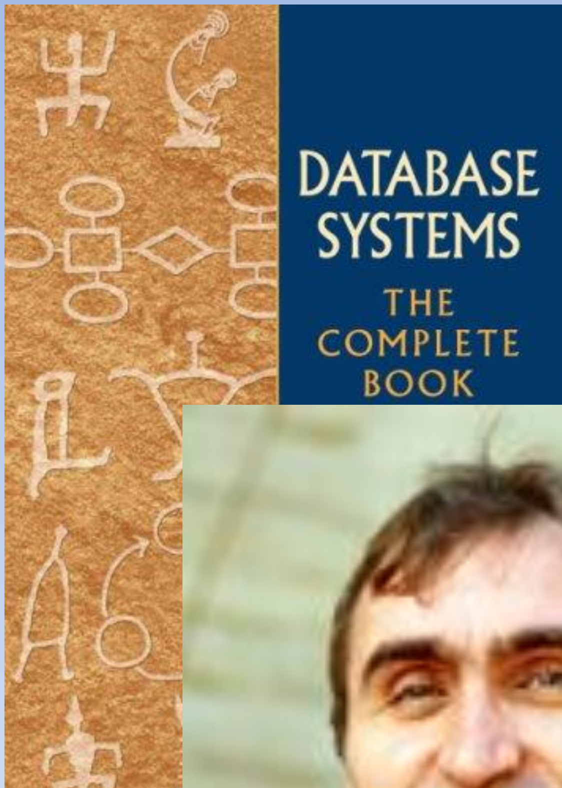
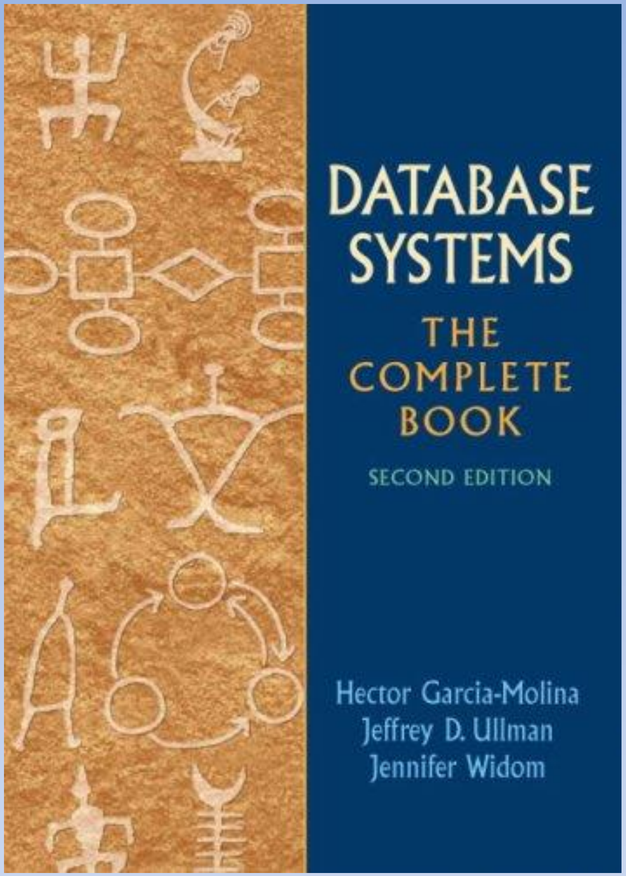
Database Systems: The Complete Book(3^{rd})

