

ECE30030/ITP30010 Database Systems

# Transactions

*Reading: Chapters 27*

---

***Charmgil Hong***

charmgil@handong.edu

Spring, 2023

Handong Global University



# Agenda

---

- Transactions
  - Concept and examples
  - Levels of transactions

# Transactions

---

- A **transaction**
  - An **indivisible** “unit” of program execution that accesses and updates data items
    - Indivisible: Either execute entirely or not at all
  - A collection of operations that form a **single logical unit** of work
  - Consists of a sequence of query and/or update statements

# Transactions

---

- Why transactions?
  - Database systems are normally being accessed by **many users or processes** at the **same time**
    - Both queries and modifications
  - Unlike operating systems, which *support* interaction of processes, a DMBS needs to **keep processes from troublesome interactions**

# Transactions

---

- *E.g.*, Bank



# Transactions

---

- *E.g.*, Bank



```
UPDATE accounts  
SET balance = balance + 100  
WHERE accNo = 456;
```

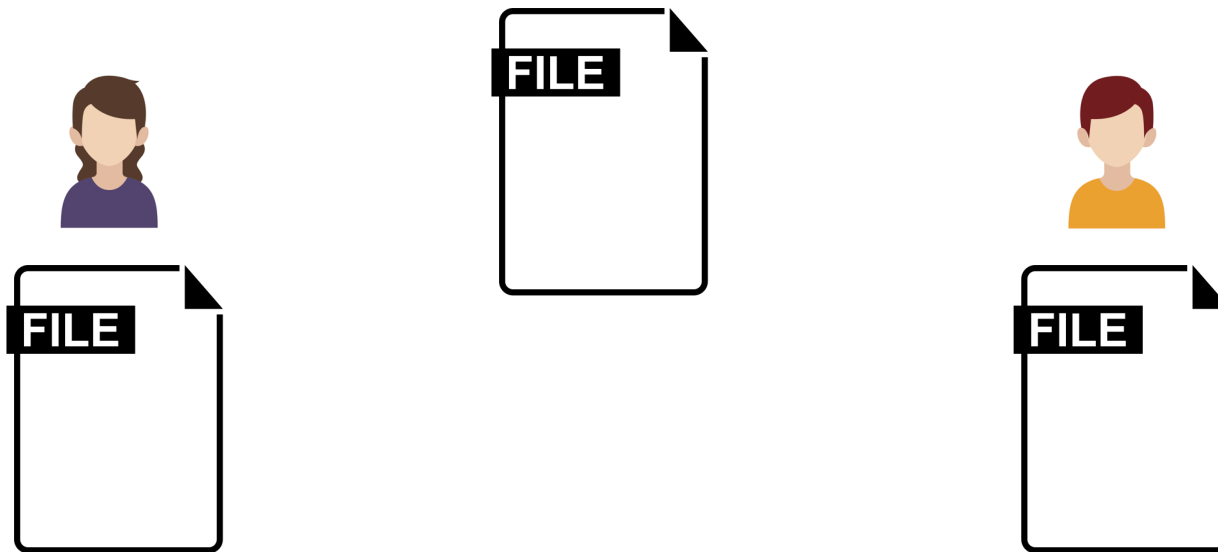


```
UPDATE accounts  
SET balance = balance - 100  
WHERE accNo = 123;
```

# Transactions

---

- *C.f.*, File management in an OS
  - An OS allows two people to edit a document at the same time. **If both write, one's changes get lost**



# Transactions

---

- A **transaction** consists of *a sequence of query and/or update statements* and is a "unit" of work
  - To address both atomicity and serialization, **group database operations into transactions**
  - A transaction is a **collection of one or more operations** on DB that **must be executed atomically**
  - Transactions are executed in a **serializable** manner



# Transactions

---

- **Transaction** = process involving database queries and/or modification
  - A transaction **begins** implicitly **when an SQL statement is executed** (the SQL standard)
  - The transaction must **end** with one of the following statements:
    - **Commit work**: The updates performed by the transaction become **permanent** in the database
    - **Rollback work**: All the updates performed by the SQL statements in the transaction are **undone**

