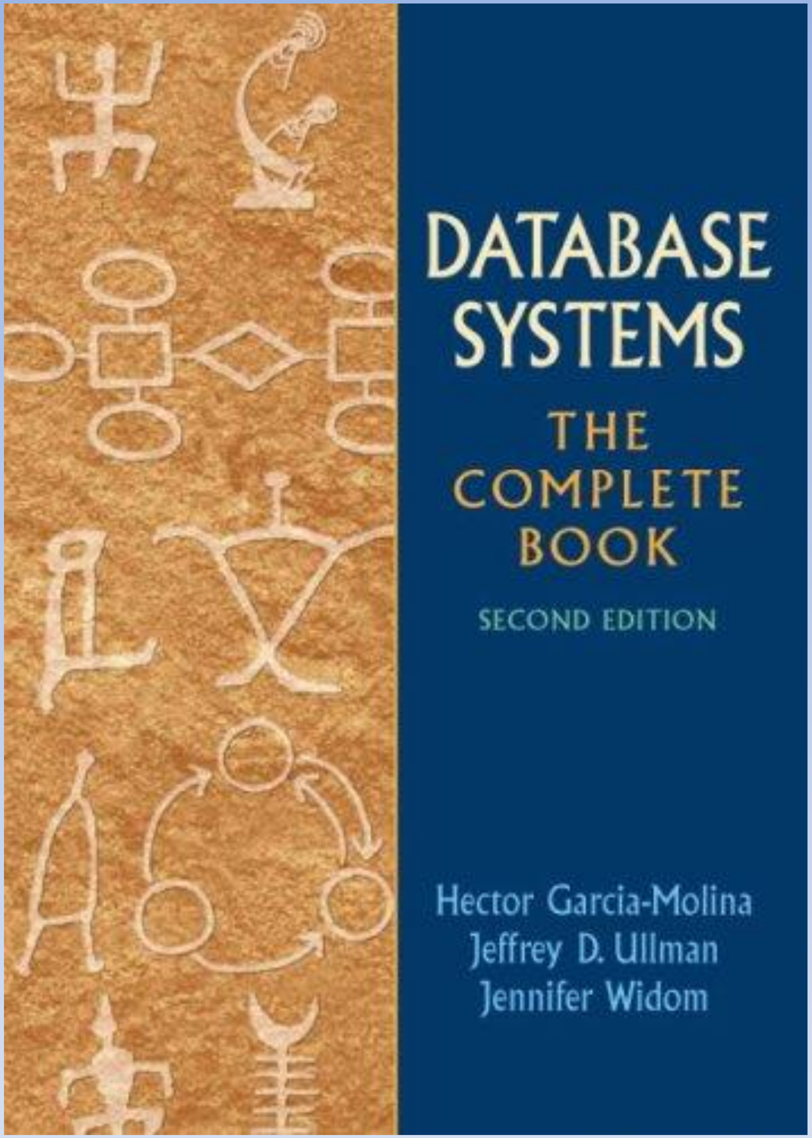


Chapter #9

SQL in a Server Environment

Three-Tier Architecture

- Web Servers
- Application Servers
- Database Servers

Web-Server Tier

- User makes contact via URL
- Web Server (Apache, IIS, etc…) responds
- Web Server works with Application Tier for client request fulfillment.

Application Tier

- Does the processing work for client fulfillment.
- Application Tier may be multi-layered.

Database Tier

- Database Tier executes application queries.
- May involve multiple processes.
- May involve multiple ‘connections’ from clients.

Connections

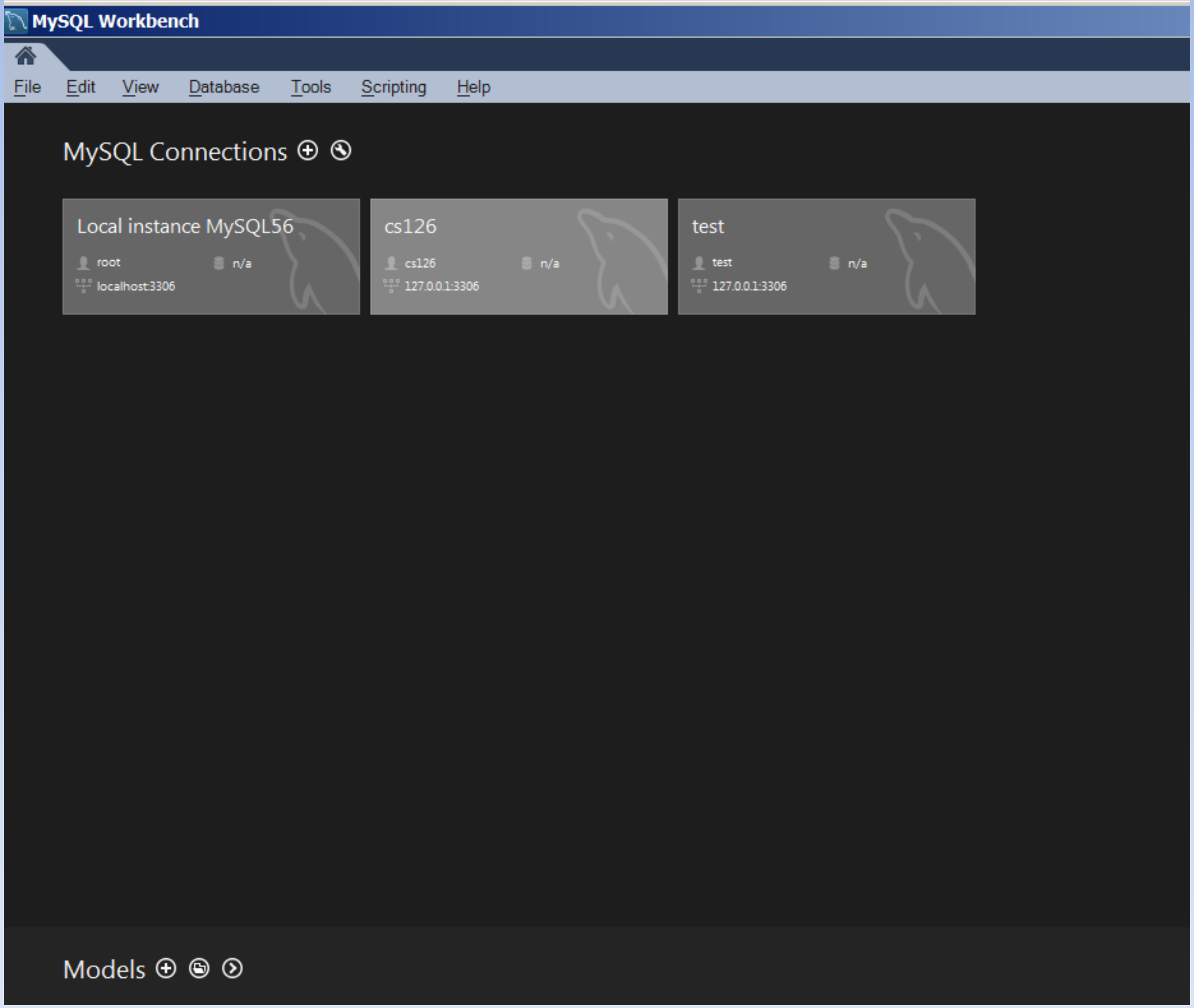
- Running SQL commands from a program requires a Connection between Client & Server.

