Managing Multiuser Databases

Database Processing:

Fundamentals, Design, and Implementation

David M. Kroenke's

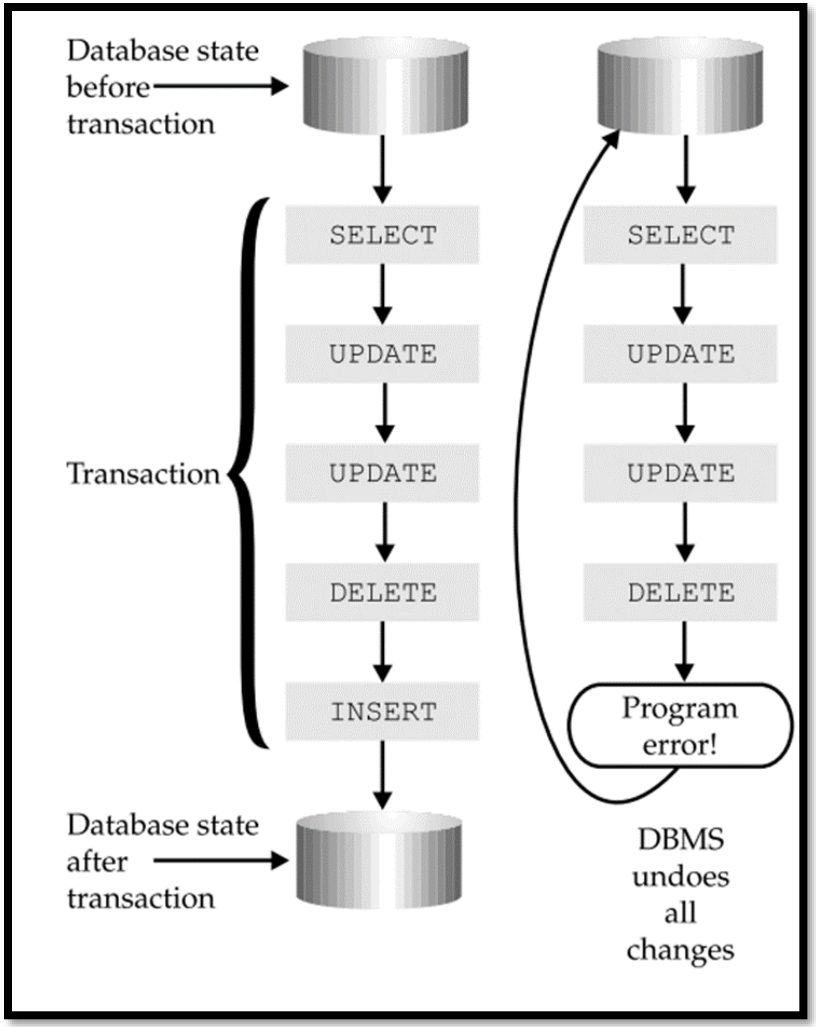
Oct-Nov, 2023

Concurrency Control

- Most challenging problem in the multiuser DB environment
- Many users attempt to access the same data resource simultaneously and each user thinks that he/she has sole use of the computer system
- **Concurrency control** ensures that one user's work does not inappropriately influence another user's work
  - No single concurrency control technique is ideal for all circumstances
  - Trade-offs need to be made between level of protection and throughput

Transactions

- Each user submits work in units called transaction
- Also called "Atomic transactions" or "Logical Unit of Work (LUW)"
- Each transaction is a series of actions taken against the database that occurs as an atomic unit
  - Either all actions in a transaction occur or none of them do



Transactions

- Ex: transaction to enter customer order actions:
    1. Change customer record with new order data
    2. Change salesperson record with new order data
    3. Insert a new order record into the database

- Now visualize the errors introduced without atomic transaction

Errors Introduced Without Atomic Transaction

Before

CUSTOMER

CNum	OrderNum	Description	AmtDue
123	1000	400 Baseballs	$2400

SALESPERSON

Name	Total-Sales	Commission Due
JONES	$3200	$320

ORDER

OrderNum	
1000	. . .
2000	. . .
3000	. . .
4000	. . .
5000	. . .
6000	. . .
7000	. . .

FULL

Action

START

1. Add new-order data to CUSTOMER.
2. Add new-order data to SALESPERSON.
3. Insert new ORDER.

STOP

After

CUSTOMER

CNum	OrderNum	Description	AmtDue
123	1000	400 Baseballs	$2400
123	8000	250 Basketballs	$6500

SALESPERSON

Name	Total-Sales	Commission Due
JONES	$9700	$970

ORDER

OrderNum	
1000	. . .
2000	. . .
3000	. . .
4000	. . .
5000	. . .
6000	. . .
7000	. . .

