Transactions

SEI.ECNU

華東師範大學
EAST CHINA NORMAL UNIVERSITY

Transaction Concept

- A **transaction** is a unit of program execution that accesses and possibly updates various data items.
- E.g., transaction to transfer $50 from account A to account B:
  1. read(A)
  2. A := A – 50
  3. write(A)
  4. read(B)
  5. B := B + 50
  6. write(B)
- Two main issues to deal with:
  - **Failures** of various kinds, such as hardware failures and system crashes
  - **Concurrent** execution of multiple transactions

Requirements

Atomicity requirement

- If the transaction fails after step 3 and before step 6, money will be “lost” leading to an **inconsistent** database state
- Failure could be due to software or hardware
- The system should ensure that updates of a **partially** executed transaction are not reflected in the database

Durability requirement

- Once the user has been notified that the transaction has completed , the updates to the database by the transaction must **persist** even if there are software or hardware failures.

Requirements

- **Atomicity requirement**
  - If the transaction fails after step 3 and before step 6, money will be “lost” leading to an **inconsistent** database state
  - Failure could be due to software or hardware
  - The system should ensure that updates of a **partially** executed transaction are not reflected in the database
- **Durability requirement**
  - Once the user has been notified that the transaction has completed , the updates to the database by the transaction must **persist** even if there are software or hardware failures.

Requirements

Consistency requirement

- The sum of A and B is **unchanged** by the execution of the transaction
- Explicitly specified integrity **constraints** such as primary keys and foreign keys
- Implicit integrity constraints
- A transaction must see a **consistent** database.
  - During transaction execution the database may be **temporarily** inconsistent.
  - When the transaction completes successfully the database must be consistent
  - Erroneous transaction logic can lead to inconsistency

Requirements

Isolation requirement

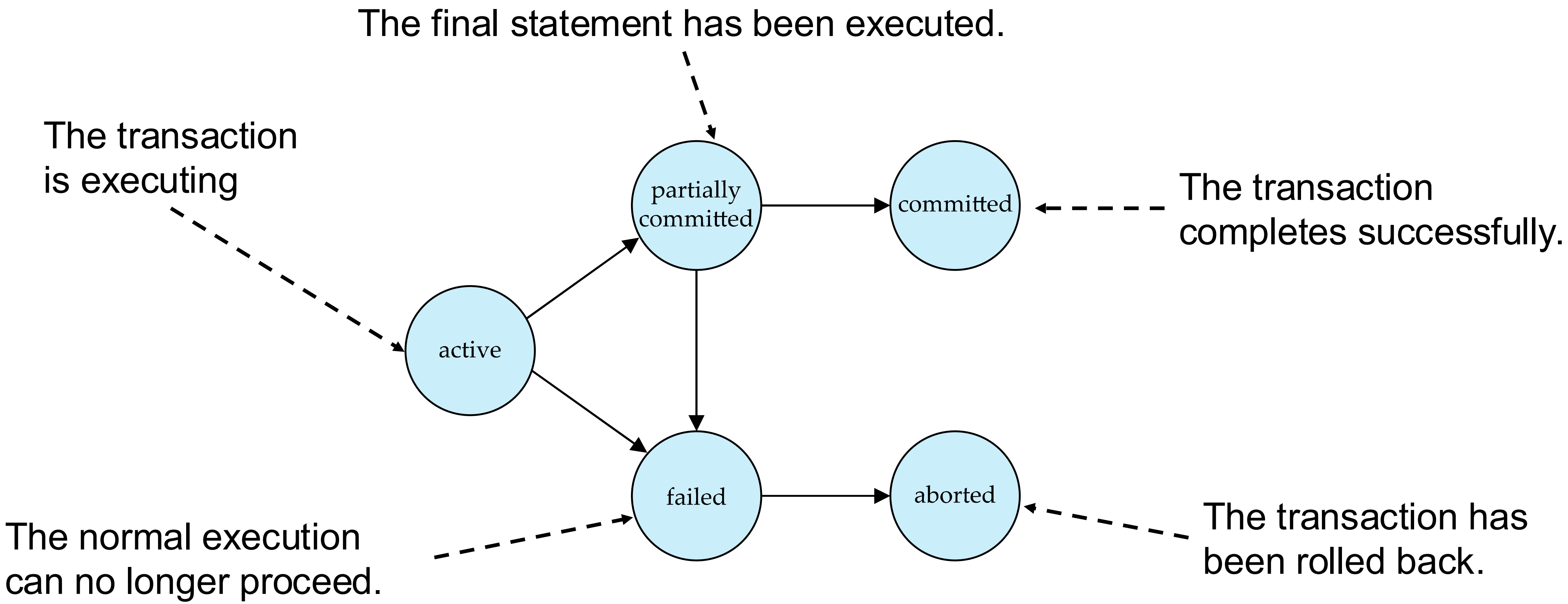
- If T2 is allowed to access the partially updated database, it will see an inconsistent database.
- Isolation can be ensured trivially by running transactions serially