by Hector Garcia-Molina,
Jeffrey D. Ullman & Jennifer Widom

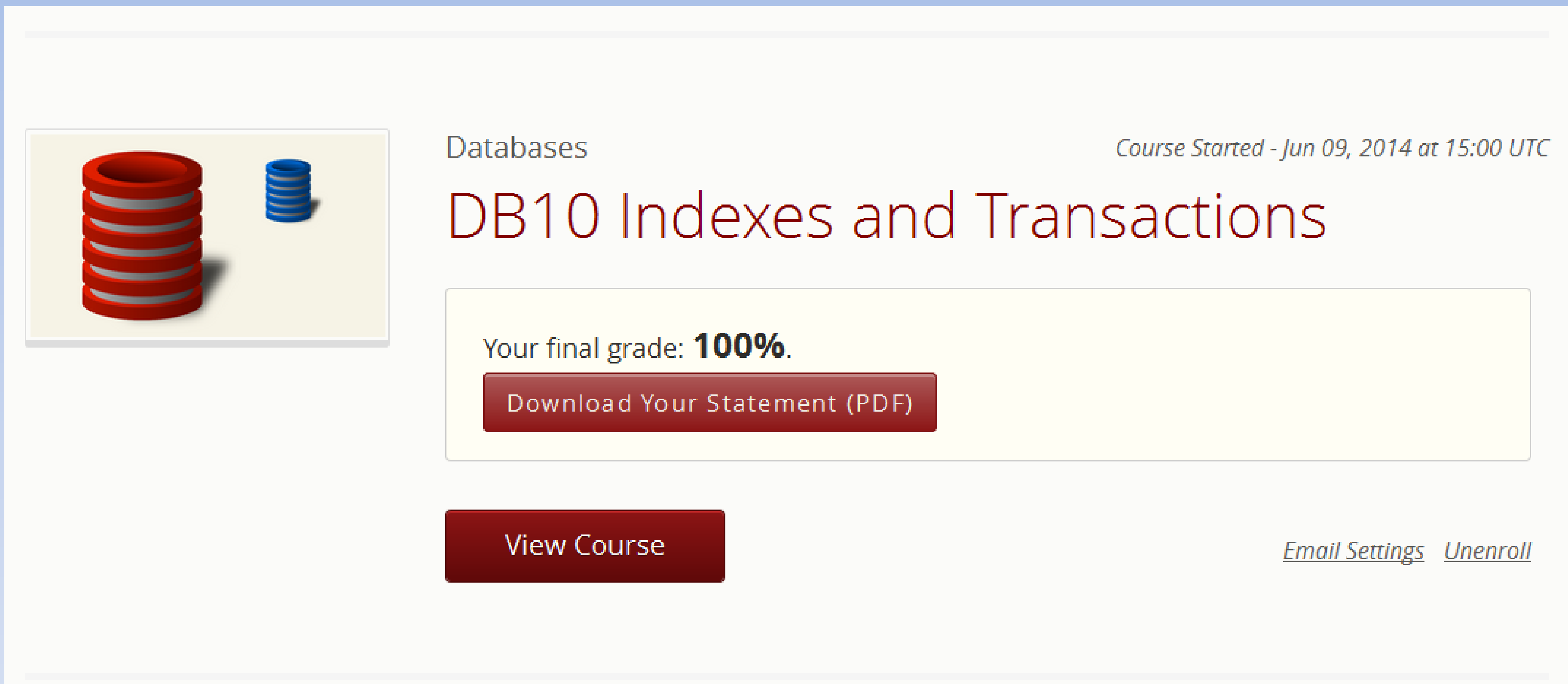




Chapter 6:
End of the Chapter

- Transactions

Stanford Online



Motivation!

- Concurrent access to data.
- Resilience to failure!

Servers w/ Database

- Many Client apps operating concurrently
- Many Users accessing data concurently
- Leads to many kinds of issues

Attribute Level Inconsistencies Ships Domain Example

- Updates can arrive concurrently from multiple users/applications.
- Two Database Clients making changes at Sametime

- Client 1: Modify the ‘Classes ‘ relation so that gun bores are measured in inches (one inch = 2.5 centimeters) and displacements are measured in tons (one metric ton = 1.1 tons).

- Client 2: Correct an error (dropped fixed weight) with Tennessee class, by adjusting its displacement by an extra 600.