- SQL Connection Command Syntax:

```
CONNECT TO <server name>
AS  <connection name>
AUTHORIZATION <name and password>
```

- Server name can be DEFAULT.

Connections in MySQL

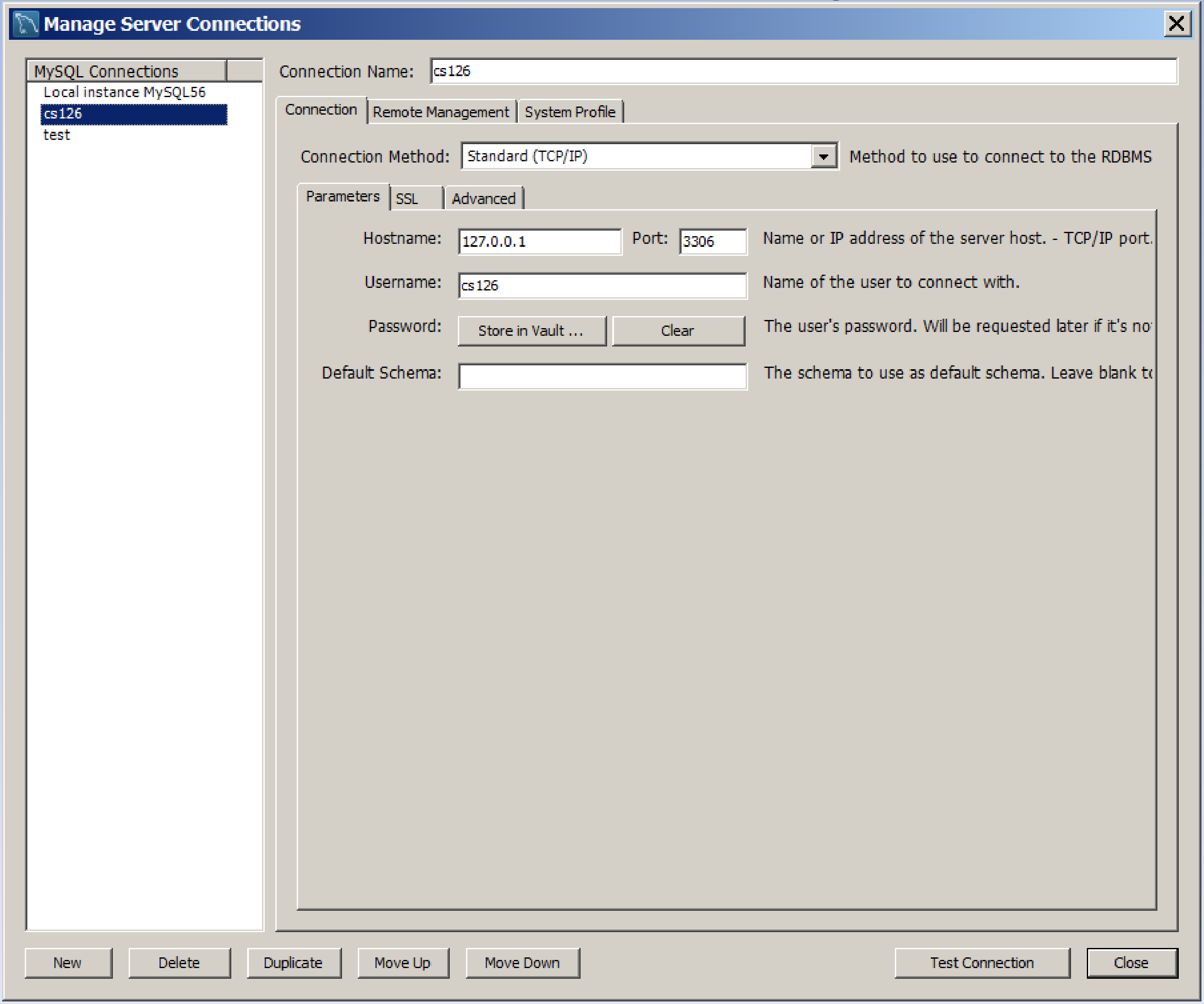


Local instance MySQL56
root n/a
localhost:3306

cs126
cs126 n/a
127.0.0.1:3306

test
test n/a
127.0.0.1:3306

Connections in MySQL



Manage Server Connections

Connection Name: cs126

Connection Method: Standard (TCP/IP) Method to use to connect to the RDBMS

Hostname: 127.0.0.1 Port: 3306 Name or IP address of the server host. - TCP/IP port.

Username: cs126 Name of the user to connect with.

Password: Store in Vault ... Clear The user's password. Will be requested later if it's no

Default Schema: The schema to use as default schema. Leave blank to

Connection w/ Python

- Similar Connection through Python
- Early Example Code: Mysqlt.py

Mysqlt.py

```
def test1(password):
  conn = mysql.connector.connect (user="root", passwd=password)
  cursor = conn.cursor ()
  cursor.execute ("SELECT VERSION()")
  row = cursor.fetchone ()
  print "server version:", row[0]
  cursor.close ()
  conn.close ()
```

Sessions

- SQL operations performed while active connection.

Stored Procedures

- PSM, or "*persistent stored modules*," allows us to store procedures as database schema elements.
- PSM = a mixture of conventional statements (if, while, etc.) and SQL.
- Lets us do things we cannot do in SQL alone.

Basic PSM Form

```
CREATE PROCEDURE <name> (
        <parameter list> )
   <optional local declarations>
   <body>;
```

- Function alternative:

```
CREATE FUNCTION <name> (
        <parameter list> ) RETURNS <type>
```

Basic PSM Form

- Function alternative:

```
CREATE FUNCTION <name> (
        <parameter list> )
<optional local declarations>
   <body>;
RETURNS <type>
```

Parameters in PSM

- Unlike the usual name-type pairs in languages like C, PSM uses mode-name-type triples, where the *mode* can be:
  - IN = procedure uses value, does not change value.
  - OUT = procedure changes, does not use.
  - INOUT = both.

Example: Stored Procedure

- Let's write a procedure that takes two arguments *a* and *b*, and adds a tuple to BankAccount that has PersonID = 123, AccountID = *a*, and amount = *b*.

