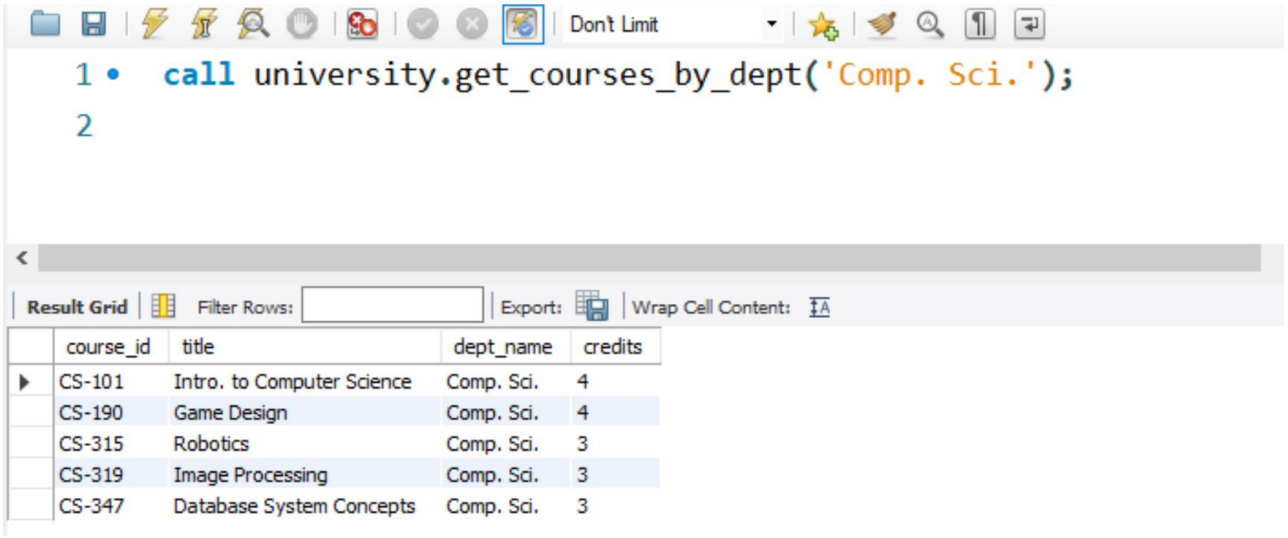
course_id	title	dept_name	credits
CS-101	Intro. to Computer Science	Comp. Sci.	4
CS-190	Game Design	Comp. Sci.	4
CS-315	Robotics	Comp. Sci.	3
CS-319	Image Processing	Comp. Sci.	3
CS-347	Database System Concepts	Comp. Sci.	3

Calling a Stored Procedure

- You can also execute the SP by clicking on the Execute button next to its name





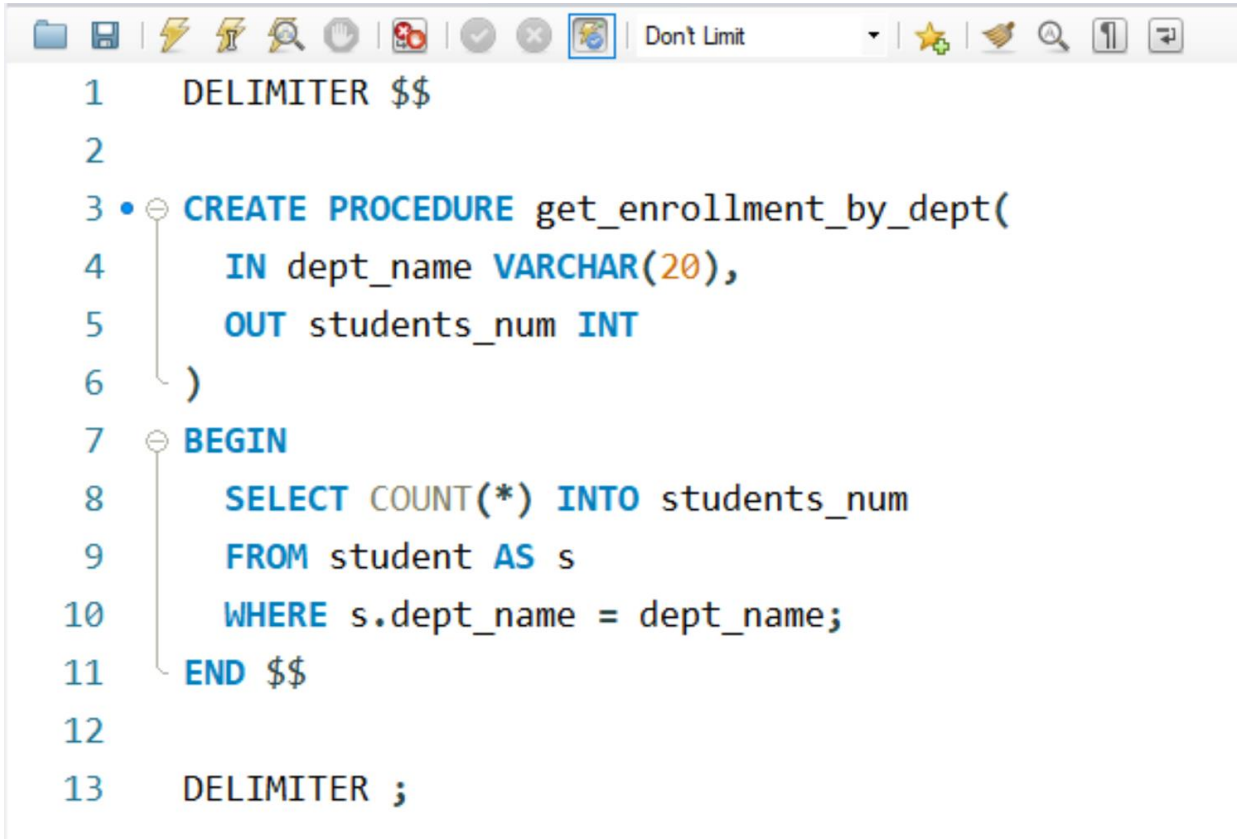


course_id	title	dept_name	credits
CS-101	Intro. to Computer Science	Comp. Sci.	4
CS-190	Game Design	Comp. Sci.	4
CS-315	Robotics	Comp. Sci.	3
CS-319	Image Processing	Comp. Sci.	3
CS-347	Database System Concepts	Comp. Sci.	3