FULL

Courtesy: David M. Kronke

Errors Prevented With Atomic Transaction

Before

CUSTOMER

CNum	OrderNum	Description	AmtDue
123	1000	400 Baseballs	$2400

SALESPERSON

Name	Total-Sales	Commission Due
JONES	$3200	$320

ORDER

OrderNum	
1000	. . .
2000	. . .
3000	. . .
4000	. . .
5000	. . .
6000	. . .
7000	. . .

FULL

Transaction

Begin Transaction
  Change CUSTOMER data
  Change SALESPERSON data
  Insert ORDER data
If no errors then
  Commit Transactions
Else
  Rollback Transaction
End If

The commands marking transaction logic boundaries must be issued by the application program

After

CUSTOMER

CNum	OrderNum	Description	AmtDue
123	1000	400 Baseballs	$2400

SALESPERSON

Name	Total-Sales	Commission Due
JONES	$3200	$320

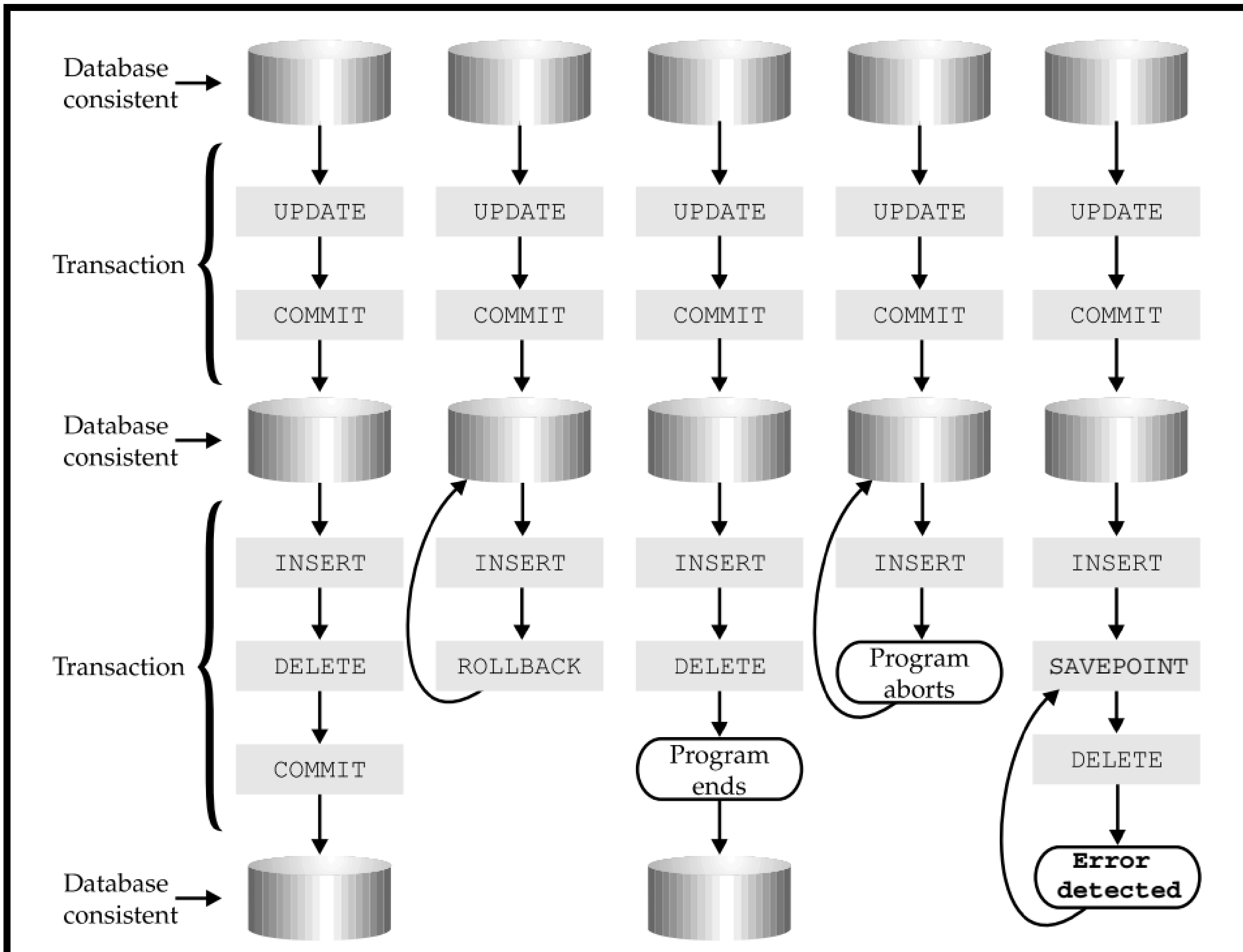
ORDER

OrderNum	
1000	. . .
2000	. . .
3000	. . .
4000	. . .
5000	. . .
6000	. . .
7000	. . .

