😄

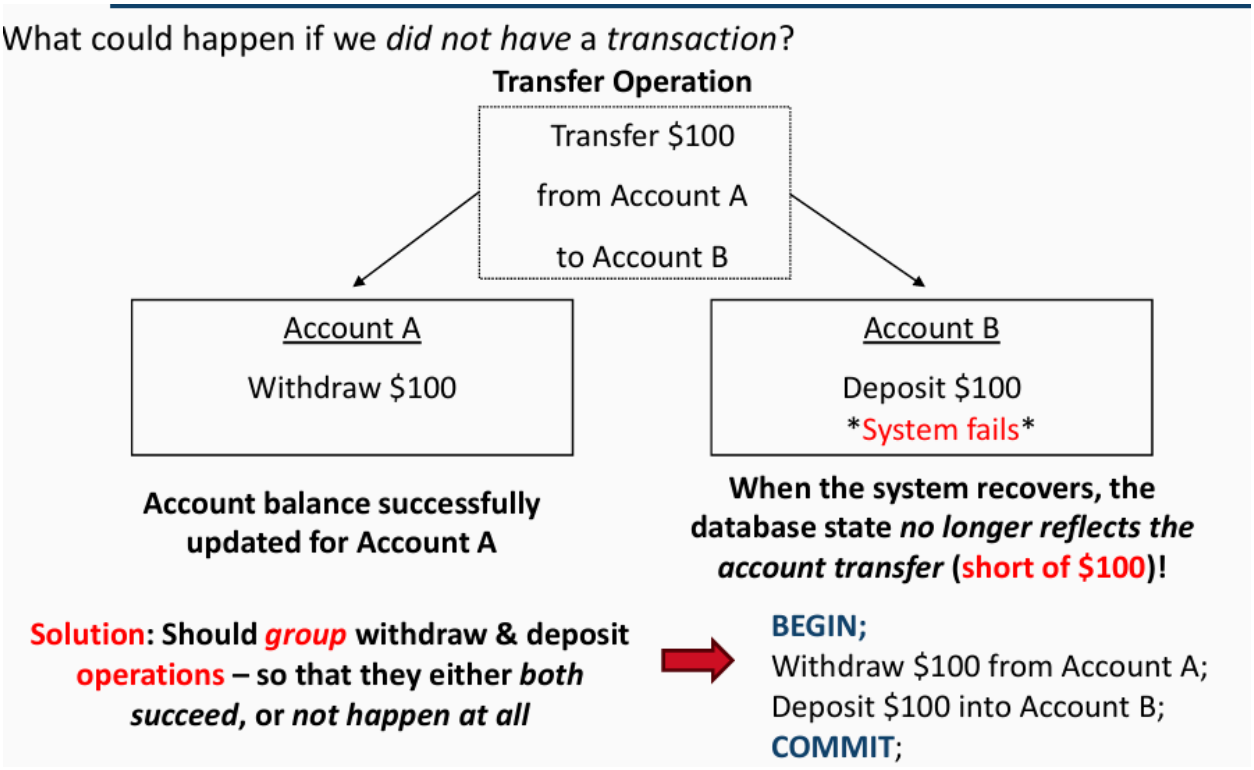
8.Transaction

8.1 ACID

- Atomicity : transaction is performed entirely or not performed at all.
- Consistency：A correct execution of a transaction must take a database from one consistent state to another.
- Isolation：Effect of multiple transaction is the same as the transaction running one after another(并发 concurrent)
- Durability：Once a transaction changes the database and the changes are committed, these changes should never be lost.

8.2 Transaction

8.2.1 Example of no Transaction



8.2.2 Definition

A program consisting one or more SQL statements.