# Transactions

---

- Transactions could be formed by explicit programmer controls  
**START TRANSACTION;**  
...  
**COMMIT;** or **ROLLBACK;**
- **START TRANSACTION** statement is to **declare** that the guarded queries are of a group of operations that **must be executed atomically**
  - Each SQL statement that does not belong to any transaction explicitly is *a transaction with the single statement*

# Transactions

---

- Transactions could be formed by explicit programmer controls  
**START TRANSACTION;**  
...  
**COMMIT;** or **ROLLBACK;**
- **COMMIT** or **ROLLBACK** declares the end of a transaction
  - **COMMIT** causes a transaction to **complete**
    - The database modifications are now permanent in the database
  - **ROLLBACK** ends the transaction by **aborting**
    - No effects on the database
    - Failures like division by 0 or a constraint violation can also cause rollback, even if the programmer does not request it

# Transactions

---

- *E.g.*, Bank



- **START TRANSACTION;**  
**UPDATE** accounts **SET** balance = balance + 100 **WHERE** accNo = 456;  
**UPDATE** accounts **SET** balance = balance - 100 **WHERE** accNo = 123;  
**COMMIT;**

# A Transaction Example

- **START TRANSACTION;**  
**SELECT @A:=SUM(salary) FROM instructor WHERE dept\_name='Comp. Sci.';**  
**UPDATE budget\_summary SET summary=@A WHERE dept\_name='Comp. Sci.';**  
**COMMIT;**

- *C.f., Session variable - @var\_name*
  - Usages
    - **SET @var\_name = value** or **SET @var\_name := value**
    - **@var\_name := value** in a **SELECT** clause
  - Declaration is not required
  - Data type: Defined at the assignment
  - Scope: Until the end of the current session

## Examples

-- Initialize to string

```
SET @id = 'A';
```

```
SELECT CONCAT(@id, 'B');
```

-- Result: AB

-- Assign a number to the

-- same session variable

```
SET @id = 13;
```

```
SELECT @id * 3;
```

-- Result: 39

```
SELECT CONCAT(@id, 'B');
```

-- Result: 13B

\* Source: [http://www.sqlines.com/mysql/session\\_variables](http://www.sqlines.com/mysql/session_variables)

# Another Transaction Example

---

- **SELECT \* FROM sales\_history;**

code	sales	month
A103	101	4
A102	54	5
A104	181	4
A101	184	4
A101	300	5
A103	17	5
A102	200	6
A104	87	6

# Another Transaction Example

---







- `SELECT * FROM sales_history;`
- **START TRANSACTION;**

code	sales	month
A103	101	4
A102	54	5
A104	181	4
A101	184	4
A101	300	5
A103	17	5
A102	200	6
A104	87	6

# Another Transaction Example

---

- `SELECT * FROM sales_history;`
- `START TRANSACTION;`
- **`DELETE FROM sales_history;`**
- **`SELECT * FROM sales_history;`**

 code		 sales		 month	
--	---	---	---	---	---



# Another Transaction Example

---

- `SELECT * FROM sales_history;`
- `START TRANSACTION;`
- `DELETE FROM sales_history;`
- `SELECT * FROM sales_history;`
- **`ROLLBACK;`**
- **`SELECT * FROM sales_history;`**

code	sales	month
A103	101	4
A102	54	5
A104	181	4
A101	184	4
A101	300	5
A103	17	5
A102	200	6
A104	87	6

# A Competing Scenario

---

- Example: Interacting process
  - Assume a usual `Sells(store, chocobar, price)` relation, and suppose that Joe's Store sells only Snickers for \$1.00 and Twix for \$1.50
  - Sally is querying `Sells` for the highest and lowest price Joe charges
  - Joe decides to stop selling Snickers and Twix, but to sell only M&M's at \$2.00



# A Competing Scenario

---

- Example: Interacting process
  - Sally's Program
    - Sally executes the following two SQL statements called (min) and (max) to help us remember what they do
      - (max) `SELECT MAX(price) FROM Sells  
WHERE store = 'Joe''s Store';`
      - (min) `SELECT MIN(price) FROM Sells  
WHERE store = 'Joe''s Store';`
  - Joe's Program
    - At about the same time, Joe executes the following steps: (del) and (ins)
      - (del) `DELETE FROM Sells  
WHERE store = 'Joe''s Store';`
      - (ins) `INSERT INTO Sells  
VALUES('Joe''s Store', 'M&M's', 2.00);`