FULL

Courtesy: David M. Kronke

Committed and Rolled Back Transactions



Concurrent Transaction

- **Concurrent transactions** refer to two or more transactions that appear to users as they are being processed against a database at the same time
- In reality, CPU can execute only 1 instruction at a time
  - **Transactions are interleaved** meaning that the operating system quickly switches CPU services among tasks so that some portion of each of them is carried out in a given interval
- Concurrency problems:
  - lost update or inconsistent reads

Concurrent Transaction Processing

User A

1. Read item 100
2. Change item 100
3. Write item 100

User B

1. Read item 200
2. Change item 200
3. Write item 200

Order of Processing at DB server

1. Read item 100 for A
2. Read item 200 for B
3. Change item 100 for A
4. Write item 100 for A
5. Change item 200 for B
6. Write item 200 for B

Lost-Update Problem

User A

1. Read item 100 (Item count is 10)
2. Reduce count of items by 5
3. Write item 100

User B

1. Read item 100 (Item count is 10)
2. Reduce count of items by 3
3. Write item 100

Order of Processing at DB server

1. Read item 100 from A (item count = 10)
2. Read item 100 from B (item count = 10)
3. Set item count to 5 (for A)
4. Write item 100 for A
5. Set item count to 7(for B)
6. Write item 100 for B

concurrent update problem

* Note the change and write in step 3 and 4 are lost

Inconsistent Read Problem

- Both users obtained current data from the DB
- User B reads data that have been processes by a portion of a transaction from User A
- As a result User B reads incorrect data
- Solution: prevent multiple applications from having copies of the same record when the record is about to be changed
- Resource Locking

Resource Locking

- **Resource locking** prevents multiple applications from obtaining copies of the same record when the record is about to be changed
- Disallow sharing by locking data that are retrieved for update
- Use **Locks**

Concurrent Processing with Explicit Locks

User A
1. Lock item 100
2. Read item 100
3. Reduce count by 5
4. Write item 100

User B
1. Lock item 100
2. Read item 100
3. Reduce count by 3
4. Write item 100

Order of Processing at DB server
1. Lock item 100 for A
2. Read item 100 from A
3. Lock item 100 for B; cannot so place B in wait state
4. set item count to 5 (for A)
5. Write item 100 for A
6. Release A's lock on item 100
7. Place lock on item 100 for B
8. Read item 100 for B
9. set item count to 2 for B
10. Write item 100 for B
11. Release B's lock on item 100

A's Transaction

B's Transaction

Lock Terminology

- **Implicit locks** are locks placed by the DBMS
- **Explicit locks** are issued by the application program
- **Lock granularity** refers to size of a locked resource: rows, page, table, and database level
- Large granularity:
  - easy to manage but frequently causes conflicts
- Types of locks
  - An **exclusive lock** locks the item from access of any type. No other transaction can read or change the data
  - A **shared lock** allows other users to read the locked resource, but they cannot update it

Serializable Transactions

- **Serializable Transactions** refer to two transactions that run concurrently and generate results that are consistent with the results that would have occurred if they had run separately
- **Two-phased locking** is one of the techniques used to achieve serializability
  - **Growing phase:** Transactions are allowed to obtain locks as necessary
  - **Shrinking phase:** The locks held by the transaction are being released
  - Once the 1st lock is released, no other lock can be obtained

Strict Two-phased Locking

- A special case of two-phased locking
  - Locks are obtained throughout the transaction
  - No lock is released until the COMMIT or ROLLBACK command is issued
  - This strategy is more restrictive but easier to implement than two-phased locking

Deadlock

† Locking solves one problem but introduces another

User A

1. **Lock Paper**
2. **Take Paper**
3. **Lock Pencils**

Deadly Embrace

User B

1. **Lock Pencils**
2. **Take Pencils**
3. **Lock Paper**

Order of Processing at DB server

1. Lock Paper for User A
2. Lock Pencils for User B
3. Process A's request; write paper record
4. Process B's request; write pencil record
5. Put A in wait state for Pencils
6. Put B in wait state for Paper

****Locked****

Deadlock

- **Deadlock** occurs when two transactions are each waiting on a resource that the other transaction holds
- 3 common ways to solve this situation
  - Deadlock Prevention
    - Allow users to issue all lock requests at one time
  - Deadlock Avoidance
    - Require all application programs to lock resources in some predetermined order
  - Breaking deadlock
    - Almost every DBMS has algos for detecting deadlock
    - When deadlock occurs, DBMS aborts one of the transactions and rollbacks partially completed work

Dimensions of Resource Locking

† Lock level (Lock granularity)

† Lock scope

– Number of granules to be locked

† Lock duration

– To end of job or terminal session (maximum)

– For single DBMS action (minimum)