The Procedure

```
CREATE PROCEDURE Add123(in a int, in b int)

insert into BankAccount values(123, a, b);
```

The Procedure

```
CREATE PROCEDURE Add123 (
  IN  a       int,
  IN b        int
)
INSERT INTO BankAccount
VALUES(123, a, b);
```

Parameters are both
read-only, not changed

The body ---
a single insertion

Invoking Procedures

- Use SQL/PSM statement CALL, with the name of the desired procedure and arguments.
- Example:

```
CALL Add123(802, 1000);
```

- Functions used in SQL expressions wherever a value of their return type is appropriate.

Kinds of PSM statements – (1)

- RETURN <expression> sets the return value of a function.
  - Unlike C, etc., RETURN *does not* terminate function execution.
- DECLARE <name> <type> used to declare local variables.
- BEGIN . . . END for groups of statements.
  - Separate statements by semicolons.

Kinds of PSM Statements – (2)

- Assignment statements: SET <variable> = <expression>;
  - Example: `SET b = 1000;`
- Statement labels: give a statement a label by prefixing a name and a colon.

The Next Procedure

```
CREATE PROCEDURE AddTo123(
  in a int, in b int)
begin
  declare c int;
  SELECT amount INTO c FROM BankAccount
    WHERE PersonID=123 AND AccountID=a;
  UPDATE BankAccount set amount=c+b
    WHERE PersonID=123 AND AccountID=a;
end
```

IF Statements

- Simplest form:

```
IF <condition> THEN
    <statements(s)>
END IF;
```

- Add ELSE <statement(s)> if desired, as

```
    IF . . . THEN . . . ELSE . . . END IF;
```

- Add additional cases by ELSEIF <statements(s)>: IF … THEN … ELSEIF … THEN … ELSEIF … THEN … ELSE … END IF;

Simple Conditional Test

- Checking results of update query.
- Does actual change match expected change.

Example: IF

```
CREATE PROCEDURE test1(
in count int, in expected int)
begin
  if count = expected then
     select 'success';
     commit;
  else
     select 'Error';
     rollback;
  end if;
end
```

Loops

- Basic form:

```
<loop name>: LOOP <statements>
              END LOOP;
```

- Exit from a loop by:

```
    LEAVE <loop name>
```

Example: Exiting a Loop

```
loop1: LOOP
        . . .
        LEAVE loop1;
        . . .
END LOOP;
```

If this statement is executed . . .