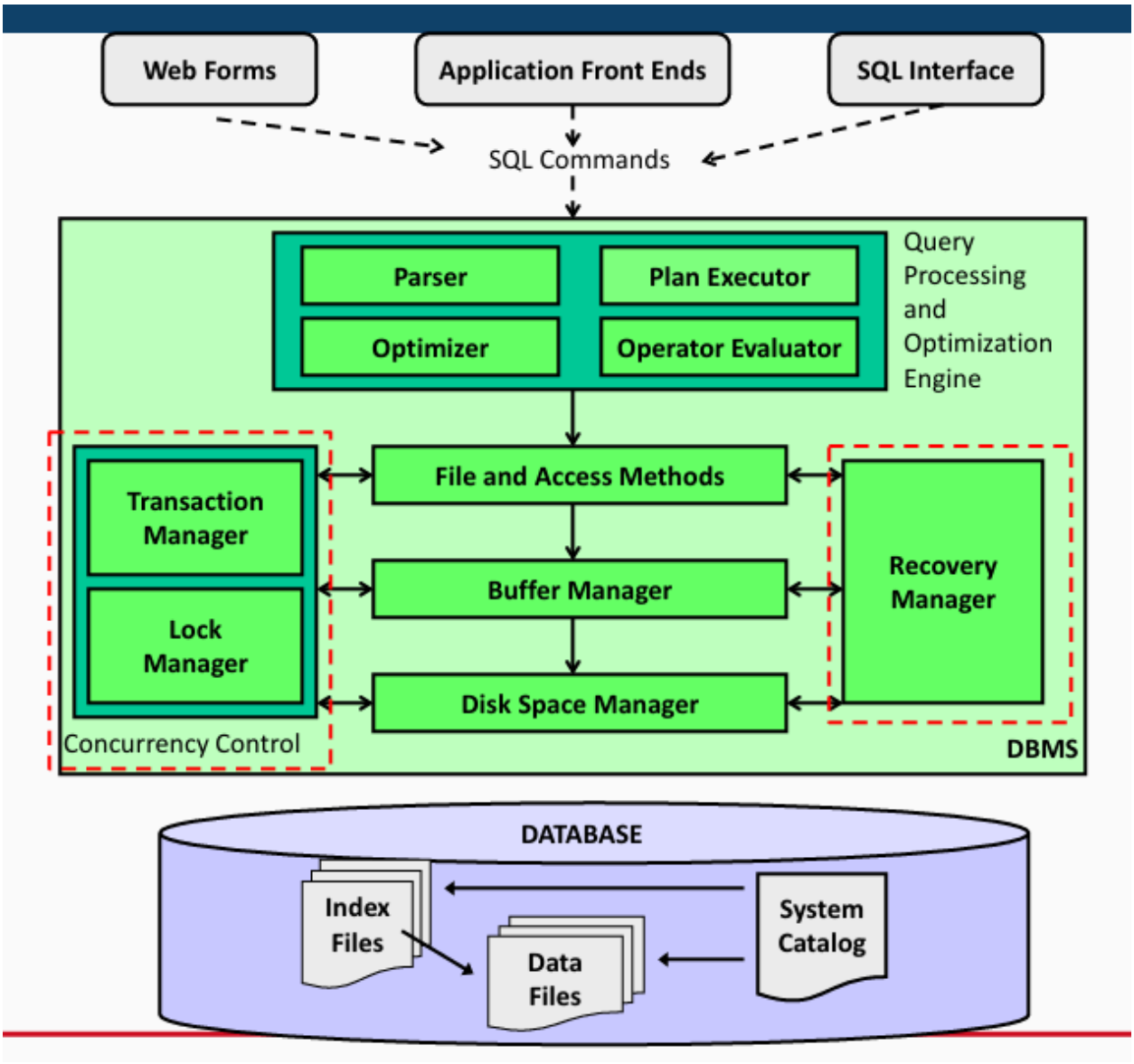
- Executed as an atomic unit
  - atomicity means executes fully or not at all

Another way to describe a transaction.

- A program that is executed to change the database state in a correct way.

A collection of one or more operations which reflect a discrete unit of work.