– To end of transaction

† Lock agent (responsibility)

– DBMS function **(more reliable, less flexible)**

– Application program function **(less reliable, more flexible)**

Optimistic versus Pessimistic Locking (two basic styles)

- **Optimistic locking** assumes that no transaction conflict will occur:
  - DBMS processes a transaction, updates are issued, and then checks whether conflict occurred:
    - If not, the transaction is finished
    - If so, the transaction is repeated until there is no conflict
- **Pessimistic locking** assumes that conflict will occur:
  - Locks are issued before a transaction is processed, and then the locks are released

Optimistic Locking

```
SELECT    PRODUCT.Name, PRODUCT.Quantity
FROM      PRODUCT
WHERE     PRODUCT.Name = 'Pencil'

Set NewQuantity = PRODUCT.Quantity – 5

{process transaction – take exception action if NewQuantity < 0, etc.

Assuming all is OK: }

LOCK PRODUCT

UPDATE    PRODUCT
SET       PRODUCT.Quantity = NewQuantity
WHERE     PRODUCT.Name = 'Pencil'
     AND  PRODUCT.Quantity = OldQuantity

UNLOCK    PRODUCT

{check to see if update was successful;
if not, repeat transaction}
```

Granularity has to be decided

Pessimistic Locking

Granularity has to be decided

```
LOCK      PRODUCT

SELECT    PRODUCT.Name, PRODUCT.Quantity
FROM      PRODUCT
WHERE     PRODUCT.Name = 'Pencil'

Set NewQuantity = PRODUCT.Quantity – 5

{process transaction – take exception action if NewQuantity < 0, etc.

Assuming all is OK: }

UPDATE    PRODUCT
SET       PRODUCT.Quantity = NewQuantity
WHERE     PRODUCT.Name = 'Pencil'

UNLOCK    PRODUCT

{no need to check if update was successful}
```

Optimistic Locking (+ve and -ve points)

- The lock is held for less time than with pessimistic locking
  - More imp in case lock granularity is large
- Transaction may have to be repeated many times
  - Transactions that involve a lot of activity on a given row are poorly suited
- Optimistic locking is preferred for the Internet and for many intranet applications

Declaring Lock Characteristics

† Most application programs do not explicitly declare locks due to its complication

† They mark transaction boundaries and declare locking behavior they want the DBMS to use

  – Transaction boundary markers: BEGIN, COMMIT, and ROLLBACK TRANSACTION

† Advantage:

  – If the locking behavior needs to be changed, only the lock declaration need be changed, not the application program

Marking Transaction Boundaries

```
BEGIN TRANSACTION:

SELECT      PRODUCT.Name, PRODUCT.Quantity
FROM        PRODUCT
WHERE       PRODUCT.Name = 'Pencil'

Old Quantity = PRODUCT.Quantity

Set NewQuantity = PRODUCT.Quantity – 5

{process transaction – take exception action if NewQuantity < 0, etc.}

UPDATE      PRODUCT
SET         PRODUCT.Quantity = NewQuantity
WHERE       PRODUCT.Name = 'Pencil'

{continue processing transaction} . . .


IF transaction has completed normally      THEN

    COMMIT TRANSACTION

ELSE

    ROLLBACK TRANSACTION

END IF

Continue processing other actions not part of this transaction . . .
```

- These boundaries are the essential information that the DBMS needs in order to enforce different locking strategies

Marking Transaction Boundaries

```
BEGIN TRANSACTION:

SELECT      PRODUCT.Name, PRODUCT.Quantity
FROM        PRODUCT
WHERE       PRODUCT.Name = 'Pencil'

Old Quantity = PRODUCT.Quantity

Set NewQuantity = PRODUCT.Quantity – 5

{process transaction – take exception action if NewQuantity < 0, etc.}

UPDATE      PRODUCT
SET         PRODUCT.Quantity = NewQuantity
WHERE       PRODUCT.Name = 'Pencil'

{continue processing transaction} . . .


IF transaction has completed normally         THEN

      COMMIT TRANSACTION

ELSE

      ROLLBACK TRANSACTION

END IF

Continue processing other actions not part of this transaction . . .
```

- If developer now declares (via a system parameter or similar means) that he/she wants optimistic locking, the DBMS will implicitly set the locks in the correct place for that locking style

ACID Transactions

- An **ACID** transaction is one that is:

  Atomic,

  Consistent,

  Isolated, and

  Durable

- **Atomic** means either all of the database actions occur or none of them do
- **Durable** means all committed changes are permanent

ACID Transactions (Consistency)

- **Consistency** means that each user sees a consistent view of the data, including visible changes made by the user's own transactions and transactions of other users
- **Consistency:**
  - **Statement level consistency**
  - **Transaction level consistency**

ACID Transactions (Consistency)