OUT Parameter

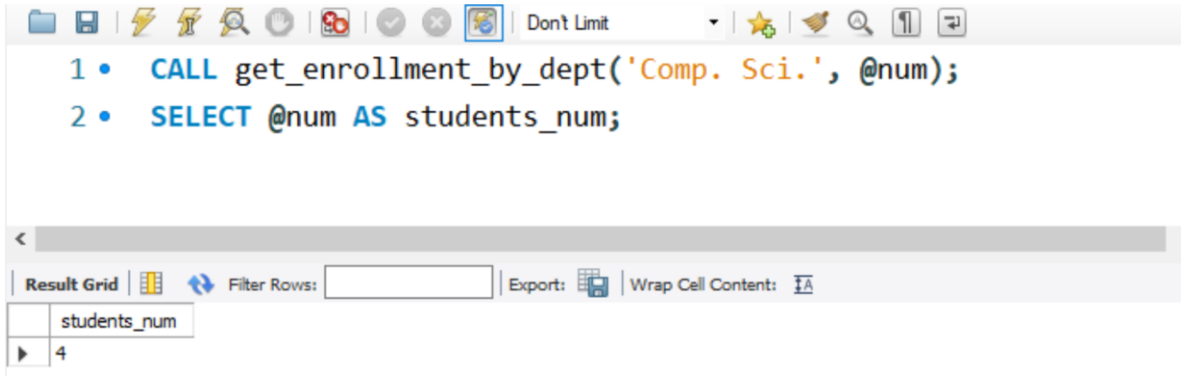
Northeastern University

- A stored procedure that returns the number of students in a given department



OUT Parameter

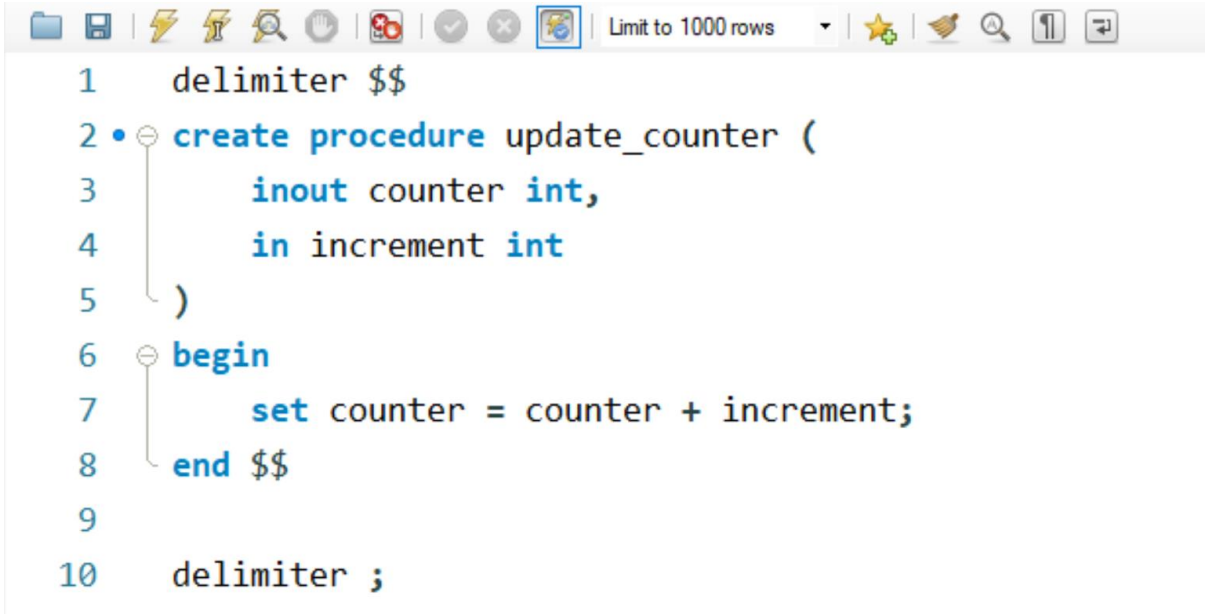
- To call a SP with an OUT parameter you need to pass a **session variable** to it
  - This variable will receive the return value from the procedure
- Session variables are variables that start with @
  - Don’t require any declaration
  - Can be used inside any SQL query or statement
  - Exist until the end of the current session
  - Assignments to the variable are performed using a SET statement
  - To display the value of a variable use the SELECT statement



InOut Parameter Example

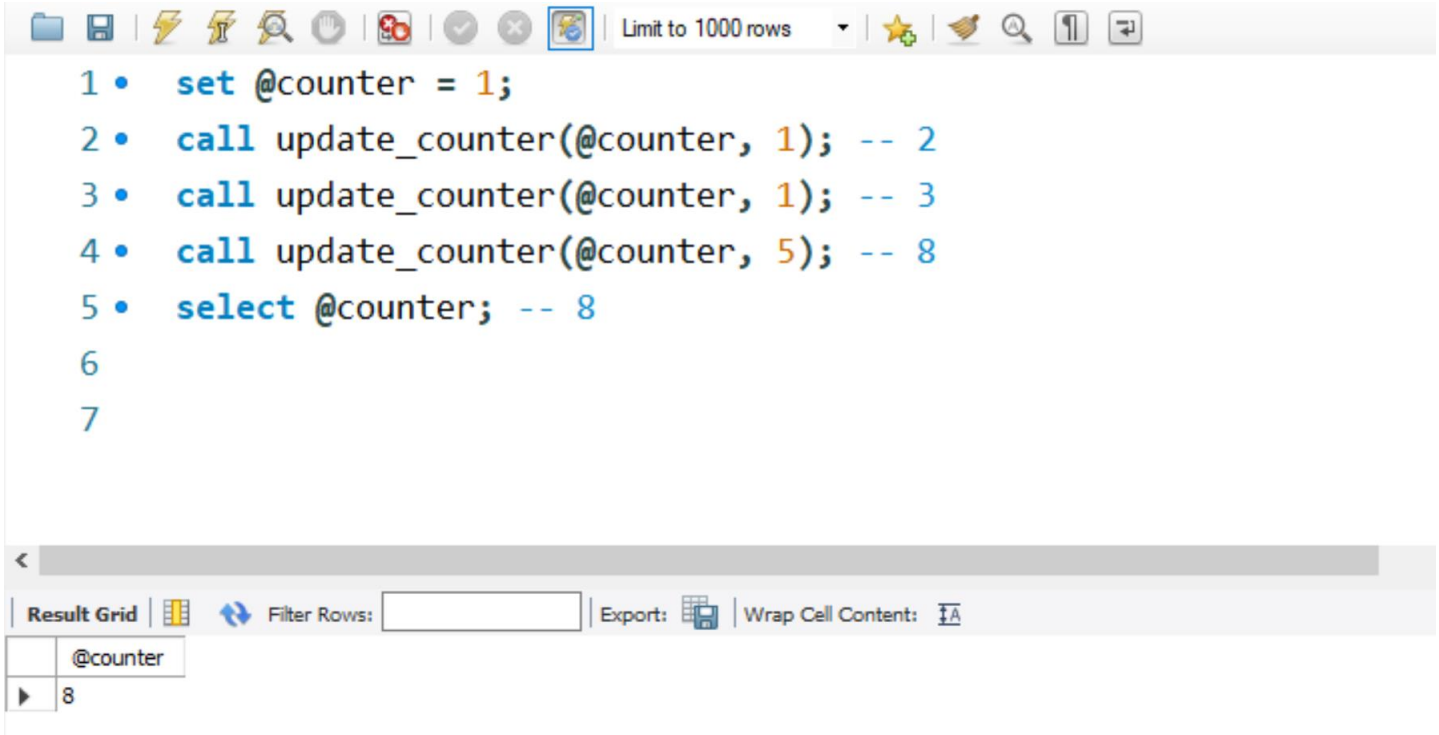
Northeastern University

- The following example demonstrates how to use an **inout** parameter in a SP:



InOut Parameter Example

- The following statements illustrate how to call the update_counter SP:



Programming Language Constructs

Northeastern University

- Stored procedures support programming language constructs like ifs and loops:

```
IF boolean expression THEN
  statement_list
[ELSEIF boolean expression THEN
  statement_list]
...
[ELSE statement_list]
END IF;

WHILE boolean expression DO
  statement_list
END WHILE;

REPEAT
  statement_list
UNTIL boolean expression
END REPEAT;
```