Inconsistencies
Attribute Level Inconsistencies

- Updates can arrive concurrently from multiple users/applications.

```
CONCURRENT USERS:
(S1) update Classes
set bore = bore/2.5, displacement =
displacement*1.1;

(S2) update Classes
set displacement = displacement + 600
Where class = ‘Tennessee’;
```

Inconsistencies
Attribute Level Inconsistencies

- Updates can arrive concurrently from multiple users/applications.

```
CONCURRENT USERS:
(S1) update Classes
set bore = bore*2.5, displacement = displacement/1.1;

(S2) update Classes
set displacement = displacement + 600
where CLASS = ‘Tennessee’;
```

Class	Stype	Country	numGuns	Bore	displacement
Tennessee	bb	Gt. Britain	8	37.5	29000

Initial State

Get – Modify – Put

- Databases use a model where:
  - First retrieve data from disk into memory: **Get**
  - Modify the data in memory: **Modify**
  - Write the data back to disk: **Put**

Inconsistencies
Attribute Level Inconsistencies

- Updates can arrive concurrently from multiple users/applications.
- S1 & S2 : Get
- S1 & S2 : Modify
- ??? S1 : Put , S2 : Put
- ??? S2 : Put, S1 : Put

```
CONCURRENT USERS:
(S1) update Classes
set bore = bore/2.5, displacement = displacement*1.1;

(S2) update Classes
set displacement = displacement + 600
Where CLASS = 'Tennessee';
```

Class	Stype	Country	numGuns	Bore	displacement
Tennessee	bb	Gt. Britain	8	15	????

Inconsistencies

- Tuple Level Inconsistency
  - Updates from multiple users can try to modify the same tuple depending on the ordering of the Get-Modify-Put operations for each update.

```
CONCURRENT USERS:
(S1) update Ships
set Launched = 1945-05-05 where name = ‘Kearsarge’

(S2) update Ships
set Class = Essex where name = ‘Kearsarge’
```

Tuple Level Inconsistency

Initial State

SName	Class	Launched
Kearsarge	Iowa	04-04-1944

```
CONCURRENT USERS:
(S1) update Ships
set Launched = 1945-05-05 where name = 'Kearsarge'

(S2) update Ships
set Class = Essex where name = 'Kearsarge'
```

GET: S1, S2
MOD: S1, S2
PUT: S1
PUT: S2

SName	Class	Launched
Kearsarge	Essex	04-04-1944

Inconsistencies

- Table Level Inconsistency
  - Updates to a table could depend on data from another table being modified

```
CONCURRENT USERS:
(S1) update Classes
set bore = bore*2.5, displacement =
displacement/1.1;

(S2) update ships where class in
(select class where displacement > 30000)
Set name = concat(‘USS ‘, Name);
```

Inconsistencies

- Multi-Statement Inconsistency
  - Updates to tables could produce inconsistent results from one statement to the next.

```
CONCURRENT USERS:
(S1) Insert into ShipsArchive
Select * from Ships where Launched < 1945;
(S2) select count(*) from Ships;
Select count(*) from ShipsArchive;
```

Concurrency Goal

- LOOKs LIKE statements are running alone.
- Simple Minded Solution: Run them alone!
- BUT: We want to enable as much concurrency as possible to allow for Multi-Processors, Multi-Threaded, Asynchronous I/O.

Resilience

- During Update to DB a system failure could occur.
- Creates a situation where part of database updated, while other parts have not been updated.
- GOAL: Guarantee ALL-OR-NOTHING!

Solution for Two issues Transactions

- Transactions appear to run alone.
- Transactions appear to run completely, or not at all.

Solution for Two issues
Transactions

- A Transaction is a sequence of SQL Statements treated as a unit.
- Transactions start on first SQL Statement
- Transaction continues to a COMMIT.
- Autocommit – turns each statement into a transaction.

Transactions Provide ACID Properties Hold

- Atomicity
- Consistency
- Isolation
- Durability

Transactions Provide ACID Properties Hold

- Atomicity
- Consistency
- **Isolation** : Serializability
- Durability

Serializability

- Transactions can be interleaved
- But: Execution must be equivalent to some sequential ordering

Transactions Provide ACID Properties Hold

- **Durability** : Logging insures systems reamains in tact given system crash.

Transactions Provide ACID Properties Hold

- Atomicity : Each Transaction is All-Or-Nothing
  - ROLLBACK!

Transactions Provide ACID Properties Hold

- Consistency :
  - Each client, each transaction:
    - Can assume all constraints hold when transaction begins
    - Must guarantee all constraints hold when transaction ends
  - Serializability Given:
    - constraints always hold
- Isolation : Serializability
- **Durability** : Logging insures systems reamains in tact given system crash.