- **Statement level consistency:**
  - Each statement independently processes rows consistently; but
  - Changes from other users to these rows might be allowed during the interval between the two SQL statements
- **Transaction level consistency:**
  - All rows impacted by either of the SQL statements are protected from changes during the entire transaction
    - With transaction level consistency, a transaction may not see its own changes

ACID Transactions (Consistency)

- Statement level consistency
- Transaction level consistency
  - Second SQL statement may not see rows changed by the first SQL statement

```
UPDATE   CUSTOMER
SET      Areacode = ‘425’
WHERE    Zipcode = ‘98050’
COMMIT TRANSACTION
```

```
BEGIN TRANSACTION
UPDATE   CUSTOMER
SET      Areacode = ‘425’
WHERE    Zipcode = ‘98050’

{OTHER TRANSACTION WORK}

UPDATE   CUSTOMER
SET      Discount = 0.05
WHERE    Areacode = ‘425’

{OTHER TRANSACTION WORK}

COMMIT TRANSACTION
```

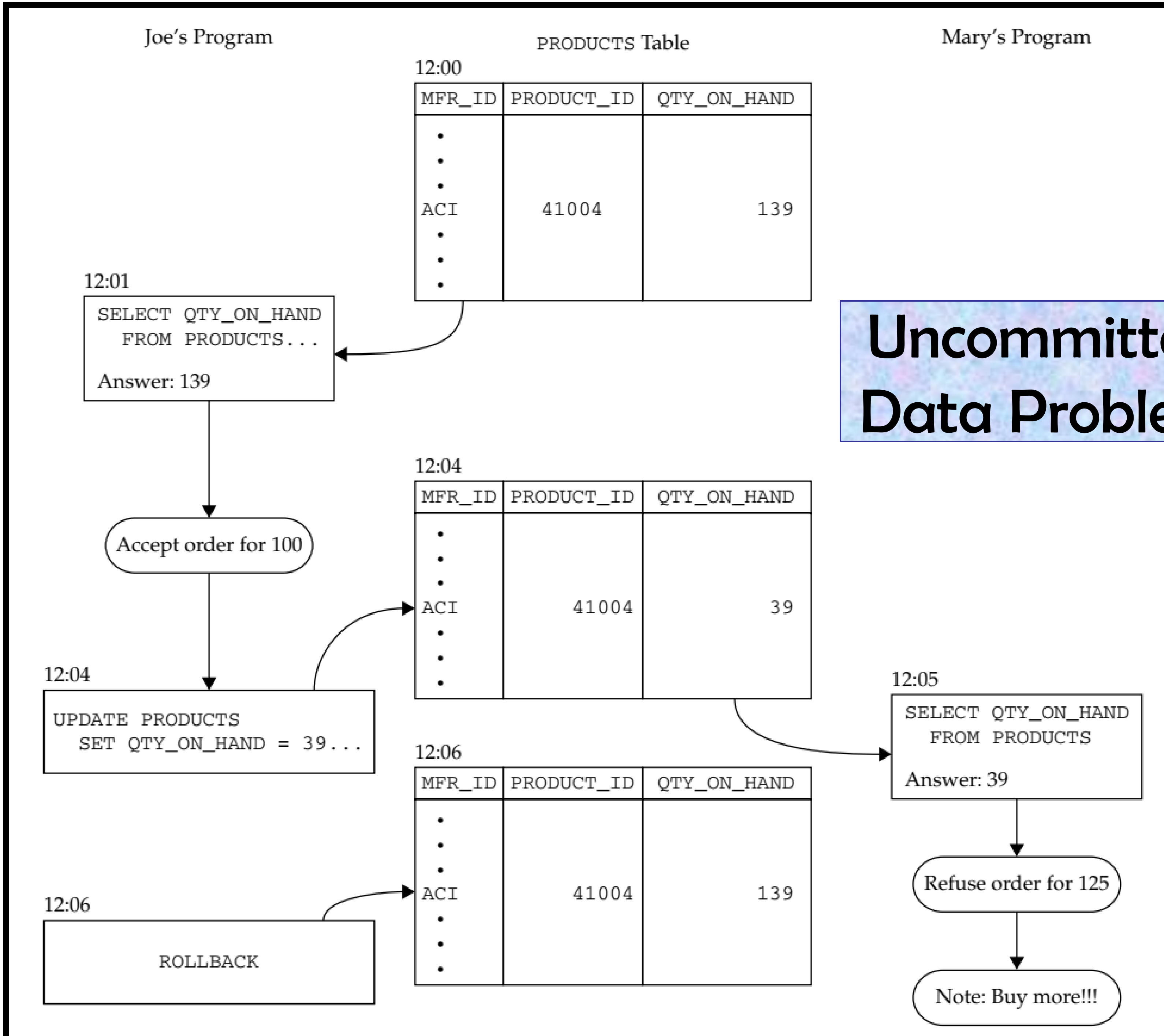
Transactions (isolation)