# A Competing Scenario

---

- Example: Interacting process
  - Interleaving of Statements
    - Although (max) must come before (min), and (del) must come before (ins), there are no other constraints on the order of these statements
      - Unless we group Sally's and/or Joe's statements into transactions
  - Strange interleaving
    - Suppose the steps execute in the order (max)(del)(ins)(min)
      - Joe's Prices: {1.00, 1.50} {1.00, 1.50} {2.00}
      - Statement: (max) (del) (ins) (min)
      - Result: 1.50 2.00
  - Sally sees MAX < MIN

# A Competing Scenario

---

- Example: Interacting process
  - Fixing the Problem by Using Transactions
    - If we group Sally's statements (max)(min) into one transaction, then she cannot see the previous inconsistency
    - She sees Joe's prices at some fixed time
      - Either before or after he changes prices, or in the middle, but the MAX and MIN are computed from the same prices

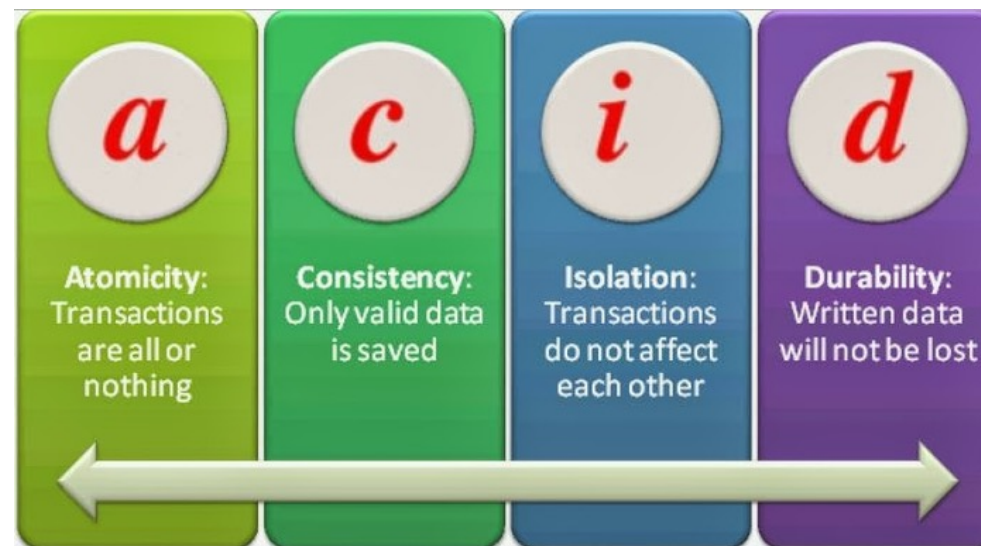
# A Competing Scenario

---

- Example: Interacting process
  - Another Problem: Rollback
    - Suppose Joe executes **(del)(ins)**, not as a transaction, but after executing these statements, thinks better of it and issues a ROLLBACK statement
    - If Sally executes her statements after Joe's **(ins)** but before the rollback, she sees a value, 2.00, that never existed in the database
  - Solution
    - If Joe executes **(del)(ins)** as a transaction, its effect cannot be seen by others until the transaction executes COMMIT
    - If the transaction executes ROLLBACK instead, then its effects can *never* be seen

# ACID Properties

- **ACID** properties
  - **Atomic**: Either fully executed or rolled back as if it never occurred
  - **Consistent**: Database constraints preserved
  - **Isolated**: Isolation from concurrent transactions – It appears to the user as if only one process executes at a time
  - **Durable**: Effects of a process survive a crash



\* Image src: <https://morpheusdata.com/blog/2015-01-29-when-do-you-need-acid-compliance>