8.2.3 Internal Structure of a DBMS



8.2.4 Transaction consistency

- Before transaction
  - Assuming the database is in consistent state (satisfy all constraints).
- After transaction
  - All database constraints are satisfied.
  - The new database state satisfies the specification of the transaction.

Consistency means any transaction can only modify data in allowed way, which must agree with all defined rules, constraints and triggers

- Example of a *bad transaction* for a bank transfer:
  - Withdraw $100 from account A, but only deposit $90 into account B.

We can select when to enforce the database consistency in the transaction.

Example: We may *defer* the enforcement of integrity constraints.

```
CREATE TABLE UnitOfStudy (
  uos_code          VARCHAR(8),
  title             VARCHAR(20),
  lecturer_id       INTEGER,
  credit_points     INTEGER,
  CONSTRAINT UnitOfStudy_PK PRIMARY KEY (uos_code),
  CONSTRAINT UnitOfStudy_FK FOREIGN KEY (lecturer_id)
    REFERENCES Lecturer DEFERRABLE INITIALLY IMMEDIATE
);
BEGIN;
  SET CONSTRAINTS UnitOfStudy_FK DEFERRED;
  INSERT INTO UnitOfStudy VALUES('INFO1000', 'Graphics', 3, 6);
  INSERT INTO Lecturer VALUES(3, 'Steve', CSE);
  SET CONSTRAINTS UnitOfStudy_FK IMMEDIATE ;
COMMIT;
```

does not exist yet!

UnitOfStudy

uos_code	title	lecturer_id	credit_points
COMP9120	DBMS	1	6
COMP9007	Algorithm	2	6

Lecturer

Lecturer_id	name	department
1	Adam	CSE
2	Lily	IT

8.2.5 Transaction atomic

Logs all actions that would need to be undone if the transaction is aborted (incomplete).

- all actions undone
- do a redo

- If the transaction successfully completes, it is said to have committed.
- else, it is said to have been aborted

8.2.6 Durable

Once the transaction is committed, its effects should persist in a database.

- Transaction logs
  - use stable storage as a log to store a history of modifications made to the database.
  - Recovery Manager should take responsible to the transaction logs.
  - Every transaction has a log associated with it.
  - If a transaction aborts, depending on the recovery protocol, use the log to undo(rollback)/redo(copy the log value of an item from stable storage to disk) the transaction.

8.2.7 Isolation

Transaction can run concurrently, their operations can be interleaved（交错）.

The interleaving is usually decided by host operating system.

If no intervention from transaction manager, it may leave incorrect state.

So the `Control concurrent` is `correctness` but also `efficiency` .

- 3 types of problems
  - Lost update : concurrent update make the former update lost.

    T1 : READ X

    T2 : READ X

    T1 : WRITE X = X + 20

T2 : WRITE X = X + 30

When concurrent happen, return ONLY X + 30

- Temporary update : update an item an then fails, another read the wrong temporary value

  T1: Update X to 200
  T2: Read X = 200 ← 读取了临时、不合法的值
  T1: Rollback ← X 实际上还是 100

- Incorrect summary : when a transaction is updating an aggregate of items, another is reading or updating the value

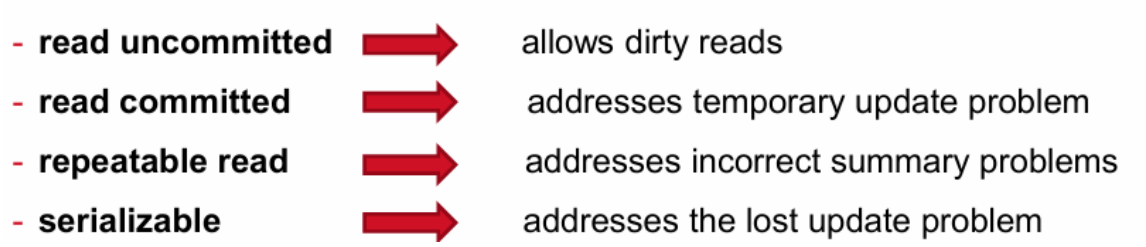
  X = 100 Y = 50

  T1: Read X = 100
  T1: Write X = 50
  T2: Read X = 50 ← 新值
  T2: Read Y = 50 ← 旧值