- DBMS must ensure that the user's actions do not interfere with one another
- Four fundamental problems to be addressed:
  - **Lost update**
  - **Dirty read**
  - **Nonrepeatable reads**
  - **Phantom reads**

ACID Transactions (isolation)

- Dirty read: (Uncommitted Data Problem)
  - One transaction reads a changed record that has not been committed to the db
    - If one transaction reads a row changed by a second uncommitted transaction, and this second transaction later aborts
    - The danger of a dirty read is that the uncommitted change can be rolled back. If so, the transaction that made the dirty read will process incorrect data.

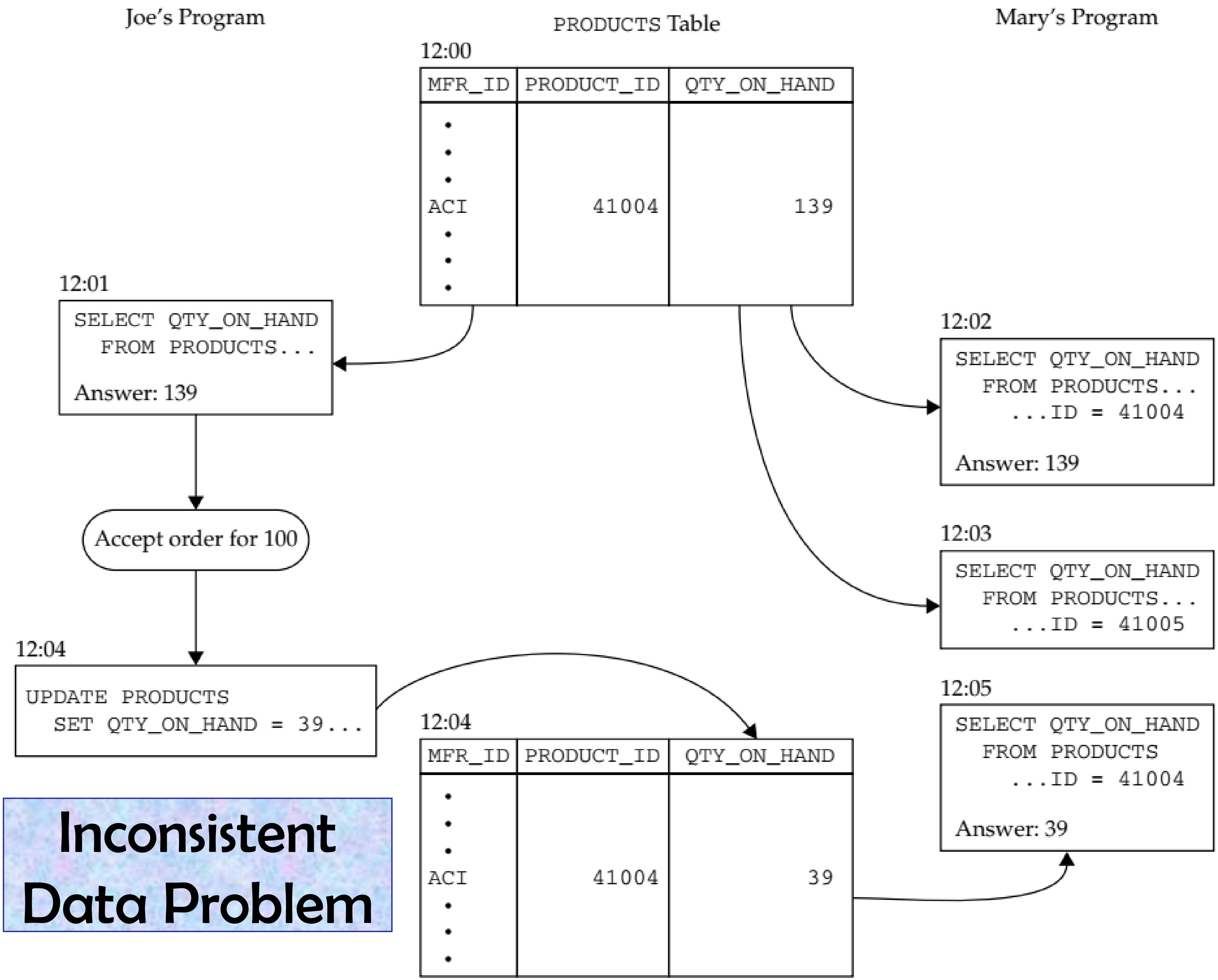
Uncommitted Data Problem



ACID Transactions (isolation)