Programming Language Constructs

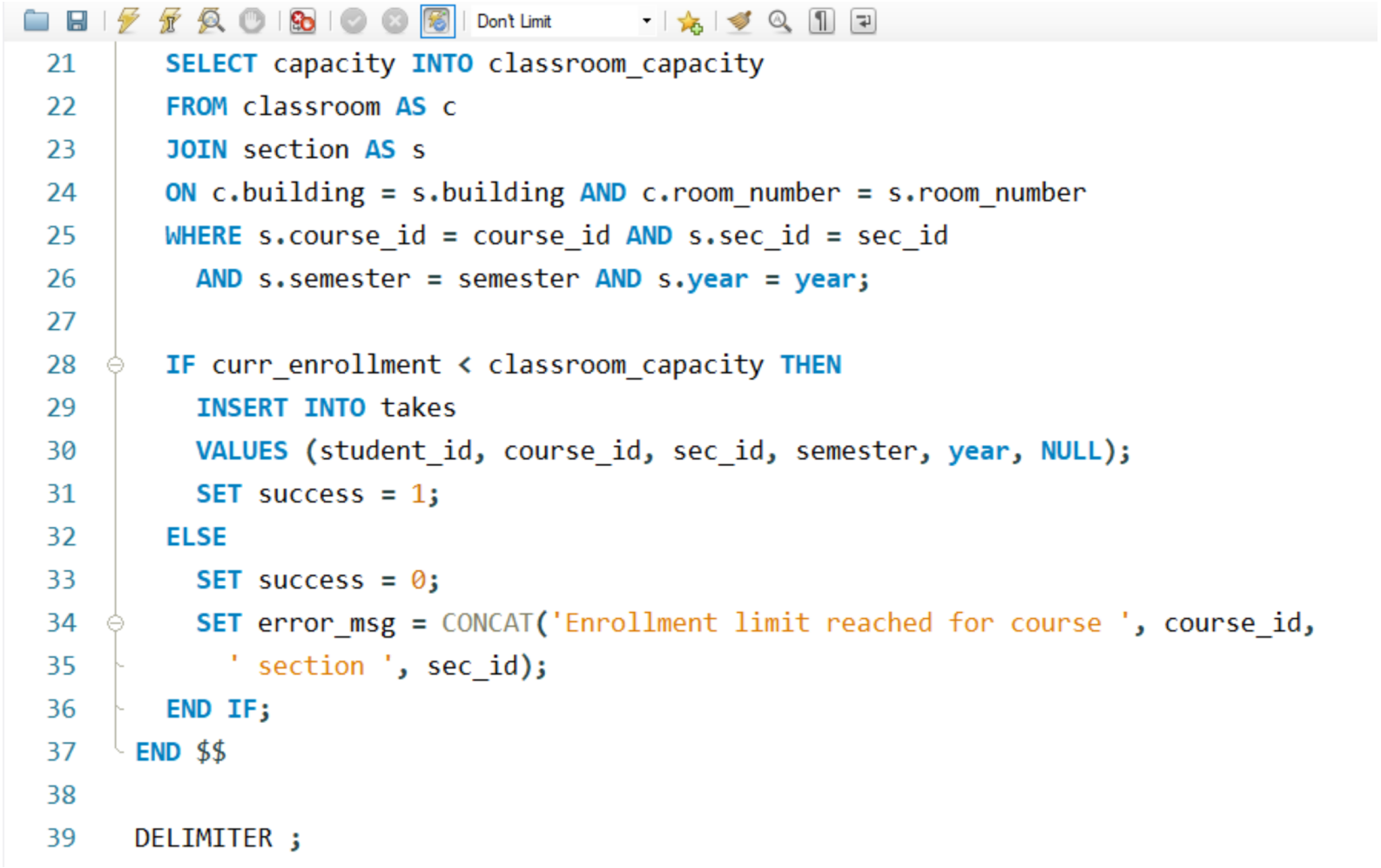
Northeastern University

- The following procedure registers a student to a course section only if there is enough room left in the classroom allocated for this section

```
DELIMITER $$

CREATE PROCEDURE register_student(
  IN student_id VARCHAR(5),
  IN course_id VARCHAR(8),
  IN sec_id VARCHAR(8),
  IN semester VARCHAR(6),
  IN year NUMERIC(4, 0),
  OUT success INT,
  OUT error_msg VARCHAR(100)
)
BEGIN
  DECLARE curr_enrollment INT;
  DECLARE classroom_capacity INT;

  SELECT COUNT(*) INTO curr_enrollment
  FROM takes AS t
  WHERE t.course_id = course_id AND t.sec_id = sec_id
    AND t.semester = semester AND t.year = year;
```

Programming Language Constructs

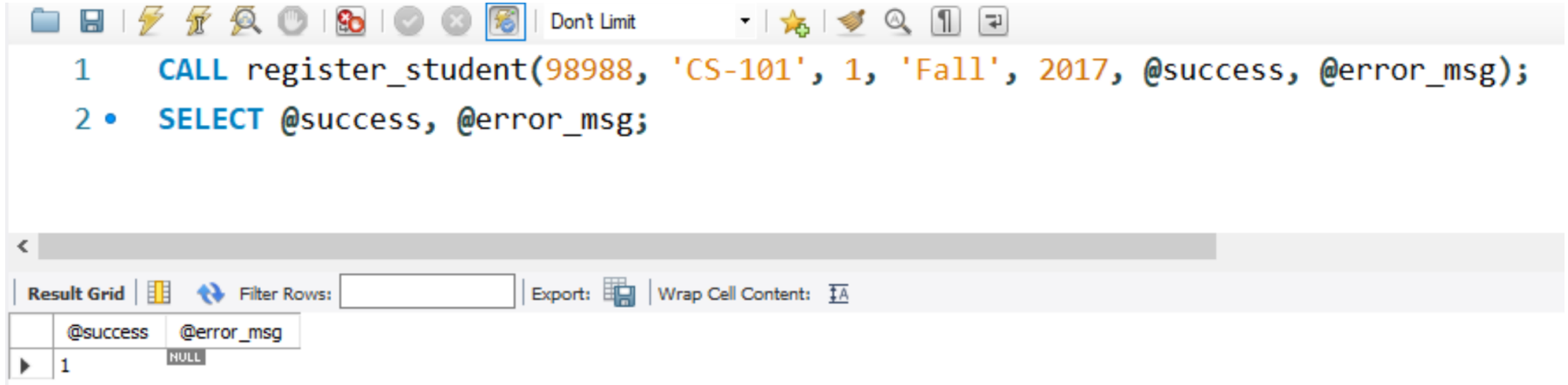
Northeastern University



Programming Language Constructs

Northeastern University

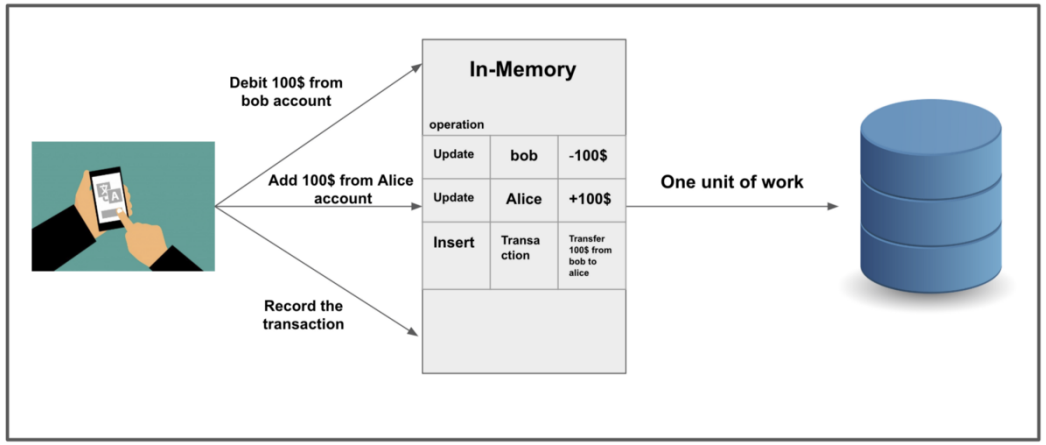
- Calling the procedure:



Transactions

Northeastern University

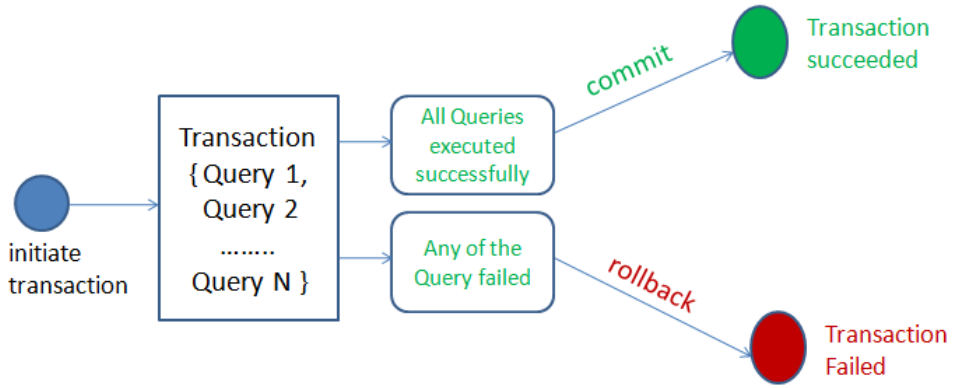
- A transaction is a sequence of SQL statements that represents a **single unit of work**
- For example, consider a Bank amount transfer, which involves two operations:
  - Withdrawal of money from account *A*
  - Deposit money to Account *B*
- If the system crashes after subtracting the amount from *A* but before adding it to *B*, the bank balances will be inconsistent



Transactions

Northeastern University

- To define a new transaction, use the START TRANSACTION statement
- All the subsequent SQL statements will belong to the transaction
- The transaction must end with one of the following statements:
  - COMMIT: The updates performed by the transaction become permanent in the database
  - ROLLBACK: All updates performed by the transaction are undone



Transactions

Northeastern University

- In MySQL, transactions need to be written inside a stored procedure
  - So you can automatically rollback the transaction whenever an error occurs
  - The following statement needs to be written in the beginning of the procedure:

```
DECLARE EXIT HANDLER FOR SQLEXCEPTION ROLLBACK;
```

- The following procedure updates a student's grade in a course
- This procedure needs to update both the takes table and the student's tot_cred
  - These two updates should be performed in a single transaction

Transaction Example

Northeastern University

```
1   DELIMITER $$
2
3   CREATE PROCEDURE update_grade(
4     student_id VARCHAR(5),
5     course_id VARCHAR(8),
6     sec_id VARCHAR(8),
7     semester VARCHAR(6),
8     year DECIMAL(4),
9     grade VARCHAR(2)
10  )
11  BEGIN
12    -- If an error occurs, the transaction will be rolled back automatically
13    DECLARE EXIT HANDLER FOR SQLEXCEPTION ROLLBACK;
14
15    START TRANSACTION;
16    -- Update the student's grade in the course
17    UPDATE takes AS t
18    SET t.grade = grade
19    WHERE t.id = student_id AND t.course_id = course_id AND t.sec_id = sec_id
20      AND t.semester = semester AND t.year = year;
```

Transaction Example

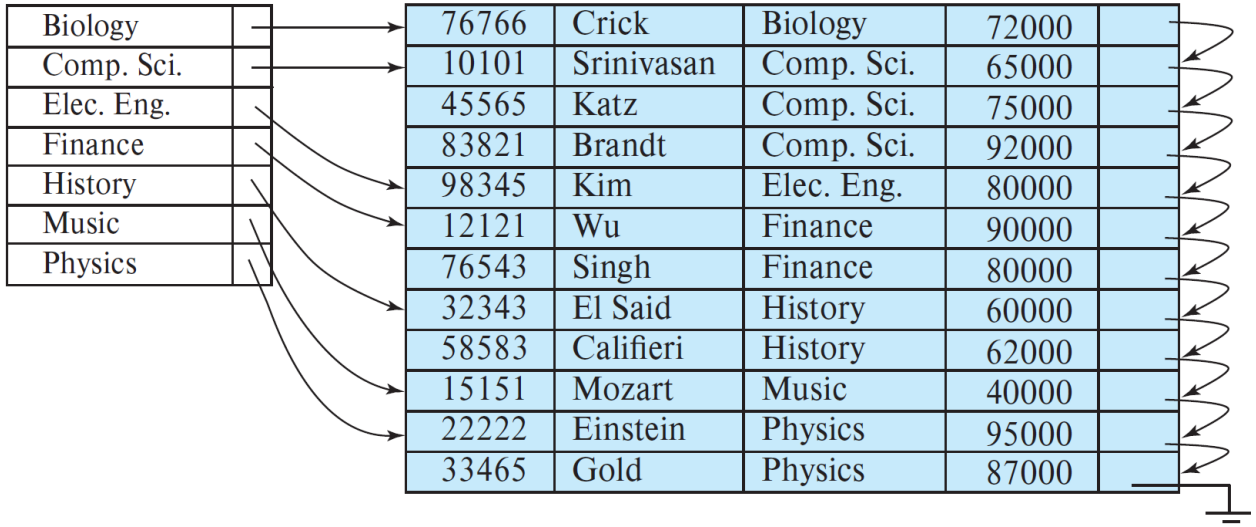
Northeastern University

```
22    -- Update the student's total credits
23    UPDATE student AS s
24    SET s.tot_cred = s.tot_cred + (
25      SELECT credits
26      FROM course AS c
27      WHERE c.course_id = course_id
28    )
29    WHERE s.id = student_id;
30
31    COMMIT;
32  END $$
33
34  DELIMITER ;
```

Indexes

Northeastern University

- An index is a data structure that can speed up queries/searches
  - e.g., searching for instructors in a specific department



- Drawbacks of indexes
  - Causes updates to the table to become slower (since the index also needs to be updated)
  - Storage space

Creating Indexes

Northeastern University

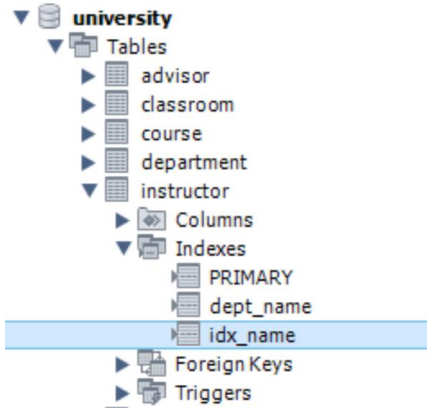
- We create an index using the CREATE INDEX command

```
CREATE [UNIQUE] INDEX index_name
ON table_name (column1, column2, ...);
```

  - A UNIQUE index creates a constraint such that all values in the index must be distinct
- For example, let's define an index on the name column in the instructor table:

```
CREATE INDEX idx_name
ON instructor (name);
```