  T1 : WRITE Y = X + Y

  T2: WRITE SUM = X + Y = 100
  → Total = 100 ← 实际应该是 X=50, Y=100 → Total=150

- 4 isolation levels in SQL standard
  - read uncommitted : dirty read — A transaction can read data written by a concurrent uncommitted transaction.
  - read committed : no dirty read → Lowest level of PostgreSQL
  - repeatable read : In a transaction, it can only see 1 version of data. (which is data committed before transaction began)
  - serializable : highest level of isolation, concurrent executing = some serial execution of these same transactions.

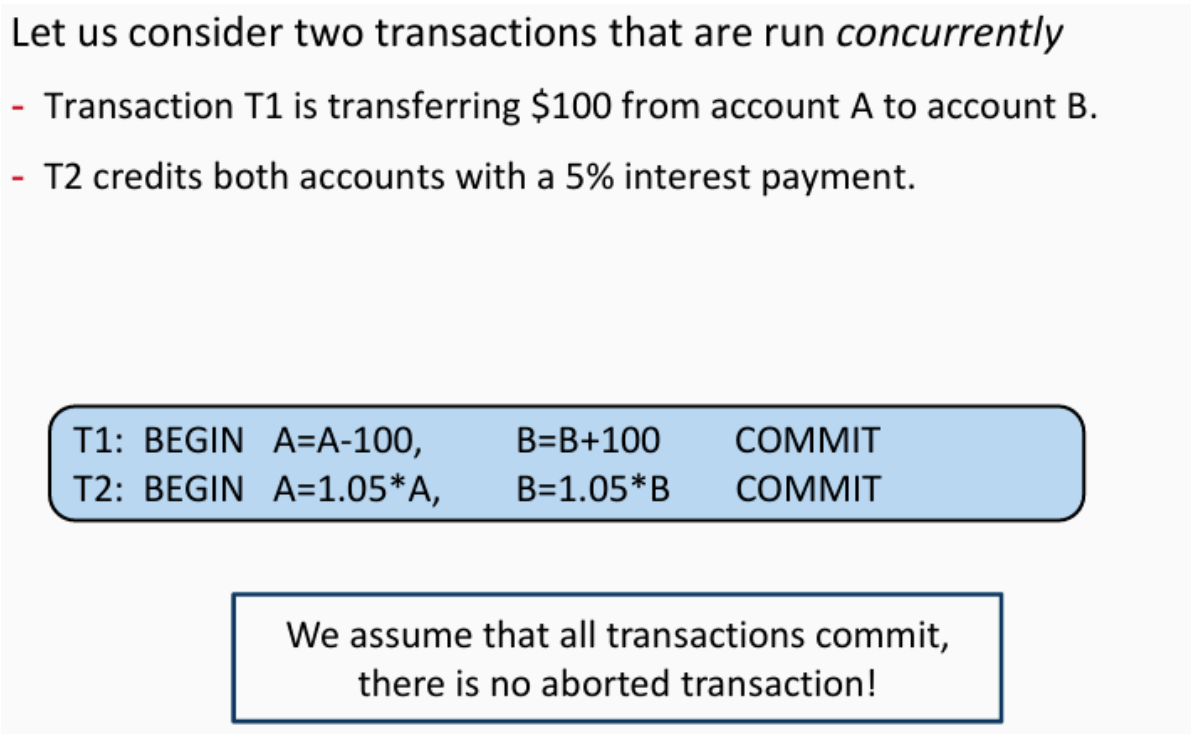



```
BEGIN;
SET TRANSACTION ISOLATION LEVEL

{ SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ UNCOMM
-- 下面执行的所有语句都在该隔离级别下运行
SELECT * FROM Account WHERE balance > 1000;

COMMIT;
```

- Transactions should be isolated



- Serial execution:

  A = A - 100 B = B+ 100

  THEN

  A = A * 1.05 B = 1.05 *B

- Else

  A = A * 1.05 B = B * 1.05

  THEN

  A = A - 100 B = B + 100

- Serial Schedule

  A schedule in which all transactions are executed from start to finish, without any interleaving.