Control winds up here

Other Loop Forms

```
• WHILE <condition>
      DO <statements>
  END WHILE;
• REPEAT
    <statements>
  UNTIL <condition>
END REPEAT;
```

Queries & Cursors

- General SELECT-FROM-WHERE queries are *not* permitted in PSM.
- There are three ways to get the effect of a query:

1. Queries producing one value can be the expression in an assignment.
2. Single-row SELECT . . . INTO.
3. Cursors.

Example: Assignment/Query

- Using local variable *p* and BankAccount(PersonID, AccountID, Amount), we can get the account balance for Person 123 with account 801:

```
SET p =
(SELECT amount FROM BankAccount
   WHERE PersonID = 123
    AND AccountID=801);
```

SELECT . . . INTO

- Another way to get the value of a query that returns one tuple is by placing INTO <variable> after the SELECT clause.
- Example:

```
SELECT amount INTO p
FROM BankAccount
WHERE PersonID = 123
AND AccountID=801;
```

Cursors

- A *cursor* is essentially a tuple-variable that ranges over all tuples in the result of some query.
- Declare a cursor *c* by:

DECLARE c CURSOR FOR <query>;

Opening and Closing Cursors

- To use cursor *c*, we must issue the command:

```
OPEN c;
```

  - The query of *c* is evaluated, and *c* is set to point to the first tuple of the result.

- When finished with *c*, issue command:

```
CLOSE c;
```

Fetching Tuples From a Cursor

- To get the next tuple from cursor c, issue command:

```
FETCH FROM c INTO x1, x2,…,xn ;
```

- The *x* ’s are a list of variables, one for each component of the tuples referred to by *c*.
- c is moved automatically to the next tuple.

Breaking Cursor Loops – (1)

- The usual way to use a cursor is to create a loop with a FETCH statement, and do something with each tuple fetched.
- A tricky point is how we get out of the loop when the cursor has no more tuples to deliver.

Breaking Cursor Loops – (2)

- Each SQL operation returns a *status*, which is a 5-digit character string.
  - For example, 00000 = "Everything OK," and 02000 = "Failed to find a tuple."
- In PSM, we can get the value of the status in a variable called SQLSTATE.

Breaking Cursor Loops – (3)

- We may declare a *condition*, which is a boolean variable that is true if and only if SQLSTATE has a particular value.
- Example: We can declare condition Not_Found to represent 02000 by:

```
DECLARE Not_Found CONDITION FOR
  SQLSTATE '02000';
```

Breaking Cursor Loops MySQL w/ handler

- We may declare a *handler*, which is a statement executed when condition occurs.
- Example: We can declare handler for `Not_Found` to set boolean ‘done’ true:

```
DECLARE CONTINUE HANDLER FOR
   Not_Found SET done = TRUE;
```

Breaking Cursor Loops

- The structure of a cursor loop is thus:

```
cursorLoop: LOOP
  …
  FETCH c INTO … ;
  IF Not_Found THEN LEAVE cursorLoop;
  END IF;
  …
END LOOP;
```

The Needed Declarations

```
CREATE PROCEDURE curdemo()
BEGIN
  DECLARE done INT DEFAULT FALSE;
  DECLARE a1 CHAR(3);
  DECLARE b  INT;
  DECLARE cur1 CURSOR FOR
  SELECT PersonID FROM Person;
  DECLARE CONTINUE HANDLER FOR
    NOT_FOUND SET done = TRUE;
```

The Procedure Body

```
OPEN cur1;

read_loop: LOOP
  FETCH cur1 INTO a1;
  IF done THEN
    LEAVE read_loop;
  END IF;
  select sum(amount) into b
    from BankAccount where PersonID = a1 ;
  INSERT INTO BankAccount
    VALUES(a1, null, b);
END LOOP;

CLOSE cur1;
```

Exercise 9.4.2

- Write the following PSM functions or procedures, based on the database schema:

  Product(maker, model, ctype)

  PC(model, speed, ram, hd, price)

  Laptop(model, speed, ram, hd, screen, price)

  Printer(model, color, ptype, price)

- Take a price as argument and return the model number of the PC whose price is closest.

Exercise 9.4.2

```
delimiter //
CREATE FUNCTION closestMatchPC(targetPrice INT)
  RETURNS int
BEGIN
    DECLARE closestmodel int;
    DECLARE diffSq INT;
    DECLARE currSq INT;
    DECLARE m int;
    DECLARE p INT;
    DECLARE Not_Found INT DEFAULT FALSE;
    DECLARE PCCursor CURSOR FOR   SELECT model, price FROM PC;
    DECLARE CONTINUE HANDLER FOR NOT FOUND
        SET Not_Found = TRUE;
```

Exercise 9.4.2

```
  SET closestmodel = 0;
  SET diffSq = -1;
  OPEN PCCursor;
  mainLoop: LOOP
    FETCH PCCursor INTO m, p;
    IF Not_Found THEN
      LEAVE mainLoop;
     END IF;
    SET currSq = (p - targetPrice)*(p - targetPrice);
    IF diffSq = -1 OR diffSq > currSq THEN
        SET closestmodel = m;
        SET diffSq = currSq;
    END IF;
  END LOOP;
  CLOSE PCCursor;
  RETURN(closestmodel);
END//
delimiter ;
```

Exercise 9.4.2
Initialize

```
  SET closestmodel = 0;
  SET diffSq = -1;
  OPEN PCCursor;
  mainLoop: LOOP
    FETCH PCCursor INTO m, p;
    IF Not_Found THEN
      LEAVE mainLoop;
     END IF;
    SET currSq = (p - targetPrice)*(p - targetPrice);
    IF diffSq = -1 OR diffSq > currSq THEN
         SET closestmodel = m;
         SET diffSq = currSq;
    END IF;
  END LOOP;
  CLOSE PCCursor;
  RETURN(closestmodel);
END//
delimiter ;
```