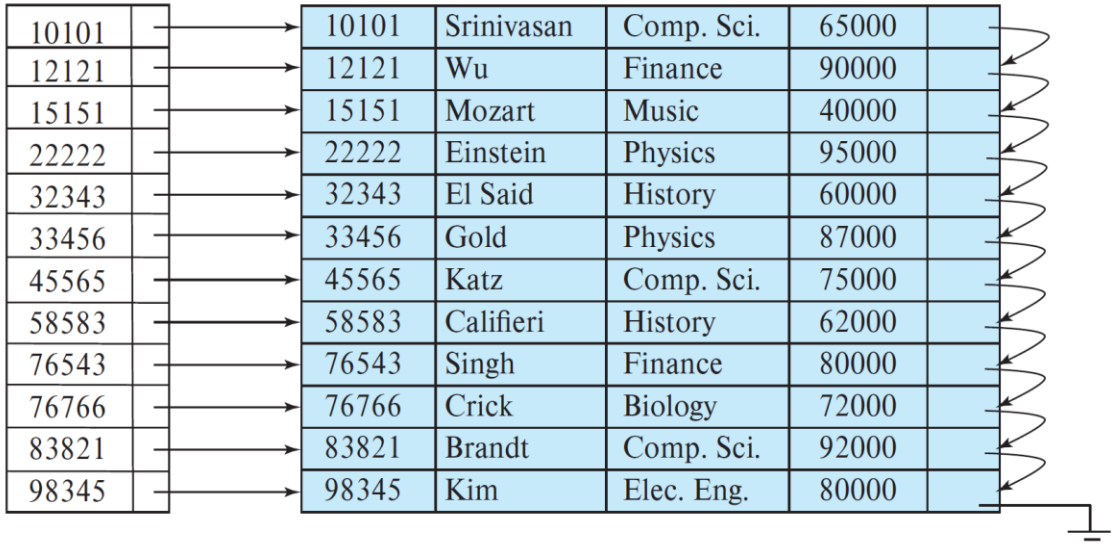
- You can see the new index under the Indexes folder of the table:



Clustered Indexes

Northeastern University

- A **clustered index** defines the physical order in which table records are stored
- There can be only one clustered index per table
  - By default a clustered index is created on a primary key column
- Accessing a row through the clustered index is fast
  - Since the index search leads directly to the page that contains the row data

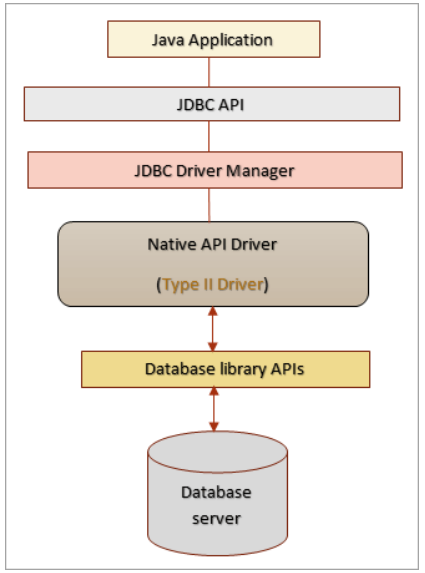


10101	10101	Srinivasan	Comp. Sci.	65000
12121	12121	Wu	Finance	90000
15151	15151	Mozart	Music	40000
22222	22222	Einstein	Physics	95000
32343	32343	El Said	History	60000
33456	33456	Gold	Physics	87000
45565	45565	Katz	Comp. Sci.	75000
58583	58583	Califieri	History	62000
76543	76543	Singh	Finance	80000
76766	76766	Crick	Biology	72000
83821	83821	Brandt	Comp. Sci.	92000
98345	98345	Kim	Elec. Eng.	80000

Accessing SQL from a Programming Languages

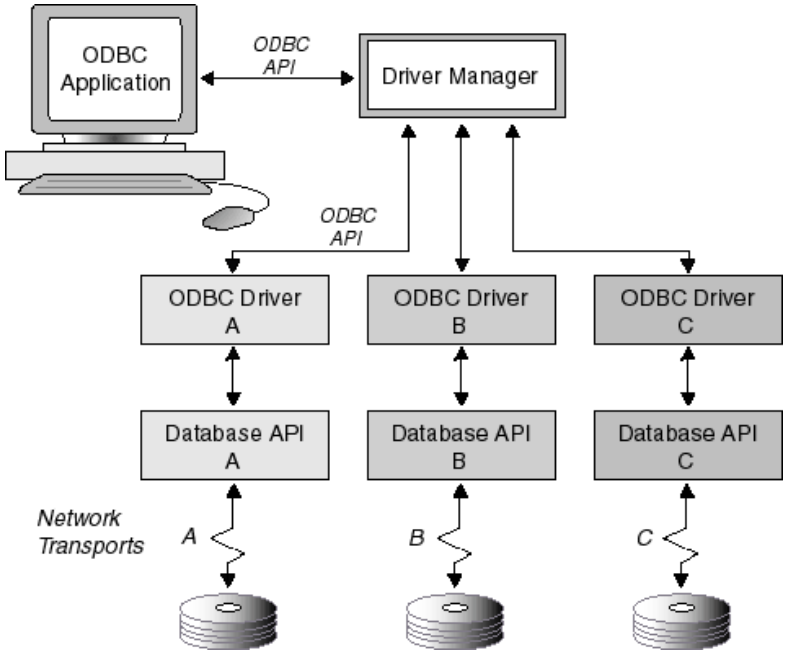
Northeastern University

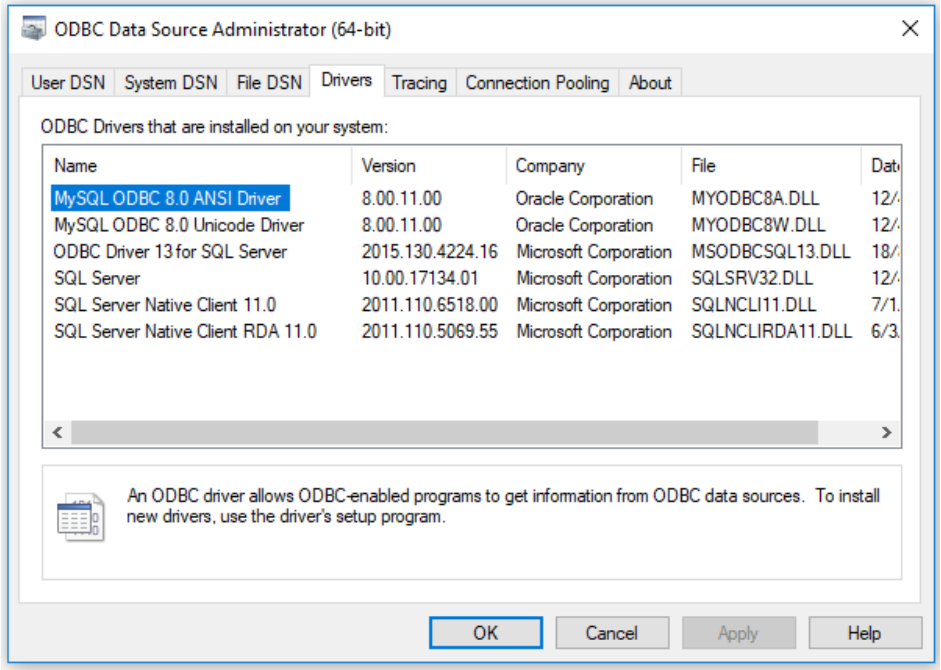
- To access SQL from a programming language you need a **database driver**
- A driver converts application commands into a format that the DBMS can understand
- Typical commands include
  - Connection commands (e.g., open or close a connection to the database)
  - SQL commands
  - Handling cursors, which allow you to traverse through result sets
  - Transaction management commands
- Types of drivers
  - Native drivers – provided by specific database vendors
  - ODBC drivers – provide a standard interface to communicate with databases
  - JDBC drivers – specific to Java-based applications