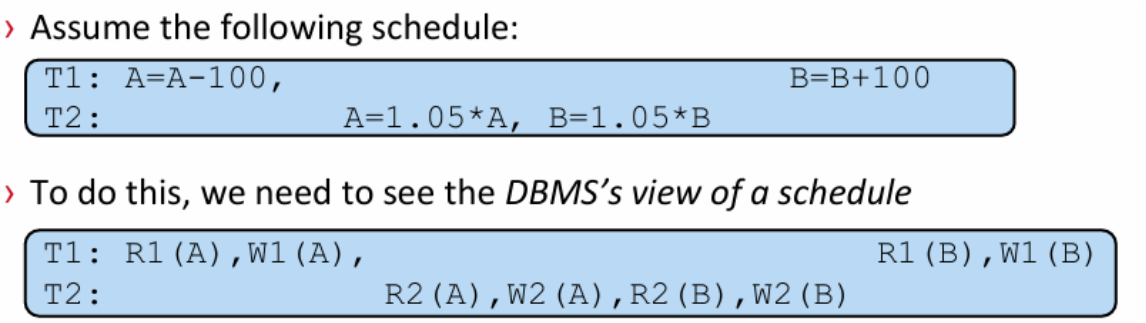
  - However, Interleaving improves the performance.

  Serial makes sure that the result of interleaving = running steps by steps

  - T1 T2 interleaving = T1 then T2

  Serializability even though is good, but it is expensive to check.

  - How to check



  R : READ

  W : WRITE

  - Then Conflict serializability

A schedule is conflict serializable if it is conflict equivalent to a serial schedule.

- Conflicts:
  - read different item in any order

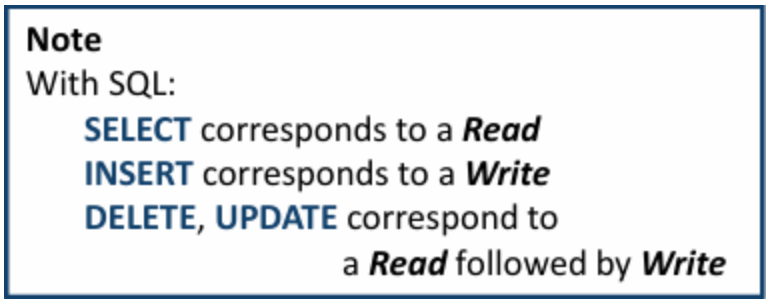
    T1 : READ(A) T2: READ(B)
  - read same item in any order

    T1: READ(A) T2: READ(A)
  - R/W different data items in any order

    T1 : READ(A) T2: WRITE(B)
  - reading / writing the same data item
    - A read of a transaction T1 , then T2 write it not semantically. Conflict

      T1 : READ(A) T2 : WRITE(A)
    - The order of 2 writes of 2 transactions does matter. Last value will depend on which write comes last. Conflict

      T1 : WRITE(A) T2 : WRITE(A)



A schedule is conflict serializable if it is conflict equivalent to some serial schedule.

- 2 schedule is conflict equivalent if
  - They involve the same set of operations of the same transactions.
  - They order every pair of conflicting operations the same way.

- How to check conflict serializability
  - Non-conflicting swapping

    Swap the non-conflicting operation until it becomes T1 → T2

    Swapping the operation that has non-conflict operation, the resulting schedule is serial (T1, T2), This means the above schedule is conflict-serializable.