# ACID Properties

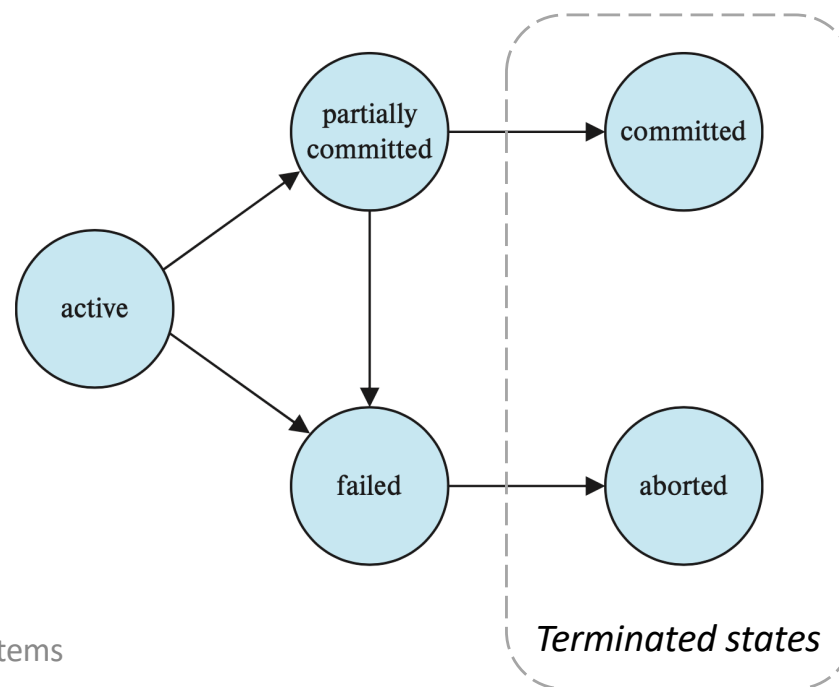
---

- **ACID** properties
  - **Atomic**: Either fully executed or rolled back as if it never occurred
    - The *all-or-none* property
  - **Consistent**: All database constraints should be preserved
  - **Isolated**: Isolation from concurrent transactions – It appears to the user as if only one process executes at a time
    - DBMS must ensure that transactions operate properly without interference from other concurrently executing statements
  - **Durable**: Effects of a process survive a crash
    - The results of transactions must persist in the system



# States of a Transaction

- A transaction must be in one of the following states:
  - **Active**: Initial state; transactions stay in this state while executing
  - **Partially committed**: After the final statement has been executed
  - **Failed**: After the discovery that normal execution can no longer proceed
  - **Aborted**: After the transaction has been rolled back and the database has been restored
  - **Committed**: After successful completion



# A Toy Example

---

- $T_1$ :  
  `read(A);`  
  `A := A - 50;`  
  `write(A);`  
  `read(B);`  
  `B := B + 50;`  
  `write(B)`
- $T_2$ :  
  `read(A);`  
  `temp := A * 0.1;`  
  `A := A - temp;`  
  `read(B);`  
  `B := B + temp;`  
  `write(B)`

# A Toy Example

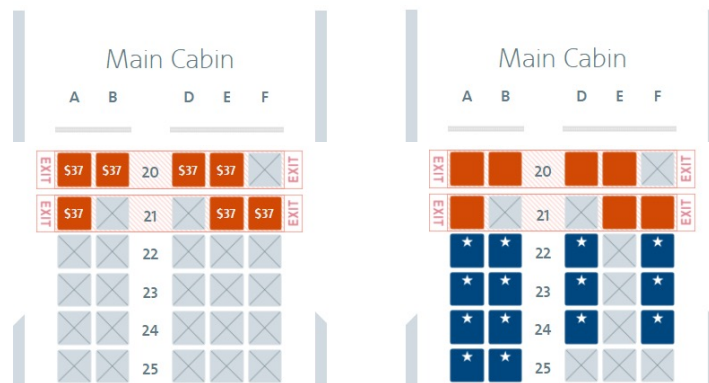
$T_1$	$T_2$	$T_1$	$T_2$	$T_1$	$T_2$
<code>read(A)</code> <code>A := A - 50</code> <code>write(A)</code> <code>read(B)</code> <code>B := B + 50</code> <code>write(B)</code> <code>commit</code>	<code>read(A)</code> <code>temp := A * 0.1</code> <code>A := A - temp</code> <code>write(A)</code> <code>read(B)</code> <code>B := B + temp</code> <code>write(B)</code> <code>commit</code>	<code>read(A)</code> <code>A := A - 50</code> <code>write(A)</code> <code>read(B)</code> <code>B := B + 50</code> <code>write(B)</code> <code>commit</code>	<code>read(A)</code> <code>temp := A * 0.1</code> <code>A := A - temp</code> <code>write(A)</code> <code>read(B)</code> <code>B := B + temp</code> <code>write(B)</code> <code>commit</code>	<code>read(A)</code> <code>A := A - 50</code>  <code>write(A)</code> <code>read(B)</code> <code>B := B + 50</code> <code>write(B)</code> <code>commit</code>	<code>read(A)</code> <code>temp := A * 0.1</code> <code>A := A - temp</code> <code>write(A)</code> <code>read(B)</code>  <code>B := B + temp</code> <code>write(B)</code> <code>commit</code>