	T1	T2
1.	**read**(A)	
2.	$A := A - 50$	
3.	**write**(A)	
		read(A), read(B), print(A+B)
4.	**read**(B)	
5.	$B := B + 50$	
6.	**write**(B	

ACID Properties

- **Atomicity**
  - Either **all** operations of the transaction are properly reflected in the database or **none** are.
- **Consistency**
  - Execution of a transaction in isolation preserves the **consistency** of the database.
- **Isolation**
  - Each transaction must be **unaware** of other concurrently executing transactions.
  - Intermediate transaction results must be **hidden** from other concurrently executed transactions.
- **Durability**
  - After a transaction completes successfully, the changes it has made to the database **persist**, even if there are system failures.

Transaction State



Concurrent Executions

- Advantages
  - **Increased processor and disk utilization**, leading to better transaction throughput
  - **Reduced average response time** for transactions.
- Concurrency control schemes
  - Control the interaction among the concurrent transactions.

Schedules

- Schedule – **a sequences of instructions** that specify the chronological order in which instructions of concurrent transactions are executed
  - A schedule for a set of transactions must consist of **all instructions** of those transactions
  - Must preserve the **order** in which the instructions appear in each individual transaction.
- A transaction that successfully completes its execution will have a **commit** instructions as the last statement
  - By default transaction assumed to execute commit instruction as its last step
- A transaction that fails to successfully complete its execution will have an **abort** instruction as the last statement

Serial Schedule

T_1 : transfer \$50 from *A* to *B*; T_2 : transfer 10% of the balance from *A* to *B*.

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

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

Concurrent Schedule

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

Equivalent *to Schedule <T_1, T_2>*

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

Not equivalent *to any serial schedule*

Does not preserve *the value of ($A + B$).*

Serializability

- Basic Assumption
  - Each transaction preserves database consistency.
  - Thus, serial execution of a set of transactions preserves database consistency.
- A (possibly concurrent) schedule is **serializable** if it is equivalent to a serial schedule
  - Conflict serializability
  - View serializability

> - Ignore operations other than **read** and **write**
> - Transactions may perform arbitrary computations on data in local buffers

Conflicting Instructions

- Instructions I_i and I_j **conflict**
  - Belong to **different** transactions
  - Access the **same** data item Q
  - At least one of them is **write(Q)**

1.	I_i = **read**(Q), I_j = **read**(Q).	don't conflict.
2.	I_i = **read**(Q), I_j = **write**(Q).	conflict.
3.	I_i = **write**(Q), I_j = **read**(Q).	conflict
4.	I_i = **write**(Q), I_j = **write**(Q).	conflict

- If I_i and I_j do not conflict
  - Their results would remain the same even if they had been **interchanged** in the schedule.