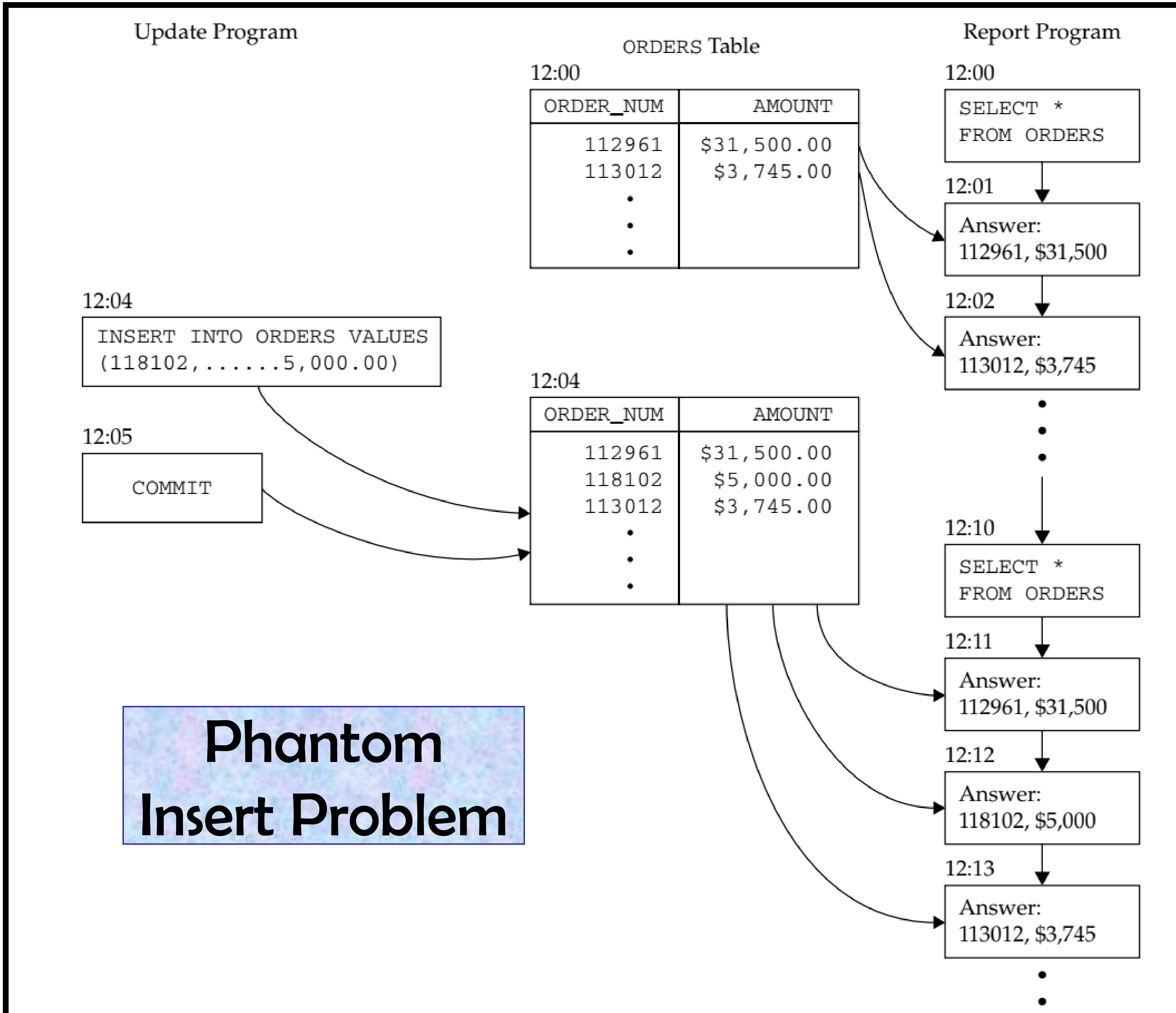
- Nonrepeatable reads: (Inconsistent Data Problem)
  - When a transaction rereads data it has previously read and finds modification or deletions caused by a committed transaction

Inconsistent Data Problem



ACID Transactions (isolation)

- Phantom insert:
  - When a transaction rereads data and finds new rows that were inserted by a committed transaction since the prior read



Update Program
ORDERS Table
Report Program
12:00
ORDER_NUM
AMOUNT
112961
$31,500.00
113012
$3,745.00
12:00
SELECT *
FROM ORDERS
12:01
Answer:
112961, $31,500
12:02
Answer:
113012, $3,745
12:04
INSERT INTO ORDERS VALUES
(118102,......5,000.00)
12:05
COMMIT
12:04
ORDER_NUM
AMOUNT
112961
$31,500.00
118102
$5,000.00
113012
$3,745.00
12:10
SELECT *
FROM ORDERS
12:11
Answer:
112961, $31,500
12:12
Answer:
118102, $5,000
12:13
Answer:
113012, $3,745
Phantom
Insert Problem