# A Toy Example

$T_1$	$T_2$	$T_1$	$T_2$	$T_1$	$T_2$
<code>read(A)</code> <code>A := A - 50</code> <code>write(A)</code> <code>read(B)</code> <code>B := B + 50</code> <code>write(B)</code> <code>commit</code>	<code>read(A)</code> <code>temp := A * 0.1</code> <code>A := A - temp</code> <code>write(A)</code> <code>read(B)</code> <code>B := B + temp</code> <code>write(B)</code> <code>commit</code>	<code>read(A)</code> <code>A := A - 50</code> <code>write(A)</code> <code>read(B)</code> <code>B := B + 50</code> <code>write(B)</code> <code>commit</code>	<code>read(A)</code> <code>temp := A * 0.1</code> <code>A := A - temp</code> <code>write(A)</code> <code>read(B)</code> <code>B := B + temp</code> <code>write(B)</code> <code>commit</code>	<code>read(A)</code> <code>A := A - 50</code> <code>write(A)</code>  <code>read(B)</code> <code>B := B + 50</code> <code>write(B)</code> <code>commit</code>	<code>read(A)</code> <code>temp := A * 0.1</code> <code>A := A - temp</code> <code>write(A)</code>  <code>read(B)</code> <code>B := B + temp</code> <code>write(B)</code> <code>commit</code>

# Serializable Behaviors

- When there are more than one operations **overlap in time**, **affecting the same data source**
  - Each operation could perform correctly
  - While the **global result might not be correct**
- E.g.*, Flight reservation



time

User 1 finds  
seat empty

User 2 finds  
seat empty

User 1 sets seat  
22A occupied

User 2 sets seat  
22A occupied

# Serializable Behaviors

---

- When there are more than one operations **overlap in time, affecting the same data source**
  - Each operation could perform correctly
  - While the **global result might not be correct**
- SQL allows the programmer to state that certain operations must be **serializable** with respect to other operations
  - Operations must behave "as if" they were run *serially* – **one at a time, with no overlap**

# Agenda

---

- Transactions
  - Concept and examples
  - **Levels of transactions**

# Read-Only Transactions

---

- **SET TRANSACTION READ ONLY;**  
**START TRANSACTION;**  
...  
**COMMIT; or ROLLBACK;**
- Declare that the coming transaction reads data from the database, but **never writes**
  - By default, a transaction is set as READ WRITE
- READ ONLY is useful to increase the parallelism of read-only transactions, compared to the regular (read & write) transactions



# Isolation Levels of Transactions

---

- **SET TRANSACTION ISOLATION LEVEL** <level>;  
**START TRANSACTION;**  
...  
**COMMIT;** or **ROLLBACK;**
- Declare kinds of interferences (by other transactions) allowed for a transaction
  - Declare the level of "locking" (enforcing limits on access to) data
  - Tradeoffs: Concurrency  $\rightleftharpoons$  Data integrity

# Isolation Levels of Transactions

---

- SQL supports four isolation levels (*i.e.*, <level>)
  - **SERIALIZABLE (level 3)**
  - **REPEATABLE READ (level 2)** – *the default isolation level in MySQL*
  - **READ COMMITTED (level 1)**
  - **READ UNCOMMITTED (level 0)**
- Higher isolation level  $\Rightarrow$  more data integrity
- Lower isolation level  $\Rightarrow$  more concurrency; higher throughput