Transactions Provide ACID Properties Hold

- Atomicity : Each Transaction is All-Or-Nothing
  - ROLLBACK!
- Consistency :
  - Each client, each transaction:
    - Can assume all constraints hold when transaction begins
    - Must guarantee all constraints hold when transaction ends
  - Serializability Given:
    - constraints always hold
- Isolation : Serializability
- **Durability** : Logging insures systems reamains in tact given system crash.

Levels of Isolation

- Most Drastic Level of Isolation:
  - **Serializable:**
  - Operations may be interleaved,
  - BUT: execution must be equivalent to *some* sequential (serial) order of all transactions
- High Overhead
- Reduced Concurrency

Levels of Isolation

- Controlled from the Transaction.
- Perspective from the Transaction.

Dirty Reads

- Dirty Data: written by an Uncommitted Transaction.

```
(S1) update Classes set bore = bore*2.5;

CONCURRENT WITH

(s2) select avg(bore) from Classes;
```

Dirty Reads

- Isolation Level of 'Read Uncommitted'
  - Allows 'Dirty Reads'

```
(S1) :
Start transaction
update Classes set bore = bore*2.5;
Commit;

CONCURRENT WITH

(s2) :
set session transaction isolation level read uncommitted;
Start transaction;
select avg(bore) from Classes;
Commit;
```

Dirty Reads

- Isolation Level of ‘Read Committed’
  - Does not allow ‘Dirty Reads’
  - Does not guarantee serializability

```
(S1) :
Start transaction
update Classes set bore = bore*2.5;
Commit;

CONCURRENT WITH

(s2) :
set session transaction isolation level read committed;
Start transaction;
select avg(bore) from Classes;
Select max(bore) from Classes;
Commit;
```

Dirty Reads

- Isolation Level of 'Repeatable Read'
  - Does not allow 'Dirty Reads'
  - An item read once cannot change value
  - Does not guarantee serializability
  - Phatom Tuples

```
(s1) :
Start transaction
update Classes set bore = bore*2.5;
Update Classes set Displacement = displacement/1.1;
Commit;

CONCURRENT WITH

(s2) :
set session transaction isolation level repeatable read;
Start transaction;
select avg(bore) from Classes;
Select avg(displacement) from Classes;
Commit;
```

MySQL Default Isolation Level

- Default Isolation Level for MySQL is ‘Repeatable Read’
- You can check it by looking at either the global or session default values:

- SELECT @@GLOBAL.tx_isolation, @@tx_isolation;

Exercise 6.6.1

- This and the next exercises involve certain programs that operate on the two relations:
  - Product(maker, model, type)
  - PC(model, speed, ram, hd, price)

  from our running PC exercise. Sketch the following programs, including SQL statements and work done in a conventional language. Do not forget to issue 'BEGIN TRANSACTION', 'COMMIT', and 'ROLLBACK' statements at the proper times and to tell the system your transactions are read-only if they are.

Exercise 6.6.1a

- Given a speed and amount of RAM (as arguments of the function), look up the PC's with that speed and RAM, printing the model number and price of each.

Exercise 6.6.1a

- This transaction is only Reading from DB.
  - Set Transaction: READ ONLY
- Since transaction is only reading, isolation level need only worry about dirty reads.
  - READ COMMITTED provides the optimum ISOLATION LEVEL for concurrency while not allowing dirty reads.

Exercise 6.6.1a

```
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()
```

cursor.close()
conn.close()

Exercise 6.6.1b

- Given a model number, delete the tuple for that model from both PC and Product.

Exercise 6.6.1b

- Given a model number, delete the tuple for that model from both PC and Product.

- The ISOLATION LEVEL can be anything since there is no risk of dirty read (no select statement).