  - Another way to test conflict

    Precedence Graph (Serialization Graph Testing or SGT)

    The edge corresponds to one of the following case :

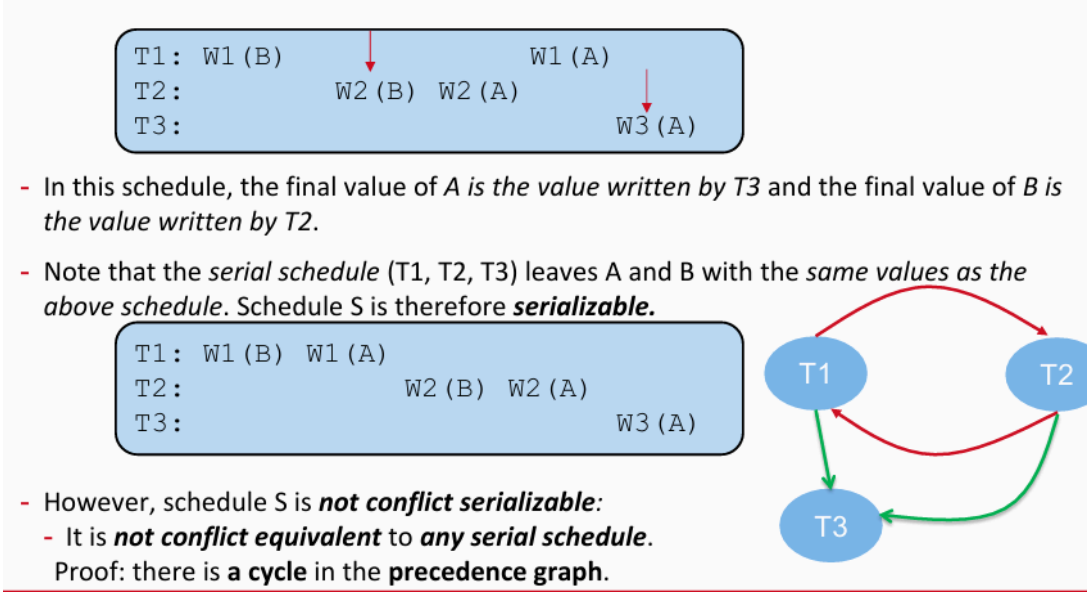
    - T1 executes W before T2 execute R
    - T1 executes R before T2 W
    - T1 W before T2 W
    - No read & read , have edge