# Possible Issues

---

- Phantom read

- Same SELECT queries in **the same transaction** can have different results by **'INSERT'** in another committed transaction

- Nonrepeatable read

- Same SELECT queries in **the same transaction** can have different results by **'UPDATE'** or **'DELETE'** in another committed transaction

- Dirty read

- Read dirty data that is written by an ongoing transaction (not committed yet)

# Isolation Levels of Transactions

---

- **SERIALIZABLE** transactions
  - A serializable transaction must behave with respect to other transactions **as if they were executed one by one** without any parallel execution (*i.e., serially*)
    - While a serializable transaction runs, all data it accesses are locked (other parallel transactions cannot modify nor insert)
  - Serializable is the **most strict** transaction isolation level – guarantees the highest level of data integrity
  - May slow down the transaction handling performance

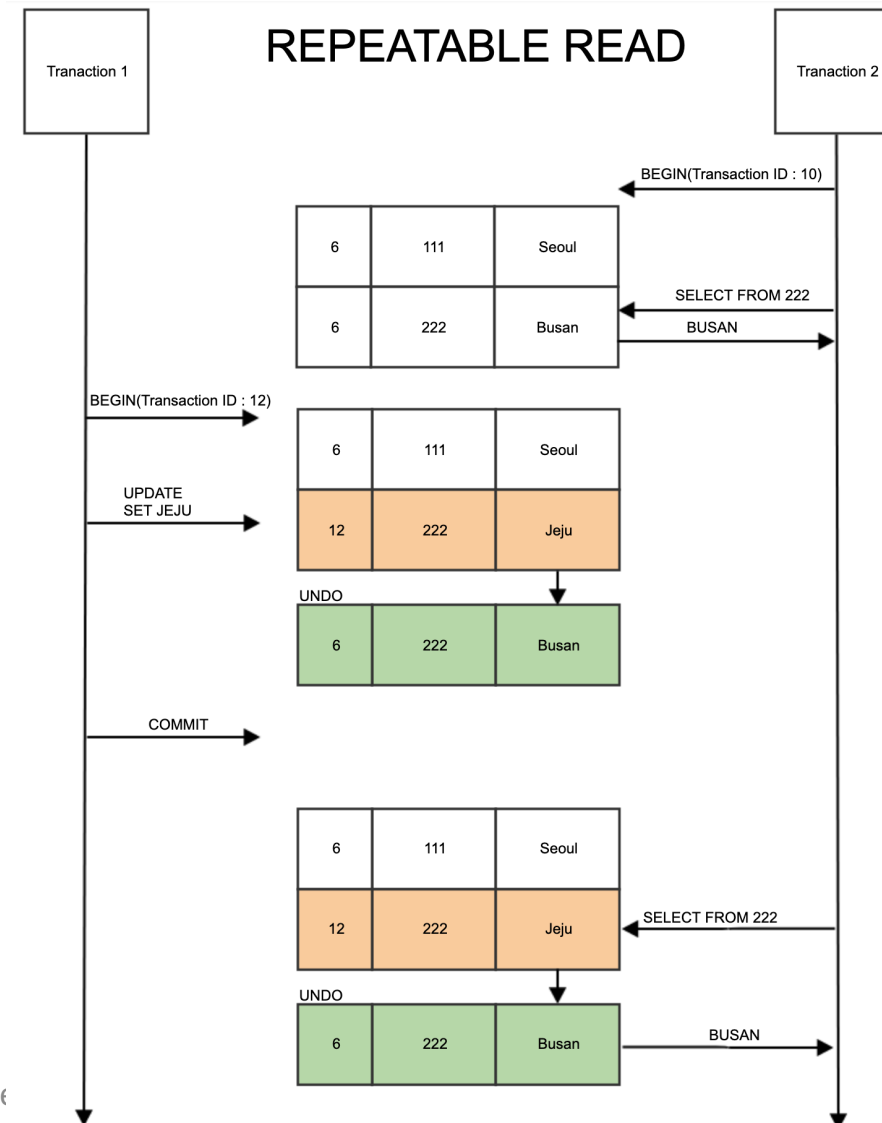
# Isolation Levels of Transactions

---

- **REPEATABLE READ** transactions
  - A repeatable read transaction must see for multiple executions of the same query that a tuple in the first result also appears at the later results
  - A repeatable read transaction must be isolated from the other transaction committed concurrently
    - A repeatable read transaction **only accesses data that has been committed before it starts**
  - The second and the subsequence results of the same query **may have phantom reads**
    - **Other parallel transactions may insert** new tuples in a middle of a transaction, while not changing the existing tuples