Exercise 6.6.1b

```
#Exercise 6.6.1b
def deleteModel(model):
    # connect
    conn = mysql.connector.connect(
            user="anonymous", passwd="test", database="computers")
    cursor = conn.cursor ()

    #set transaction isolation level and delete
    cursor.execute("SET TRANSACTION ISOLATION LEVEL SERIALIZABLE");
    cursor.execute("DELETE FROM Product where model = %s", (model,) );
    cursor.execute("DELETE FROM PC where model = %s", (model,) );

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

Exercise 6.6.1c

- Given a model number, decrease the price of that model PC by $100.00.

Exercise 6.6.1c

- Given a model number, decrease the price of that model PC by $100.00.

- Again, the ISOLATION LEVEL can be anything since there is no risk of dirty read (no select statement).

Exercise 6.6.1c

```
#Exercise 6.6.1c
def updatePCPrice(model):
    # connect
    conn = mysql.connector.connect(
            user="anonymous", passwd="test", database="computers")
    cursor = conn.cursor ()

    #set transaction isolation level and execute update
    cursor.execute("SET TRANSACTION ISOLATION LEVEL SERIALIZABLE");
    cursor.execute("UPDATE PC set price=price-100 WHERE model=%s", (model,) );

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

Exercise 6.6.1d

- Given a maker, model number, processor speed, RAM size, hard-disk size, and price:
  - Check that there is no product with that model.
  - If there is such a model, print an error message for the user.
  - If no such model existed in the database, enter the information about that model into the PC and Product tables.

Exercise 6.6.1d

- Given a maker, model number, processor speed, RAM size, hard-disk size, and price:
  - Check that there is no product with that model.
  - If there is such a model, print an error message for the user.
  - If no such model existed in the database, enter the information about that model into the PC and Product tables.
- NOT Read Only
  - Reading and Updating
- Read Committed Isolation Level

Exercise 6.6.1d

```
#Exercise 6.6.1d
def insertPC(maker, model, speed, ram, hdd, price):
   # connect
   conn = mysql.connector.connect(user="anonymous", passwd="test", database="computers")
   cursor = conn.cursor ()

   #set transaction isolation level and execute updates
   cursor.execute("SET TRANSACTION ISOLATION LEVEL READ COMMITTED");
   cursor.execute("SELECT 1 FROM Product R WHERE R.model=%s", (model,) );
   if cursor.fetchone() <> None : exist = 1
   else : exists = 0

   if exists == 1:
      print "ERROR:Model No: " + model + " already exists in database"
   else :
#   Add model into database
      cursor.execute("INSERT INTO Product VALUES(%s, %s, 'pc')", (maker, model) )
      cursor.execute("INSERT INTO PC VALUES(%s,%s,%s,%s,%s)", (model, speed, ram, hdd, price) )

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

Exercise 6.6.2a
Atomicity Issues: System Crash

```
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()
```

Exercise 6.6.1b
Atomicity Issues: System Crash

- Given a model number, delete the tuple for that model from both PC and Product.

Exercise 6.6.2b
Atomicity Issues: System Crash

```
#Exercise 6.6.1b
def deleteModel(model):
    # connect
    conn = mysql.connector.connect(
            user="anonymous", passwd="test", database="computers")
    cursor = conn.cursor ()

    #set transaction isolation level and delete
    cursor.execute("SET TRANSACTION ISOLATION LEVEL SERIALIZABLE");
    cursor.execute("DELETE FROM Product where model = %s", (model,) );
    cursor.execute("DELETE FROM PC where model = %s", (model,) );

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

Exercise 6.6.2b
Atomicity Issues: System Crash

- Atomicity Issue IF System crash occurs:
  - AFTER the model was deleted from Product
  - BEFORE deletion from PC
- Databases keep a log of activities
- Log used with some kind of recovery strategy to bring the database to a consistent state on system restart.