ACID Transactions (isolation)

- Under a strict definition of a SQL transaction, no action by a concurrently executing transaction is allowed to impact the data visible during the course of your transaction
- If your program performs a DB query during a transaction, proceeds with other work, and later performs the same DB query a second time, the SQL transaction mechanism guarantees that the data returned by the two queries will be identical (unless your transaction acted to change the data)

ACID Transactions (isolation)

- Absolute isolation of your transaction from all other concurrently executing transaction is very costly in terms of DB locking and loss of DB concurrency
- **Isolation** means application programmers are able to declare the type of isolation level and to have the DBMS manage locks so as to achieve that level of isolation
- SQL 2 defines four transaction isolation levels:
  - **Read uncommitted**
  - **Read committed**
  - **Repeatable read**
  - **Serializable** (default isolation level in ANSI/ISO SQL standard)

Transaction Isolation Level

Isolation Level	**Lost Update**	**Dirty Read (uncommitted data)**	**Nonrepeatable Read (inconsistent data)**	**Phantom Insert**
Read Uncommitted	Prevented by DBMS	Can occur	Can occur	Can occur
Read Committed	Prevented by DBMS	Prevented by DBMS	Can occur	Can occur
Repeatable Read	Prevented by DBMS	Prevented by DBMS	Prevented by DBMS	Can occur
Serializable	Prevented by DBMS	Prevented by DBMS	Prevented by DBMS	Prevented by DBMS

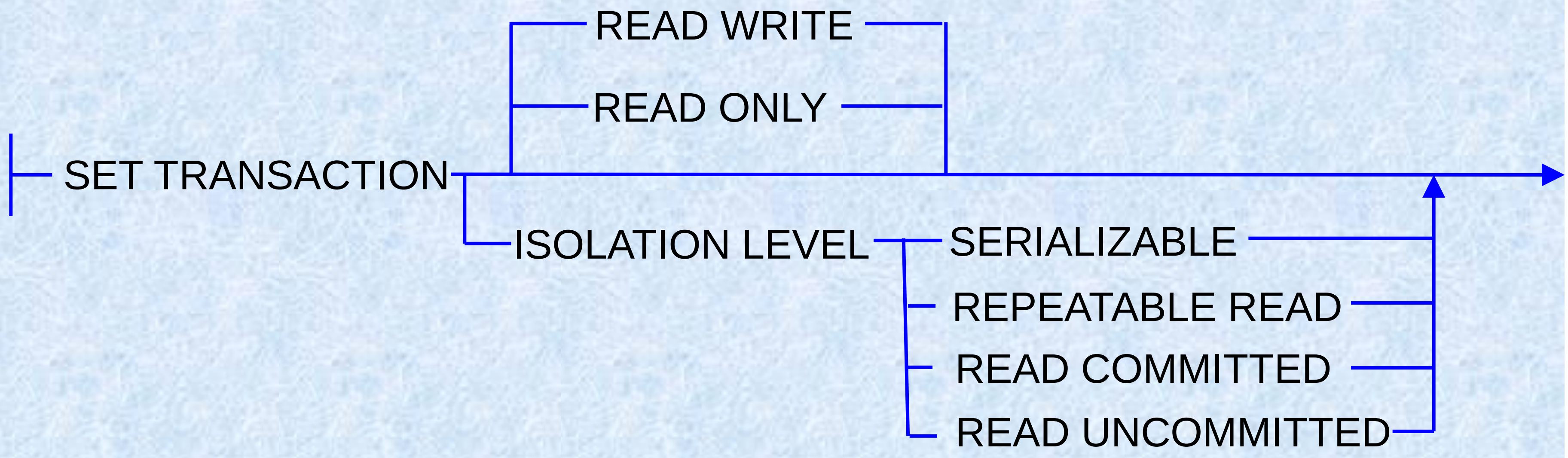
- SQL 2 standard defines 4 isolation levels
- Application programmer declares the type of isolation level he/she wants, and DBMS manages locks so as to achieve that level of isolation

Transaction Isolation Level

Isolation Level	**Lost Update**	**Dirty Read**	**Nonrepeatable Read**	**Phantom Insert**
Read Uncommitted	Prevented by DBMS	Can occur	Can occur	Can occur
Read Committed	Prevented by DBMS	Prevented by DBMS	Can occur	Can occur
Repeatable Read	Prevented by DBMS	Prevented by DBMS	Prevented by DBMS	Can occur
Serializable	Prevented by DBMS	Prevented by DBMS	Prevented by DBMS	Prevented by DBMS

- Generally, more restrictive the level, the less the throughput
- Not all DBMS products support all of these levels

Set Transaction Statement



† http://www.timlin.net/csm/cis363/tranproc.html (transaction processing with MySQL)