Exercise 9.4.2
Main Loop

```
    SET closestmodel = 0;
    SET diffSq = -1;
    OPEN PCCursor;
    mainLoop: LOOP
        FETCH PCCursor INTO m, p;
        IF Not_Found THEN
            LEAVE mainLoop;
         END IF;
        SET currSq = (p - targetPrice)*(p - targetPrice);
        IF diffSq = -1 OR diffSq > currSq THEN
                 SET closestmodel = m;
                 SET diffSq = currSq;
        END IF;
    END LOOP;
    CLOSE PCCursor;
    RETURN(closestmodel);
END//
delimiter ;
```

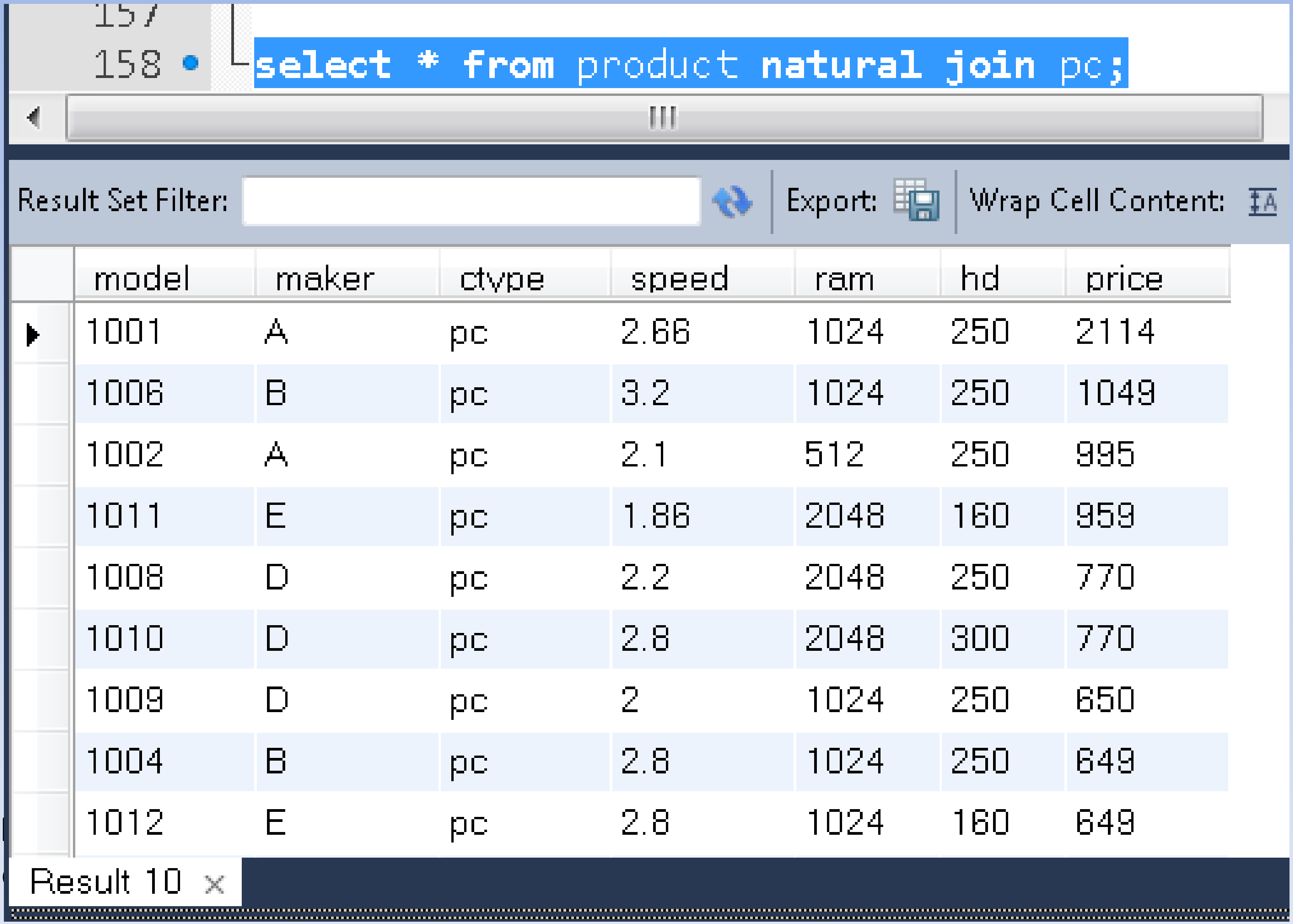
Exercise 9.4.2
Exit Loop

```
   SET closestmodel = 0;
   SET diffSq = -1;
   OPEN PCCursor;
   mainLoop: LOOP
      FETCH PCCursor INTO m, p;
      IF Not_Found THEN
         LEAVE mainLoop;
       END IF;
      SET currSq = (p - targetPrice)*(p - targetPrice);
      IF diffSq = -1 OR diffSq > currSq THEN
            SET closestmodel = m;
            SET diffSq = currSq;
      END IF;
   END LOOP;
   CLOSE PCCursor;
   RETURN(closestmodel);
END//
delimiter ;
```

Exercise 9.4.2
Store Closest Price w/ currSq

```
    SET closestmodel = 0;
    SET diffSq = -1;
    OPEN PCCursor;
    mainLoop: LOOP
        FETCH PCCursor INTO m, p;
        IF Not_Found THEN
            LEAVE mainLoop;
         END IF;
        SET currSq = (p - targetPrice)*(p - targetPrice);
        IF diffSq = -1 OR diffSq > currSq THEN
                SET closestmodel = m;
                SET diffSq = currSq;
        END IF;
    END LOOP;
    CLOSE PCCursor;
    RETURN(closestmodel);
END//
delimiter ;
```