Conflict Serializability

- **Conflict Equivalent**
  - Swap non-conflicting instructions
- **Conflict Serializable**
  - Conflict equivalent to a serial schedule

T_3	T_4
read (Q)	
	write (Q)
write (Q)	

Non-Conflict Serializable

T_1	T_2
read (A) write (A)	
	read (A) write (A)
read (B) write (B)	
	read (B) write (B)

Conflict Serializable

View Serializability

- **View Equivalent**: for each data item Q
  - T_i reads the initial value of Q in both schedules
  - T_i reads the result of the same Write(Q) in both schedules
  - The same transaction performs the final write(Q) in both schedules
- **View Serializable**
  - View equivalent to a serial schedule
  - Every conflict serializable schedule is also view serializable

T_{27}	T_{28}	T_{29}
read (Q)		
	write (Q)	
write (Q)		
		write (Q)

blind writes

Other Notions of Serializability

T_1	T_5
read (A) $A := A - 50$ write (A)	
	read (B) $B := B - 10$ write (B)
read (B) $B := B + 50$ write (B)	
	read (A) $A := A + 10$ write (A)

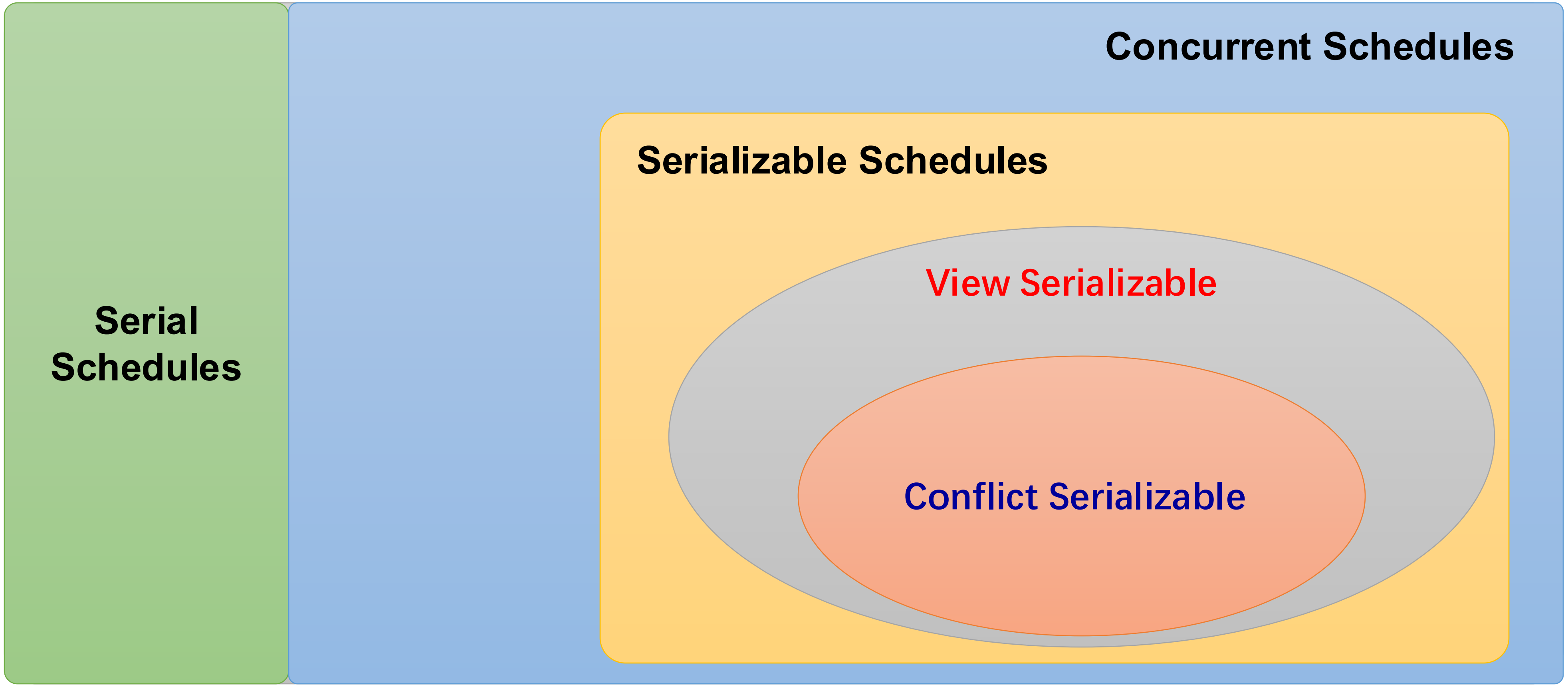
Non-Conflict Equivalent

Non-View Equivalent

Serializability

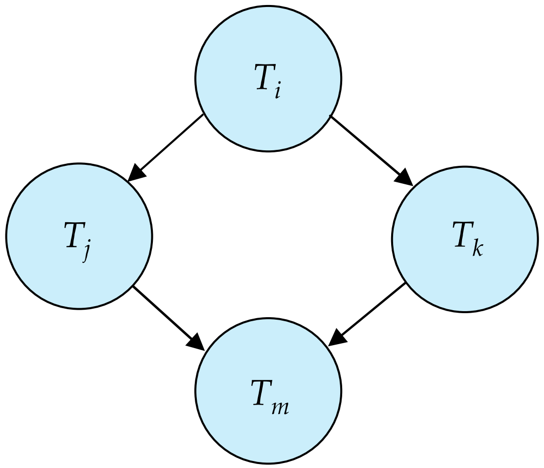
Determining such equivalence requires analysis of operations other than read and write.

Serializability

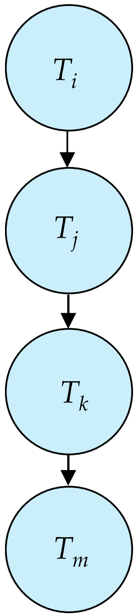


Test for Serializability

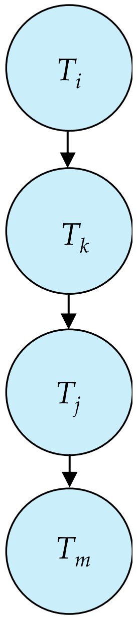
- A schedule is conflict serializable iff its precedence graph is **acyclic**.
- Test for view serializable falls in the class of **NP-complete** problems.



(a)



(b)



(c)

Recoverable Schedules

- If a transaction T_j reads a data item previously written by a transaction T_i , then the commit operation of T_i should appear before the commit operation of T_j.
- Database must ensure that schedules are recoverable

T_8	T_9
read (A)	
write (A)	
	read (A)
	commit
read (B)	

Non-Recoverable

Cascadeless Schedules

- **Cascading rollback**