Exercise 6.6.2c
Atomicity Issues: System Crash

```
#Exercise 6.6.1c
def updatePCPrice(model):
    # connect
    conn = mysql.connector.connect(
            user="anonymous", passwd="test", database="computers")
    cursor = conn.cursor ()

    #set transaction isolation level and execute update
    cursor.execute("SET TRANSACTION ISOLATION LEVEL SERIALIZABLE");
    cursor.execute("UPDATE PC set price=price-100 WHERE model=%s", (model,) );

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

Exercise 6.6.2c
Atomicity Issues: System Crash

- No atomicity problem here
  - One sql statement and each sql statement is atomic by nature.
- However, if system crashed before update completed:
  - Recall updatePCPrice again

Exercise 6.6.2d

- Given a maker, model number, processor speed, RAM size, hard-disk size, and price:
  - Check that there is no product with that model.
  - If there is such a model, print an error message for the user.
  - If no such model existed in the database, enter the information about that model into the PC and Product tables.

Exercise 6.6.2d

```
#Exercise 6.6.1d
def insertPC(maker, model, speed, ram, hdd, price):
   # connect
   conn = mysql.connector.connect(user="anonymous", passwd="test", database="computers")
   cursor = conn.cursor ()

   #set transaction isolation level and execute updates
   cursor.execute("SET TRANSACTION ISOLATION LEVEL READ COMMITTED");
   cursor.execute("SELECT 1 FROM Product R WHERE R.model=%s", (model,) );
   if cursor.fetchone() <> None : exist = 1
   else : exists = 0

   if exists == 1:
      print "ERROR:Model No: " + model + " already exists in database"
   else :
#     Add model into database
      cursor.execute("INSERT INTO Product VALUES(%s, %s, 'pc')", (maker, model) )
      cursor.execute("INSERT INTO PC VALUES(%s,%s,%s,%s,%s)", (model, speed, ram, hdd, price) )

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

Exercise 6.6.2d
Atomicity Issues: System Crash

- Similar to (b)
- Atomicity Issue IF System crash occurs:
  - AFTER Insert into Product
  - BEFORE Insert into PC
- Databases keep a log of activities
- Log used with some kind of recovery strategy to bring the database to a consistent state on system restart.

Exercise 6.6.3a

- Given transaction:
  - Given a speed and amount of RAM (as arguments of the function), look up the PC’s with that speed and RAM, printing the model number and price of each.
- What behaviors may be seen given other transactions running at sametime with isolation level:
  - READ UNCOMMITTED
  - SERIALIZABLE

Exercise 6.6.3a

```
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()
```

Exercise 6.6.3a w/ Another READ ONLY

- Transaction from 6.6.1 (a) is READ ONLY.
- Another READ ONLY transaction (like lookUpPC) can run concurrently without any difference (i.e. As if all transactions ran in SERIALIZABLE isolation).

Exercise 6.6.3a w/ deleteModel

- Possible issue if deleteModel running concurrently & ROLLBACK occurs.

Exercise 6.6.3a
w/ deleteModel

- Possible issue if deleteModel running concurrently & ROLLBACK occurs.
- If deleteModel from 6.6.1 (b) was running concurrently with lookUpPC, lookUpPC may not return a PC model when:
  - It had been deleted from Product by deleteModel,
  - THEN deleteModel does ROLLBACK.
- With SERIALIZABLE isolation, lookUpPC would return the PC model unless the delete transaction committed.

Exercise 6.6.3a w/ updatePCPrice

- Possible issue if updatePCPrice running concurrently & ROLLBACK occurs.
- If updatePCPrice from 6.6.1 (c) was running concurrently with lookUpPC, lookUpPC may return the reduced price (**dirty read**) even if ROLLBACK occurs.
- With SERIALIZABLE isolation, lookUpPC would always return correct price.

lookUpPC w/ insertPC
The Phantom Tuple

- Possible issue if insertPC running concurrently with lookUpPC.
- If insertPC from 6.6.1 (d) was running concurrently with lookUpPC, lookUpPC may return the inserted tuple even if ROLLBACK occurs .
  - The tuple that would never end up in the database, even though returned by lookUpPC is a P**hantom Tuple**.
- With SERIALIZABLE isolation, lookUpPC would never see the Phantom Tuple.

Exercise 6.6.3b: deleteModel w/ insertPC

- If running insertPC concurrently with T,
  - insertPC finds model does not exist after just deleted,
  - deletePC rolled back, model exists again.
  - Now, insertPC attempts to insert a model that already exists.

Exercise 6.6.3c: updatePCPrice w/ updatePCPrice

- updatePCPrice could read the updated price (dirty data) and decrement model price by $100.
- But then first updatePCPrice rolled back.
- However, the pc price for the model was reduced by $200 though only one updatePCPrice completed.