Exercise 9.4.2



```
select * from product natural join pc;
```

model	maker	ctype	speed	ram	hd	price
1001	A	pc	2.66	1024	250	2114
1006	B	pc	3.2	1024	250	1049
1002	A	pc	2.1	512	250	995
1011	E	pc	1.86	2048	160	959
1008	D	pc	2.2	2048	250	770
1010	D	pc	2.8	2048	300	770
1009	D	pc	2	1024	250	650
1004	B	pc	2.8	1024	250	649
1012	E	pc	2.8	1024	160	649

E

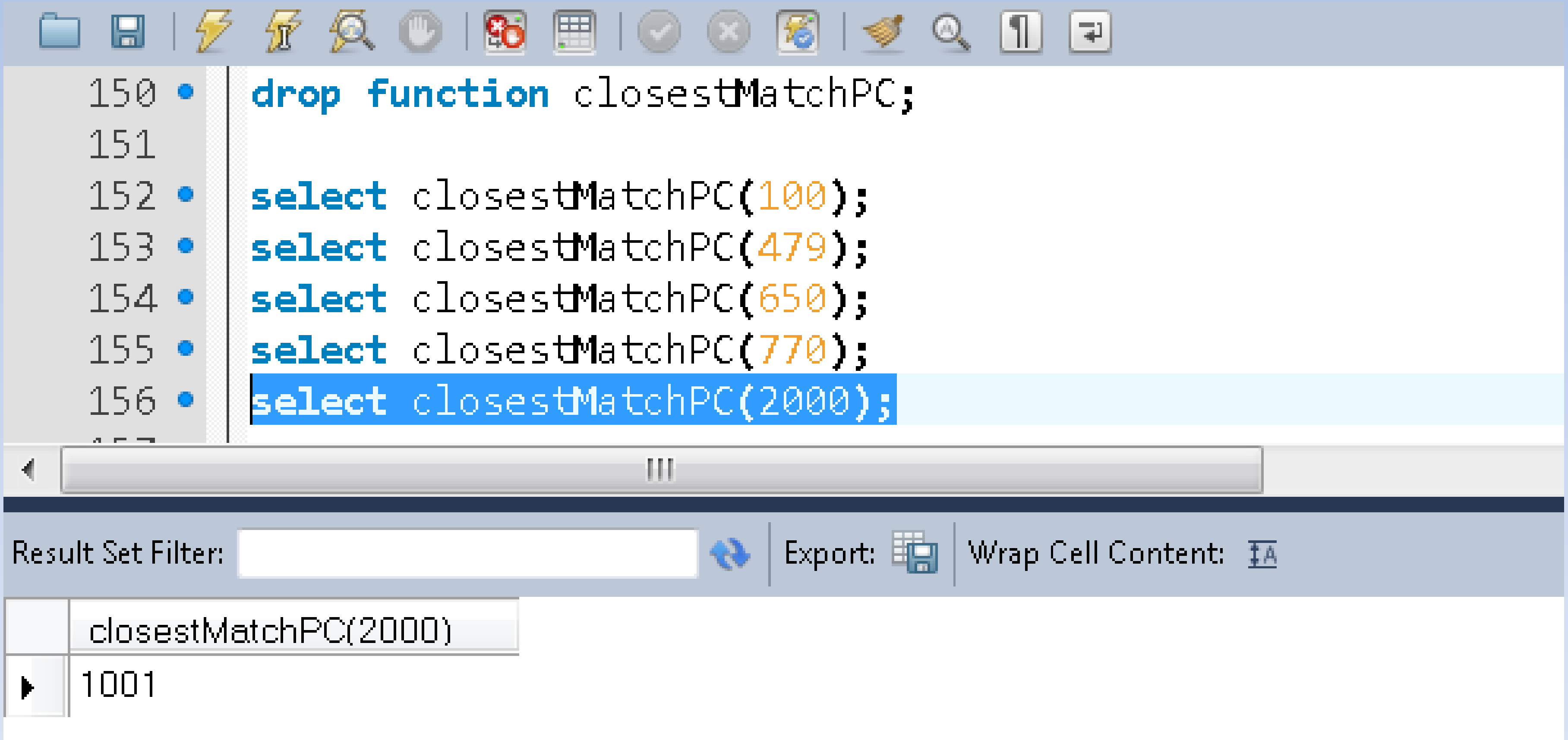
d

Exercise 9.4.2
Calling Function

```
select closestMatchPC(2000);
```

Exercise 9.4.2
Calling Function

select closestMatchPC(2000);



```
drop function closestMatchPC;

select closestMatchPC(100);
select closestMatchPC(479);
select closestMatchPC(650);
select closestMatchPC(770);
select closestMatchPC(2000);
```

```
closestMatchPC(2000)
1001
```

Exercise 9.4.2c

- Write the following PSM functions or procedures, based on the database schema:

  Product(maker, model, ctype)

  PC(model, speed, ram, hd, price)

  Laptop(model, speed, ram, hd, screen, price)

  Printer(model, color, ptype, price)

- Take model, speed, ram hard-disk and price information as arguments; and insert this information into the relation PC.
- However, if there is already a PC with that model number (tell by assuming that violation of a key constraint on insertion will raise an exception with SQLSTATE equal ‘23000’), then keep adding 1 to the model number until you find a model number that is not already a PC model number.