- Every Cascadeless schedule is also recoverable
- It is desirable to restrict the schedules to those that are cascadeless

T_{10}	T_{11}	T_{12}
read (A)		
read (B)		
write (A)		
	read (A)	
	write (A)	
		read (A)
abort		

If T_{10} fails, T_{11} and T_{12} must also be rolled back

Concurrency Control

- Need a mechanism to ensure all possible schedules are
  - either conflict or view **serializable**, and
  - are **recoverable** and preferably **cascadeless**
- Only one transaction can execute → Poor performance?
- Testing a schedule for serializability after it has executed
  - Too late!
  - Help us understand why a concurrency control protocol is correct
- Goal – to develop **concurrency control protocols** that will assure serializability.

Weak Levels of Consistency

- Some applications are willing to live with weak levels of consistency, allowing schedules that are **not serializable**
  - E.g., a read-only transaction that wants to get an approximate total balance of all accounts
  - E.g., database statistics computed for query optimization can be approximate (why?)
  - Such transactions need not be serializable with respect to other transactions
- **Tradeoff** accuracy for performance

Levels of Consistency in SQL-92

- **Serializable** — default
- **Repeatable read** — only committed records to be read.
  - Repeated reads of same record must return same value.
  - However, a transaction may not be serializable – it may find some records inserted by a transaction but not find others.
- **Read committed** — only committed records can be read.
  - Successive reads of record may return different (but committed) values.
- **Read uncommitted** — even uncommitted records may be read.
- **Snapshot Isolation**

Dirty Read Anomaly

- A transaction is allowed to read data from a row that has been modified by another running transaction and **not** yet committed.