# Isolation Levels of Transactions

- **REPEATABLE READ** transactions



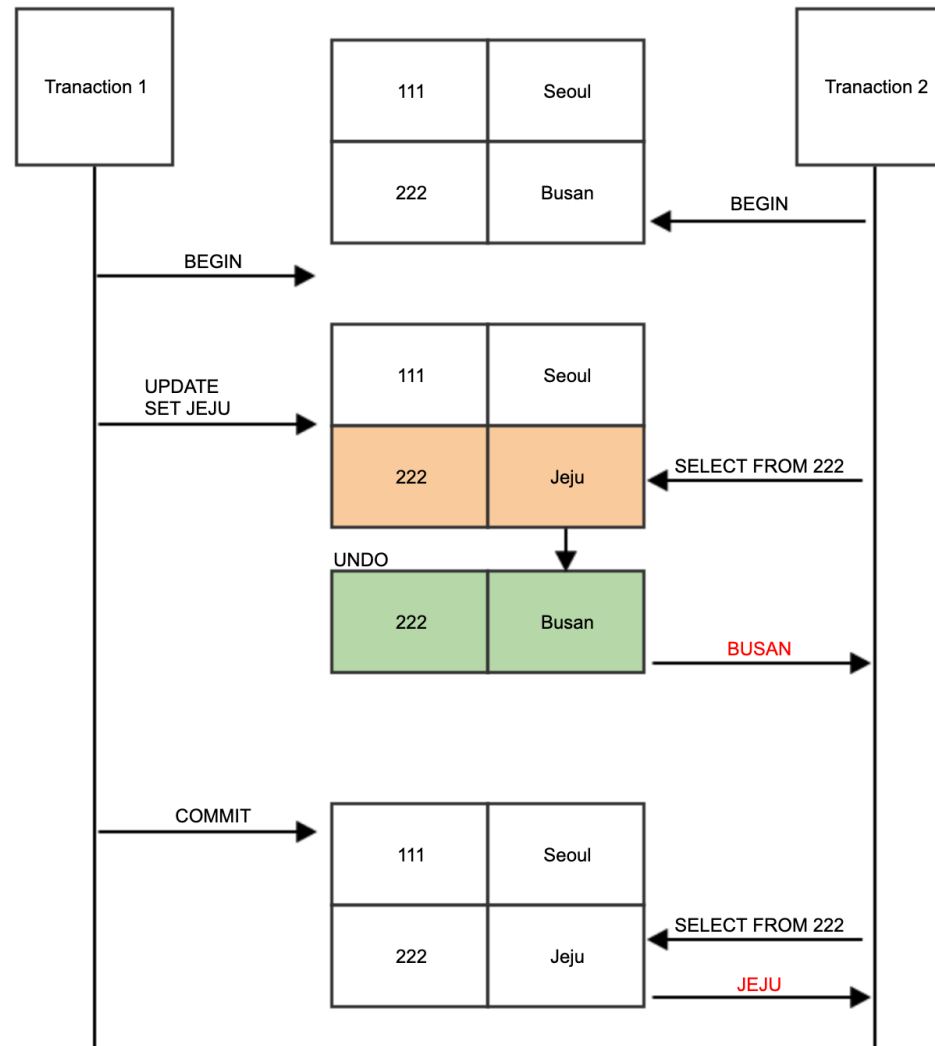
# Isolation Levels of Transactions

---

- **READ COMMITTED** transactions
  - A read committed transaction must see for multiple executions of the same query that a tuple in the first result also appears at the later results
  - A read-committed transaction **must read the databases that are committed**
  - The second and the subsequence results of the same query **may have nonrepeatable reads**
    - **Other parallel transactions may commit changes** in a middle of a transaction, while not changing the existing tuples