Exercise 9.4.2c

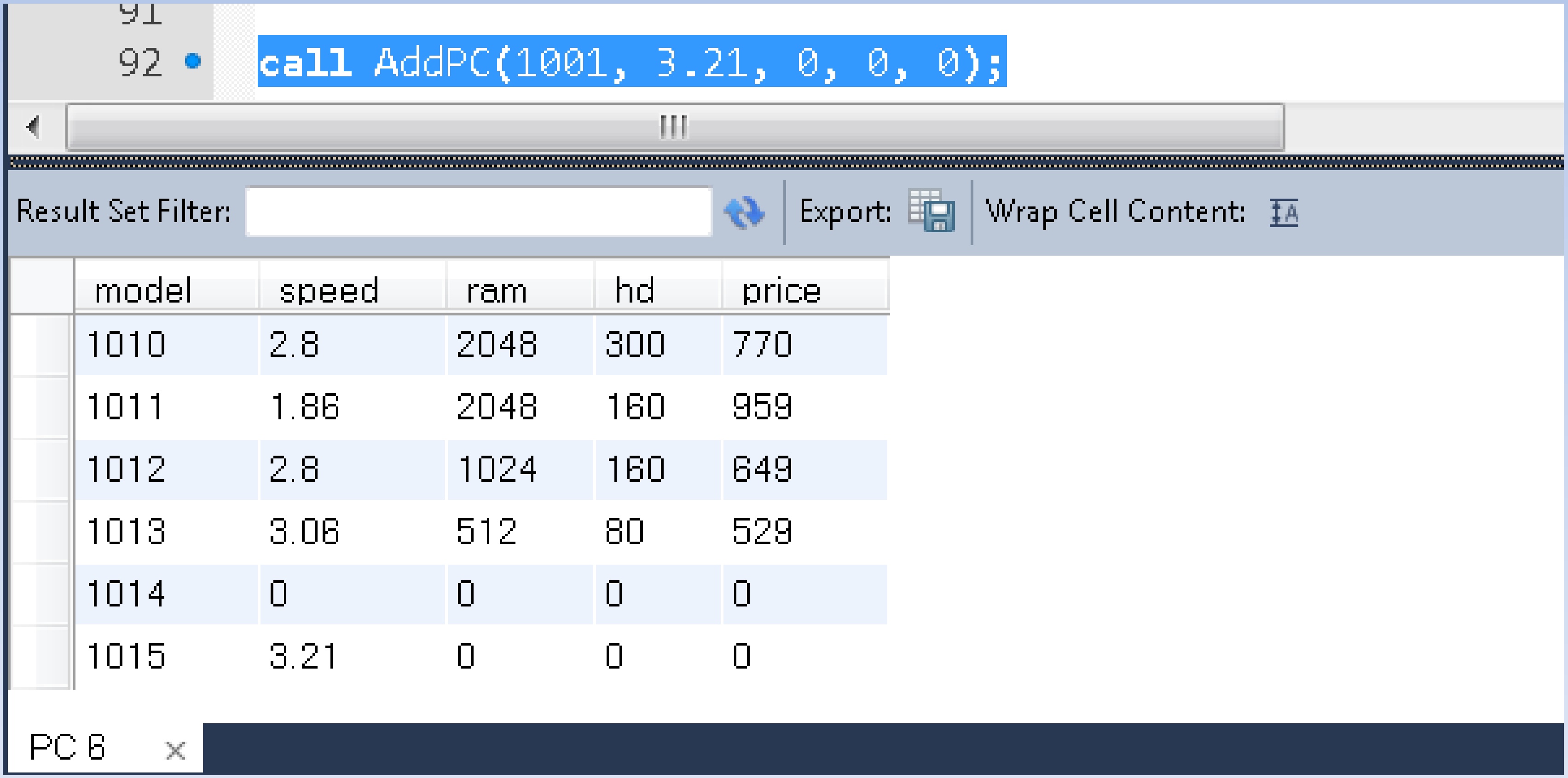
```
delimiter //
CREATE PROCEDURE addPC(IN imodel INT,  IN ispeed DECIMAL(3,2),
   IN iram   INT,  IN ihd  INT,  IN iprice INT)
BEGIN
  DECLARE Already_Exist INT DEFAULT FALSE;
  DECLARE CONTINUE HANDLER FOR SQLSTATE '23000'
     SET Already_Exist = TRUE;
  INSERT INTO PC VALUES(imodel, ispeed, iram, ihd, iprice);
  WHILE (Already_Exist) DO
    SET imodel = imodel + 1;
    set Already_Exist = FALSE;
    INSERT INTO PC VALUES(imodel, ispeed, iram, ihd, iprice);
  END WHILE;
END//
delimiter ;
```

Exercise 9.4.2c

```
call AddPC(1001, 3.21, 0, 0, 0);
```

Exercise 9.4.2c

call AddPC(1001, 3.21, 0, 0, 0);



```
call AddPC(1001, 3.21, 0, 0, 0);
```

model	speed	ram	hd	price
1010	2.8	2048	300	770
1011	1.86	2048	160	959
1012	2.8	1024	160	649
1013	3.06	512	80	529
1014	0	0	0	0
1015	3.21	0	0	0

Cursors in Python

- Cursors are what we’ve been using in our Python code examples.