Transaction 1	Transaction 2
/* Query 1 */ SELECT age FROM users WHERE id = 1; /* will read 20 */	
	/* Query 2 */ UPDATE users SET age = 21 WHERE id = 1; /* No commit here */
/* Query 1 */ SELECT age FROM users WHERE id = 1; /* will read 21 */	
	ROLLBACK; /* lock-based DIRTY READ */

Non-repeatable Read Anomaly

- During the course of a transaction, a row is retrieved twice and the values within the row **differ** between reads.

Transaction 1	Transaction 2
/* Query 1 */ SELECT age FROM users WHERE id = 1; /* will read 20 */	
	/* Query 2 */ UPDATE users SET age = 21 WHERE id = 1; COMMIT; /* in MVCC, or lock-based READ COMMITTED */
/* Query 1 */ SELECT age FROM users WHERE id = 1; /* will read 21 */	

Phantom Read Anomaly

- During the course of a transaction, **new** rows are added or removed by another transaction to the records being read.

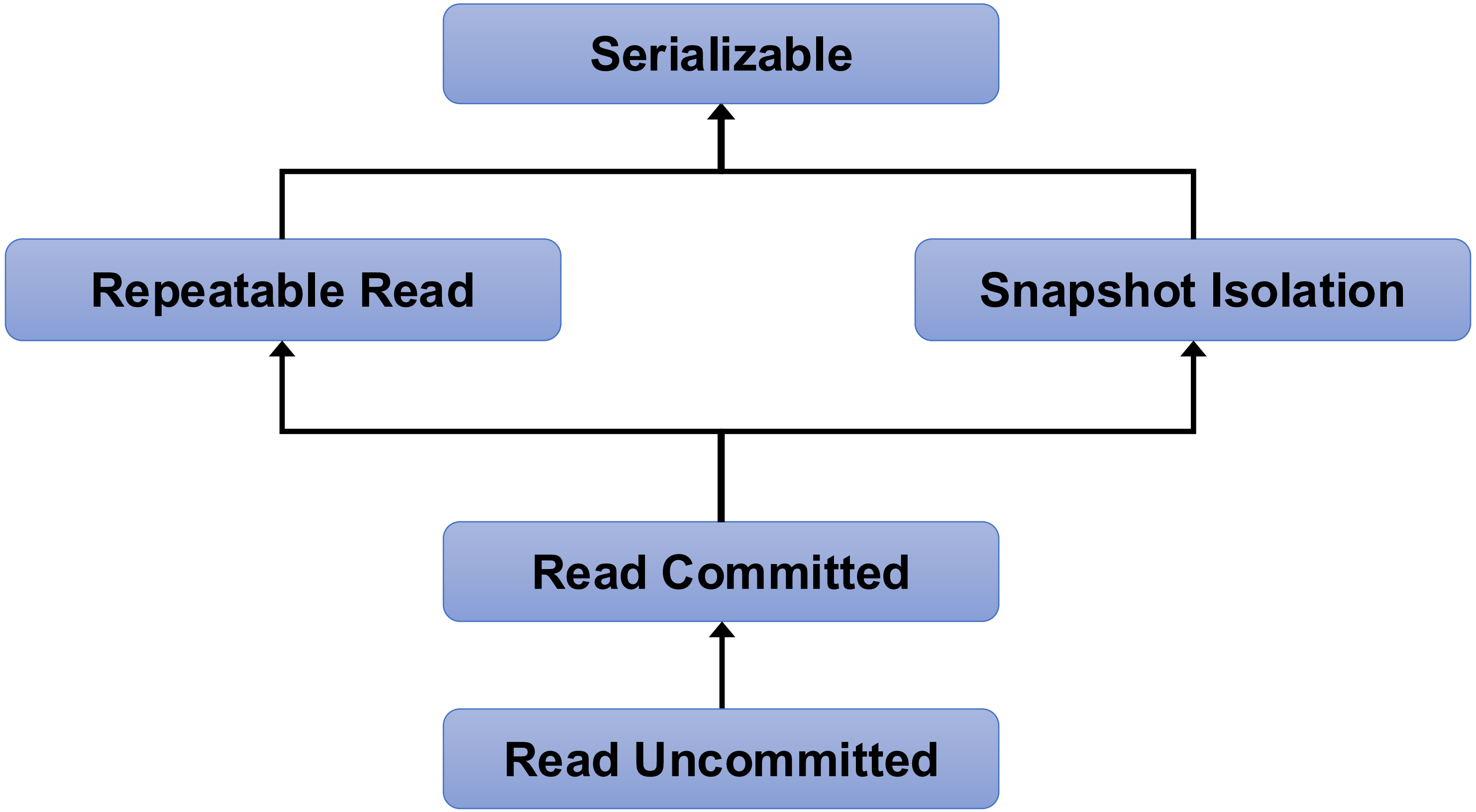
Transaction 1	Transaction 2
/* Query 1 */ SELECT * FROM users WHERE age BETWEEN 10 AND 30;	
	/* Query 2 */ INSERT INTO users(id,name,age) VALUES (3, 'Bob', 27); COMMIT;
/* Query 1 */ SELECT * FROM users WHERE age BETWEEN 10 AND 30;	