ODBC

- Open Database Connectivity (ODBC) is a standard API for accessing DBMS
- Independent of any specific DBMS or operating system
- ODBC drivers exist for most of the commercial databases





MySQL Connector/Python

Northeastern University

- **Python Database API** (DB-API) defines a standard interface for accessing relational databases from Python programs
- Different packages in Python implement this interface for different databases
  - e.g., mysql.connector for MySQL, sqlite3 for SQLite, pyodbc for ODBC
- mysql.connector is the recommended driver for interacting with MySQL
  - Developed by the MySQL group at Oracle
- Install it via pip

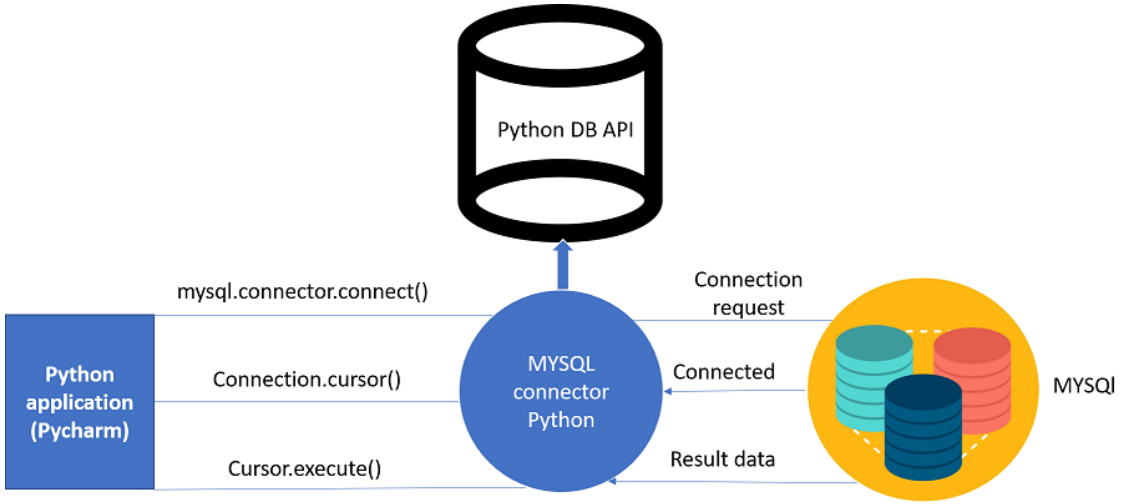
```
pip install mysql-connector-python
```

  - If you've installed Python with the Anaconda distribution, you should already have it

MySQL Connector/Python

Northeastern University

- The general workflow of a Python program that interacts with a MySQL database:
  - Connect to the MySQL server
  - Execute a SQL query
  - Fetch the results
  - Inform the database if any table changes were made (by committing the changes)
  - Close the connection to the server



Establishing a Connection with MySQL Server

Northeastern University

- To connect to a MySQL server, call the **connect()** function in mysql.connector module
  - This function gets 4 parameters: host, user, password and database name
  - It returns a MySQLConnection object

```
from mysql.connector import connect, Error
from getpass import getpass
```

```
try:
    conn = connect(
        host='localhost',
        user=input('Enter username: '),
        password=getpass('Enter password: '),
        database='university'
    )
except Error as e:
    print(e)
```

- You should always close the connection in the end by calling **conn.close()**
- You should never hard-code your login credentials directly in a Python script

The Cursor Object

Northeastern University

- In order to execute SQL queries in Python, you need a **cursor** object
  - A cursor object allows you to traverse over database records
- To create a cursor, use the **cursor()** method of your connection object
- Then you execute a SQL query by calling **cursor.execute**(query)
- If the query returns rows, you can retrieve them using one of cursor's fetch methods:
  - **fetchall**() - retrieves all the rows from the result as a list of tuples
  - **fetchone**() - retrieves the next row of the result as a tuple
    - Returns None if no more rows are available
  - **fetchmany**(*n*) - retrieves the next *n* rows from the result as a list of tuples (*n* defaults to 1)
    - Returns an empty list if no more rows are available

Reading Records from a Table

Northeastern University

- The following example selects all the records from the *instructor* table:

```
inst_query = "SELECT * FROM instructor"
with conn.cursor() as cursor:
    cursor.execute(inst_query)
    result = cursor.fetchall()
    for row in result:
        print(row)
```

```
('10101', 'Srinivasan', 'Comp. Sci.', Decimal('65000.00'))
('12121', 'Wu', 'Finance', Decimal('90000.00'))
('15151', 'Mozart', 'Music', Decimal('40000.00'))
('22222', 'Einstein', 'Physics', Decimal('95000.00'))
('32343', 'El Said', 'History', Decimal('60000.00'))
('33456', 'Gold', 'Physics', Decimal('87000.00'))
('45565', 'Katz', 'Comp. Sci.', Decimal('75000.00'))
('58583', 'Califieri', 'History', Decimal('62000.00'))
('76543', 'Singh', 'Finance', Decimal('80000.00'))
('76766', 'Crick', 'Biology', Decimal('72000.00'))
('83821', 'Brandt', 'Comp. Sci.', Decimal('92000.00'))
('98345', 'Kim', 'Elec. Eng.', Decimal('80000.00'))
```

- The result variable holds the records returned from.fetchall()
- It's a list of tuples representing individual records from the table

Reading Records from a Table

Northeastern University

- To get specific attributes in each row, specify their indexes in the returned tuples
- For example, to print only the instructor names:

```
inst_query = "SELECT * FROM instructor"
with conn.cursor() as cursor:
    cursor.execute(inst_query)
    result = cursor.fetchall()
    for row in result:
        print(row[1])
```

```
Srinivasan
Wu
Mozart
Einstein
El Said
Gold
Katz
Califieri
Singh
Crick
Brandt
Kim
```

Cursor as an Iterator

Northeastern University

- To process the rows in the result one at a time, you can use the cursor as an iterator:

```
inst_query = "SELECT * FROM instructor"
with conn.cursor() as cursor:
    cursor.execute(inst_query)
    for row in cursor:
        print(row)
```

```
('10101', 'Srinivasan', 'Comp. Sci.', Decimal('65000.00'))
('12121', 'Wu', 'Finance', Decimal('90000.00'))
('15151', 'Mozart', 'Music', Decimal('40000.00'))
('22222', 'Einstein', 'Physics', Decimal('95000.00'))
('32343', 'El Said', 'History', Decimal('60000.00'))
('33456', 'Gold', 'Physics', Decimal('87000.00'))
('45565', 'Katz', 'Comp. Sci.', Decimal('75000.00'))
('58583', 'Califieri', 'History', Decimal('62000.00'))
('76543', 'Singh', 'Finance', Decimal('80000.00'))
('76766', 'Crick', 'Biology', Decimal('72000.00'))
('83821', 'Brandt', 'Comp. Sci.', Decimal('92000.00'))
('98345', 'Kim', 'Elec. Eng.', Decimal('80000.00'))
```

More Complex Queries

Northeastern University

- You can make your **select** queries as complex as you want using the same methods
- Example: Find the courses that were taken by the highest number of students

```
most_popular_courses = """
    SELECT C.course_id, C.title
    FROM course as C, takes as T
    WHERE C.course_id = T.course_id
    GROUP BY T.course_id
    HAVING count(*) >= ALL (
        SELECT count(*)
        FROM takes
        GROUP BY course_id
    )
"""
with conn.cursor() as cursor:
    cursor.execute(most_popular_courses)
    result = cursor.fetchall()
    print(result)
```

```
[('CS-101', 'Intro. to Computer Science')]
```