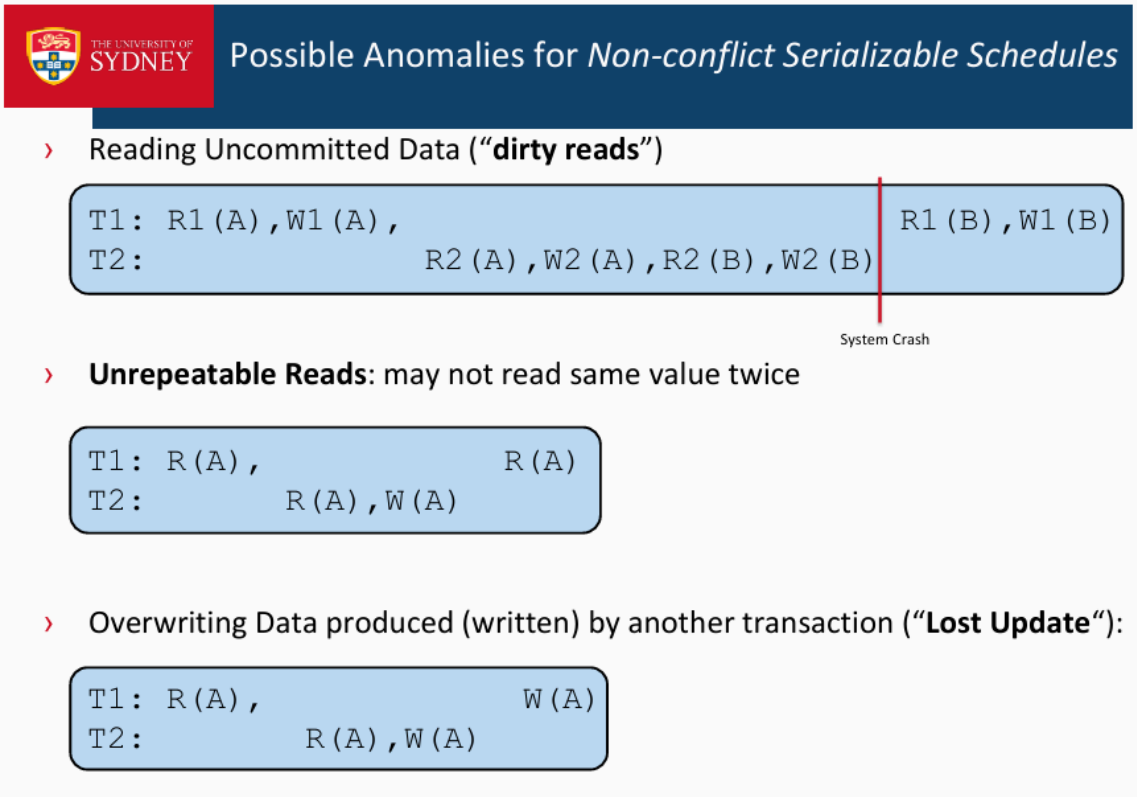
    Check for cycles, if there is any cycle, then the schedule is not conflict serializable.

    If the SGT graph is acyclic then do topological sorting. This would determine a linear order consistent to the partial order.

    - But it is expensive!

- Conflict Serializability & Serializability
  - CS : The given order is identical to a T1 → T2 (serial) through swapping the non-conflict operation
  - S : The result of the given order sequence is identical some of the serial
  - CS → S but S cannot → CS





多个事务（Transaction）
↓
构成一个调度（Schedule）
↓
是否正确？→ 判断是否 Serializable
↓
用 Conflict Serializability 来判断
↓
看冲突对 → 画前驱图 → 看是否有环

8.2.8 Lock-based Concurrency Control

A pessimistic protocols : Prevent the conflict happening

- Lock manager maintain a lock table.
- Read locks : "Shared" lock (S)

- Write locks : “Exclusive” lock (X)

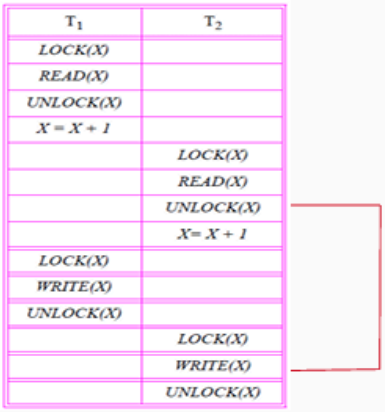
Held by T1 / T2 Requests	Shared	Exclusive
Shared	OK	T2 wait on T1
Exclusive	T2 wait on T1	T2 wait on T1

If T1 got a Shared lock, T2 can apply for the shared lock, but applying the Exclusive lock should be blocked.

If T1 got a exclusive lock, T2 cannot apply for any lock.

- The time of release a lock
  - Do it as soon as we have used that item, but may leave database in an in consistent state.

Example: Consider an airline reservation system where two airline agents are trying to make a booking on the same flight. A schedule could look like this:



T_1	T_2
LOCK(X)	
READ(X)	
UNLOCK(X)	
X = X + 1	
	LOCK(X)
	READ(X)
	UNLOCK(X)
	X= X + 1
LOCK(X)	
WRITE(X)	
UNLOCK(X)	
	LOCK(X)
	WRITE(X)
	UNLOCK(X)

This schedule would result in making one single reservation instead of 2!