Exercise 6.6.3d: insertPC w/ insertPC

- When running concurrently with another insertPC, both could check that there is no product with the model, and then try to insert the model.

Properties of SQL Isolation Levels

Isolation Level	Dirty Reads	Nonrepeatable Reads	Phantoms
Read Uncommitted	Allowed	Allowed	Allowed
Read Committed	Not Allowed	Allowed	Allowed
Repeatable Read	Not Allowed	Not Allowed	Allowed
Serializable	Not Allowed	Not Allowed	Not Allowed

Exercise 6.6.4

- Transaction T is a Function that runs ‘forever’
- Each Hour T checks for and prints a PC if both:
  - PC has a speed of 3.5 or more
  - PC sells for under $1000.00
- Describe the transaction behavior with each Isolation Level

Exercise 6.6.4 w/ Serializable

- Transaction T is a Function that runs ‘forever’
- Each Hour T checks for and prints a PC if both:
  - PC has a speed of 3.5 or more
  - PC sells for under $1000.00

- T will never see changes to the database and keep printing the same list of PCs.
- This does not serve any useful purpose.
- Application may need to periodically stop T and then restart it to see data committed in the meantime.

Exercise 6.6.4 w/ Repeatable Read

- Transaction T is a Function that runs ‘forever’
- Each Hour T checks for and prints a PC if both:
  - PC has a speed of 3.5 or more
  - PC sells for under $1000.00

- T will continue to see the list of PCs it saw once.
- However, T will also see any new PCs that are inserted in the database.
- Locking issues can occur if another transaction such as 6.6.1 (b) or (c) tries to update/delete the rows read by T.
- 6.6.1 (d) inserts a new row and thus can run concurrently with T.

Exercise 6.6.4 w/ Read Committed

- Transaction T is a Function that runs ‘forever’
- Each Hour T checks for and prints a PC if both:
  - PC has a speed of 3.5 or more
  - PC sells for under $1000.00

- Perhaps the best option.
- T can see new or updated rows after other transactions such as 6.6.1 (c) or (d) commit.
- However, if T reads the same table twice, the results are not consistent because some rows may have been updated (6.6.1 (c) or deleted(6.6.1 (b)) by other transaction.
- Moreover, if T reads a row and based on the result then tries to read/update/delete the row; the state of row may have changed in the meantime.

Exercise 6.6.4 w/ Read Uncommitted

- Transaction T is a Function that runs ‘forever’
- Each Hour T checks for and prints a PC if both:
  - PC has a speed of 3.5 or more
  - PC sells for under $1000.00

- T will not cause any locking (high concurrency)
- but uncommitted PC data might be printed out due to insert/update by other transaction e.g. 6.6.1 (c) or (d).
- However, the other transaction might rollback resulting in wrong reports.

Project

- Due: 4/30
- Worth: 200 Points
- Broken down into 7 sections

Part 1: Domain Description

- The first requirement for the project is a clear description of the domain that the data is being drawn from.

Part 2: Database Design

- Design your database using E/R diagrams and be sure to include entities and relations of the following types:
  - One-to-One
  - Many-to-Many
  - One-to-Many
  - Weak-Entity Set or ISA Hierarchy

Part 3: Database Schema

- Develop your design into a database schema for your database, and include an .SQL file for creating this schema, and loading the test data into the schema. Be sure to include:
  - Key Definitions
  - Referential Integrity Constraints
  - Triggers
  - Stored Procedures

Part 4: Normal Forms

- Include the Functional Dependencies and any Multi-Valued Dependencies for your database and state whether they are free from violations for:
  - 3rd Normal form
  - Boyce-Codd Normal Form
  - 4th Normal Form.

Part 5: Queries

- Describe queries of use for your database. Then include a set of test queries for me to execute that include:
  - Subqueries
  - Aggregation
  - Insert Queries
  - Update Queries
  - Queries demonstrating Triggers & Stored Procedures

Part 6: Application/Use Cases

- Include information about the applications run or will run on your database.
- Include at least one example of a Data Mining opportunity with your domain.

Part 7: Future Work

- Give a short description of the future plans for your project.