Cursor Properties

Northeastern University

- The cursor has a few useful properties that provide information about the result set
  - **column_names** – returns the list containing the column names of the result set
  - **rowcount** – the number of rows in the result set

```
inst_query = "SELECT * FROM instructor"
with conn.cursor() as cursor:
    cursor.execute(inst_query)
    cursor.fetchall()
    print('Column names:', cursor.column_names)
    print('Row count:', cursor.rowcount)
```

```
Column names: ('ID', 'name', 'dept_name', 'salary')
Row count: 12
```

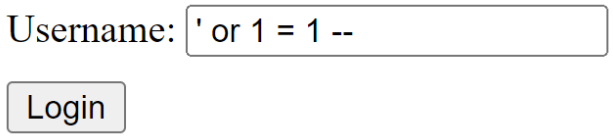
SQL Injection Attack

Northeastern University

- A common attack that enables execution of malicious SQL statements in the DB
  - By insertion ("injection") of a SQL query via the input data from the client
- Suppose you write a script that checks if a given username exists in the users table
- You construct the following SQL query:

```
username = # read from the input
query = "SELECT * FROM users WHERE username = '" + username + "'"
```

- If the user, instead of entering their username, enters:



- Then the query becomes:

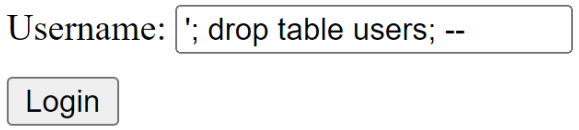
```
query = "SELECT * FROM users WHERE username = '' or 1 = 1 --'"
```

- The **where** clause is now always true and the entire users table is returned

SQL Injection Attack

Northeastern University

- Most databases allow execution of multiple SQL statements separated by semicolon
- This allows the hacker to inject whole SQL statements into the query
- For example, the hacker could enter the following string:



- The resulting query would be:

```
query = "SELECT * FROM users WHERE username = ''; drop table users; --"
```

- This query would result in deleting the entire users table!

- **Solution**: use parameterized queries whenever user input is involved in the query

SQL Injection Attack

Northeastern University

- Example:

```
inst_name = input('Enter instructor name: ')
inst_query = f"""
    SELECT * FROM instructor
    WHERE name = '{inst_name}'
"""
print(inst_query)
with conn.cursor() as cursor:
    cursor.execute(inst_query)
    for row in cursor:
        print(row)
```

```
Enter instructor name: X' or 'Y' = 'Y

    SELECT * FROM instructor
    WHERE name = 'X' or 'Y' = 'Y'

('10101', 'Srinivasan', 'Comp. Sci.', Decimal('65000.00'))
('12121', 'Wu', 'Finance', Decimal('90000.00'))
('15151', 'Mozart', 'Music', Decimal('40000.00'))
('22222', 'Einstein', 'Physics', Decimal('95000.00'))
('32343', 'El Said', 'History', Decimal('60000.00'))
('33456', 'Gold', 'Physics', Decimal('87000.00'))
('45565', 'Katz', 'Comp. Sci.', Decimal('75000.00'))
```

Parameterized Query

Northeastern University

- A **parameterized query** is a query that uses placeholders (%) for attribute values
- Strings passed to the placeholders are correctly escaped by the library at runtime
  - e.g., each quotation mark is doubled
- You can pass a parameterized query to **cursor.execute()** as follows:

```
inst_name = input('Enter instructor name: ')
inst_query = """
    SELECT * FROM instructor
    WHERE name = %s
"""
val_tuple = (inst_name,)

with conn.cursor() as cursor:
    cursor.execute(inst_query, val_tuple)
    for row in cursor:
        print(row)
```

```
Enter instructor name: Katz
('45565', 'Katz', 'Comp. Sci.', Decimal('75000.00'))
```

Parameterized Query

Northeastern University

- If a user tries to sneak in some problematic characters, the resulting statement will cause no harm since each quotation mark will be doubled:

select * **from** instructor **where** name = 'X'' or ''Y'' = ''Y'

- This query will return an empty set

```
inst_name = input('Enter instructor name: ')
inst_query = """
    SELECT * FROM instructor
    WHERE name = %s
"""
val_tuple = (inst_name,)

with conn.cursor() as cursor:
    cursor.execute(inst_query, val_tuple)
    for row in cursor:
        print(row)
```

```
Enter instructor name: X' or 'Y' = 'Y
```

Inserting New Records

Northeastern University

- To insert data, pass the **insert into** command to the cursor's **execute()** method
- For example, to add a new instructor:

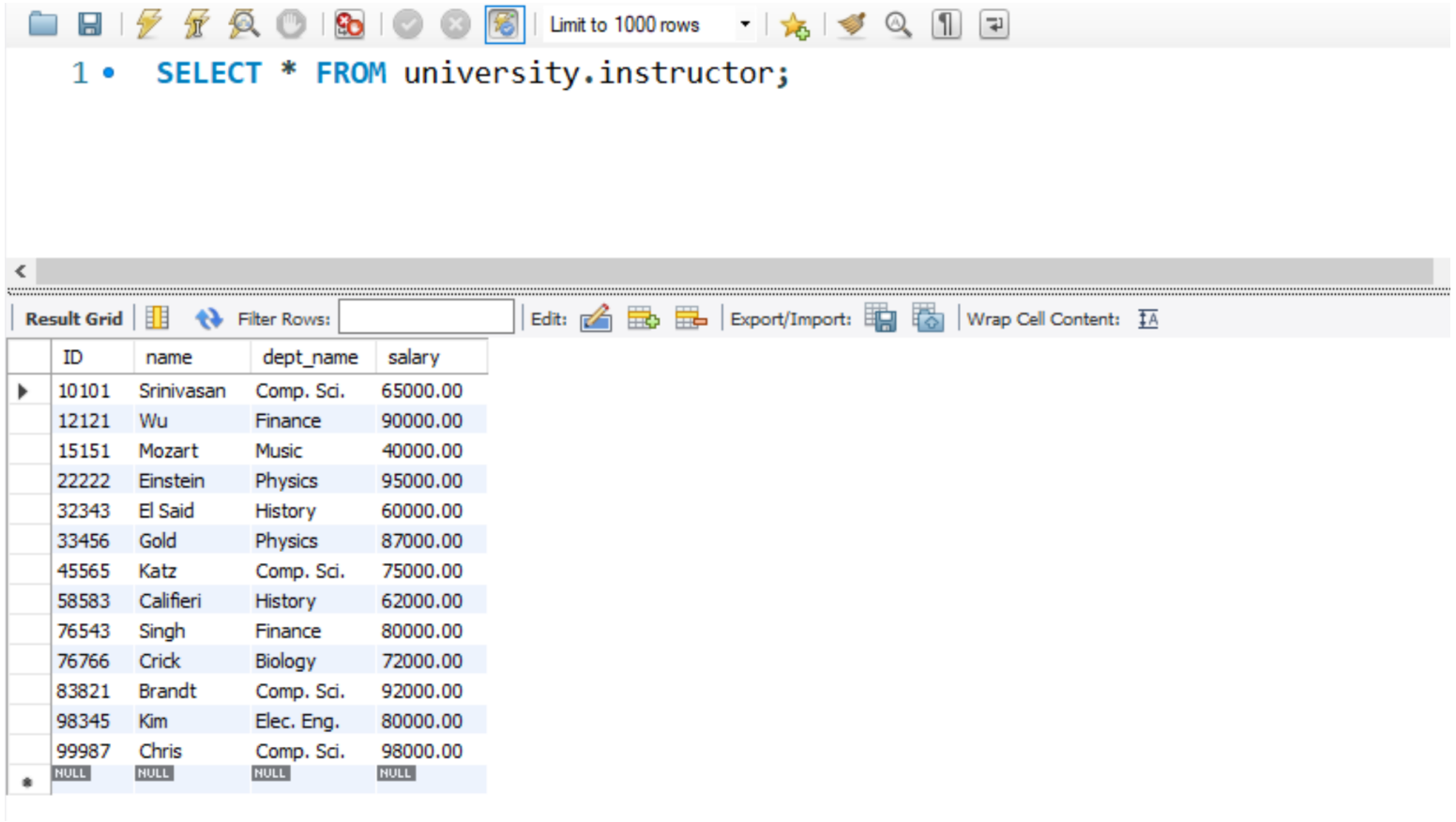
```
add_inst_query = """
    INSERT INTO instructor
    VALUES ('99987', 'Chris', 'Comp. Sci.', 98000)
"""

with conn.cursor() as cursor:
    cursor.execute(add_inst_query)
    conn.commit()
```