- Two-Phase Locking

  For every transaction

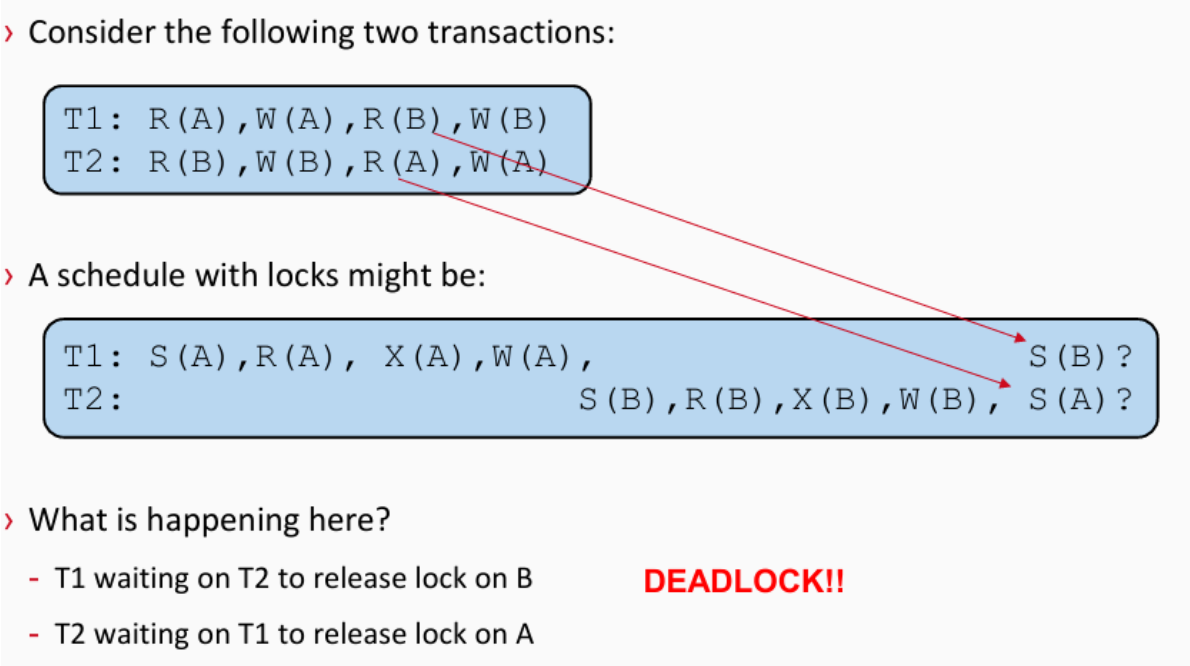
  - obtain locks
  - perform computations
  - release locks and commit

  First grant all the locks, release them finally.

  - Growing phase : the number of locks can only increase.
  - Shrinking phase : the number of locks can only decrease.
  - Commit the change to the database

  Strict 2PL : all locks are released after commit.

- Deadlock



T1 is waiting for the release of T2 while T2 is waiting the release of T1 !

- The time of transaction happening
  - T1 holds the lock on A and is requesting a lock on an item B

        - T2 holds the lock on item B and requesting a lock on A.

    2 way to deal with deadlock :

    - Deadlock prevention
        - Static 2-phase locking :

          All the transaction will get all the locks or none.
        - Deadlock detection

          Use deadlock detection algorithms to detect and aborted the waiting transaction.

    each row is an item!

8.2.8 Writing a transaction in SQL

3 key relevant SQL commands

- BEGIN
- COMMIT
- ROLLBACK / ABORT
- also can SET AUTOCOMMIT OFF or SET AUTOCOMMIT ON in pgadmin client
    - auto-commit on each statement is its own transaction

      ```
      BEGIN;
      <你的 SQL 命令>;
      COMMIT;
      ```
    - auto-commit off, you should write begin-commit on you own.