# Isolation Levels of Transactions

- **READ COMMITTED** transactions





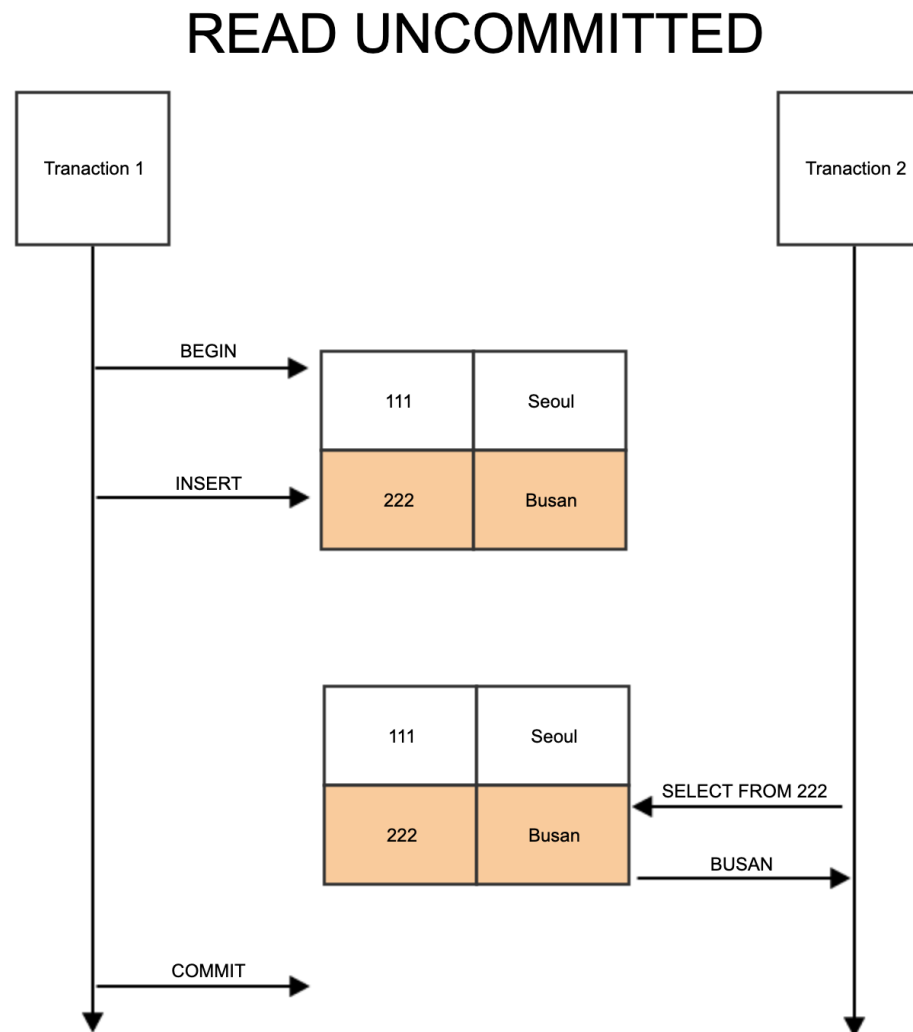
# Isolation Levels of Transactions

---

- **READ UNCOMMITTED** transactions
  - A read-uncommitted transaction **may read dirty data**, the *data written by a transaction that has not yet committed*
    - A dirty data may disappear if its writer transaction aborts
  - A careful use of dirty read **allows fast processing of transactions**
  - Practically there is no isolation; recommended not to use

# Isolation Levels of Transactions

- **READ UNCOMMITTED** transactions



# Possible Issues in Different Isolation Levels

Isolation level	Phantom read	Nonrepeatable read	Dirty read
<b>SERIALIZABLE</b>	Not allowed	Not allowed	Not allowed
<b>REPEATABLE READ</b>	Possible	Not allowed	Not allowed
<b>READ COMMITTED</b>	Possible	Possible	Not allowed
<b>READ UNCOMMITTED</b>	Possible	Possible	Possible

# EOF

---

- Coming next:
  - Stored procedures, functions, triggers