- You must call **conn.commit()** at the end, otherwise your changes will be lost!
  - Unless you turn on automatic commits by setting conn.autocommit = True

Inserting New Records

Northeastern University

- Verifying that the new record was added to the table:



ID	name	dept_name	salary
10101	Srinivasan	Comp. Sci.	65000.00
12121	Wu	Finance	90000.00
15151	Mozart	Music	40000.00
22222	Einstein	Physics	95000.00
32343	El Said	History	60000.00
33456	Gold	Physics	87000.00
45565	Katz	Comp. Sci.	75000.00
58583	Califieri	History	62000.00
76543	Singh	Finance	80000.00
76766	Crick	Biology	72000.00
83821	Brandt	Comp. Sci.	92000.00
98345	Kim	Elec. Eng.	80000.00
99987	Chris	Comp. Sci.	98000.00
NULL	NULL	NULL	NULL

Inserting New Records

Northeastern University

- If the values to be inserted come from an external source (e.g., the user), again you should use parameters in the SQL statement:

```
inst_id = input('Enter instructor id: ')
inst_name = input('Enter instructor name: ')
dept_name = input('Enter department name: ')
salary = input('Enter salary: ')

add_inst_query = """
    INSERT INTO instructor
    VALUES (%s, %s, %s, %s)
"""
val_tuple = (inst_id, inst_name, dept_name, salary)

with conn.cursor() as cursor:
    cursor.execute(add_inst_query, val_tuple)
    conn.commit()
```

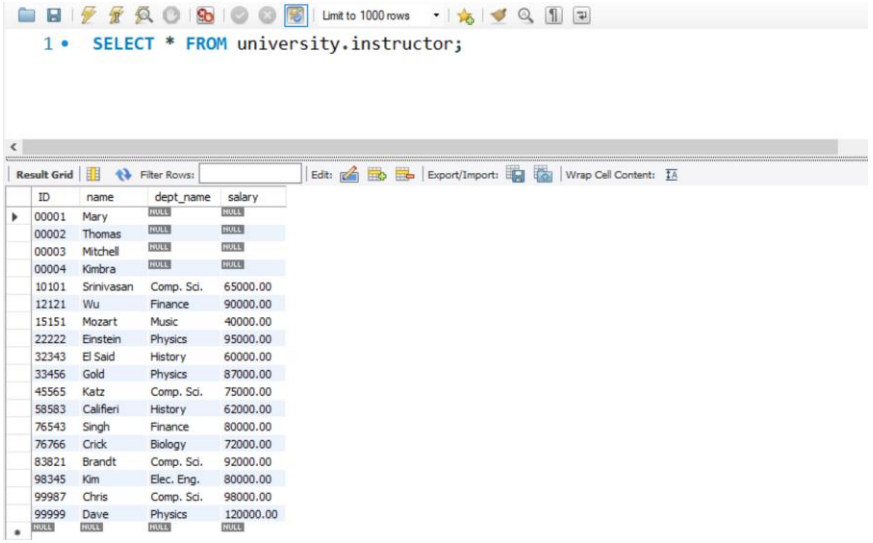
```
Enter instructor id: 99999
Enter instructor name: Dave
Enter department name: Physics
Enter salary: 120000
```

Inserting a Bulk of Records

Northeastern University

- You can insert multiple records using the **executemany()** method
- It accepts two parameters:
  - A **query** that contains placeholders for the records that need to be inserted
  - A **list** that contains all records that you wish to insert

```
add_instructors_query = """
    INSERT INTO instructor
    (ID, name)
    VALUES (%s, %s)
"""

instructors = [
    ('00001', 'Mary'),
    ('00002', 'Thomas'),
    ('00003', 'Mitchell'),
    ('00004', 'Kimbra')
]

with conn.cursor() as cursor:
    cursor.executemany(add_instructors_query, instructors)
    conn.commit()
```



Update and Delete

Northeastern University

- Updating and deleting work the same way, just pass the SQL to **cursor.execute()**
- You can use the cursor **rowcount** attribute to check how many records were affected:

```
delete_inst_query = """
    DELETE FROM instructor
    WHERE ID = '00001'
"""

with conn.cursor() as cursor:
    cursor.execute(delete_inst_query)
    conn.commit()
    print('Number of rows deleted:', cursor.rowcount)
```

```
Number of rows deleted: 1
```

Calling Stored Procedures from Python

Northeastern University

- To call a stored procedure use the **callproc()** method of the Cursor object

  cursor.callproc(procedure_name, args=())

- Then, you can call the **stored_results()** method to get an iterator with the result set
  - The rows in the result can be read by calling the **fetchall()** method
- Example:

```
with conn.cursor() as cursor:
    cursor.callproc('get_courses_by_department', ('Comp. Sci.',))
    results = cursor.stored_results()
    for result in results:
        for course in result.fetchall():
            print(course)
```

```
('CS-101', 'Intro. to Computer Science', 'Comp. Sci.', Decimal('4'))
('CS-190', 'Game Design', 'Comp. Sci.', Decimal('4'))
('CS-315', 'Robotics', 'Comp. Sci.', Decimal('3'))
('CS-319', 'Image Processing', 'Comp. Sci.', Decimal('3'))
('CS-347', 'Database System Concepts', 'Comp. Sci.', Decimal('3'))
```

Loading Data into a DataFrame

Northeastern University

- Pandas provides a **read_sql()** method that reads an SQL query into a DataFrame

pandas.read_sql(*sql, con, index_col=None, coerce_float=True, params=None, parse_dates=None, columns=None, chunksize=None*) [source]

- For example, loading the instructor table into a DataFrame:

```
import pandas as pd

query = "SELECT * FROM instructor"
df = pd.read_sql(query, conn)
```

```
df.head()
```

	ID	name	dept_name	salary
0	10101	Srinivasan	Comp. Sci.	65000.0
1	12121	Wu	Finance	90000.0
2	15151	Mozart	Music	40000.0
3	22222	Einstein	Physics	95000.0
4	32343	El Said	History	60000.0

Loading Data into a DataFrame

Northeastern University

- You can use the **params** argument to pass a list of parameters to the query
- For example, the following displays all the students that took the course CS-319

```
query = """
    SELECT * FROM takes
    WHERE course_id = %s
"""

df = pd.read_sql(query, conn, params=['CS-319'])
```

```
df
```

	ID	course_id	sec_id	semester	year	grade
0	45678	CS-319	1	Spring	2018.0	B
1	76543	CS-319	2	Spring	2018.0	A