Mysqlt.py

```
def test1(password):
    conn = mysql.connector.connect (user="root", passwd=password)
    cursor = conn.cursor ()
    cursor.execute ("SELECT VERSION()")
    row = cursor.fetchone ()
    print "server version:", row[0]
    cursor.close ()
    conn.close ()
```

MySQL w/ Python

- Establish a connection to Database Server

```
conn = mysql.connector.connect (
          user="root", passwd=“test”)
```

- Create a Cursor

```
cursor = conn.cursor ()
```

- Execute Query

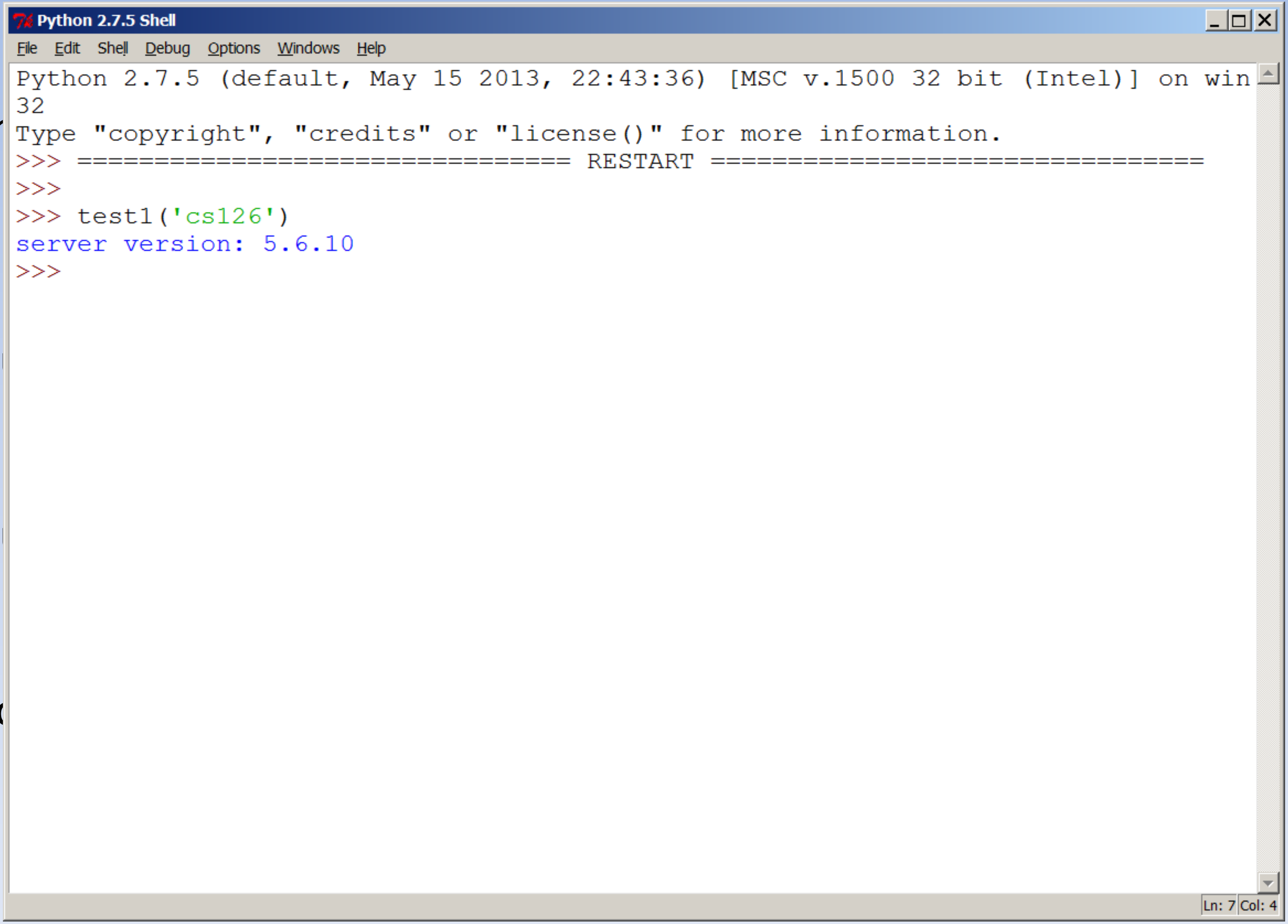
```
cursor.execute ("SELECT VERSION()")
```

- Fetch one result tuple

```
row = cursor.fetchone ()
```

MySQL w/ Python

- cor
- c
- c
- r



```
Python 2.7.5 (default, May 15 2013, 22:43:36) [MSC v.1500 32 bit (Intel)] on win
32
Type "copyright", "credits" or "license()" for more information.
>>> ================================ RESTART ================================
>>> 
>>> test1('cs126')
server version: 5.6.10
>>> 
```

Exercise 6.6.2a

```
def lookUpPC(speed, ram):
   conn = mysql.connector.connect(user="anonymous",  passwd="test",
                                              database="computers")
   cursor = conn.cursor ()

   #set transaction isolation level
   cursor.execute("SET TRANSACTION READ ONLY,  ISOLATION LEVEL READ COMMITTED");
   cursor.execute(
        "SELECT model,  price FROM PC  WHERE abs(speed-%s)<0.001 and ram=%s",
         (speed, ram) );

   results = cursor.fetchall()
   for r in results : print r[0], r[1]

   cursor.close()
   conn.close()
```

Passing Info to Query

- Execute Query with Program Info:

```
cursor.execute(
"SELECT model,  price
  FROM PC
  WHERE abs(speed-%s)<0.001
  and ram=%s",
  (1.73, 1024) );
```

Python Tuples

- (1.73, 1024): Python Tuple
  - Enclosed in parens
- Single entry tuple needs ‘,’:
  - (1001,)

Fetching All Results

- We can fetch a single tuple results, or all tuple results:

```
results = cursor.fetchall()
```

- Retrieved results available for program:

```
for r in results : print r[0], r[1]
```