Write Skew Anomaly

- Two transactions concurrently read an **overlapping** data set, concurrently make **disjoint** updates, and finally concurrently commit, neither having seen the update performed by the other.

Transaction 1	Transaction 2
/* Keep the sum of A and B balances > 0 */ SELECT * FROM balances; /* A's balance is 100, and B's balance is 100 */	
	/* Keep the sum of A and B balances > 0*/ SELECT * FROM balances; /* A's balance is 100, and B's balance is 100 */
UPDATE balance SET balance = -50 WHERE id = A;	
	UPDATE balance SET balance = -50 WHERE id = B;
COMMIT;	
	COMMIT;

Isolation Level



Isolation Levels

Isolation Level	Dirty Read	Non-repeatable Read	Phantom Read	Write Skew
Read Uncommitted	✸	✸	✸	✸
Read Committed		✸	✸	✸
Repeatable Read			✸	Lock: NO MVCC: YES
Snapshot Isolation				✸
Serializable				

Transaction Definition in SQL

- In SQL, a transaction begins **implicitly**.
- A transaction in SQL ends by:
  - **Commit work** / **Rollback work**
- SQL statement commits implicitly by default
  - Implicit commit can be turned off by a database directive
    - JDBC -- connection.setAutoCommit(false);
- Isolation level can be set at (database / session / transaction) level
  - **set transaction isolation level serializable**
  - connection.setTransactionIsolation(Connection.TRANSACTION_SERIALIZABLE)

Implementation

- **Locking**
  - Lock on whole database vs lock on items
  - How long to hold lock?
  - Shared vs exclusive locks
- **Timestamps**
  - Transaction timestamp assigned e.g. when a transaction begins
  - Timestamps are used to detect out of order accesses
- **Multiple versions of each data item**
  - Allow transactions to read from a “snapshot” of the database

Statements in OpenGauss

START TRANSACTION [{ ISOLATION LEVEL { READ COMMITTED | SERIALIZABLE | REPEATABLE READ }
| {READ WRITE | READ ONLY } } [, ...]];

BEGIN [WORK | TRANSACTION] [{ ISOLATION LEVEL { READ COMMITTED | SERIALIZABLE | REPEATABLE READ }
| { READ WRITE | READ ONLY } } [, ...]];

{ **COMMIT** | END } [WORK | TRANSACTION] ;

ROLLBACK [WORK | TRANSACTION];

{ SET [**LOCAL**] TRANSACTION | SET **SESSION** CHARACTERISTICS AS TRANSACTION }
{ ISOLATION LEVEL { READ COMMITTED | SERIALIZABLE | REPEATABLE READ } |
{ READ WRITE | READ ONLY } } [, ...]