

Introduction to Data Management

CSE 344

Lecture 22: Transactions I

Announcements

- HW6 due tomorrow night
- Next webquiz and hw out by end of the week
- HW7:
 - Some Java programming required
 - Plus connection to SQL Azure
 - Demos, overview in sections tomorrow
- Office hour moved tomorrow to 006 lab

Outline

- Serial and Serializable Schedules (18.1)
- Conflict Serializability (18.2)
- Locks (18.3) [Start today and finish next time]

Review: Transactions

- **Problem:** An application must perform *several* writes and reads to the database, as a unit
- **Solution:** multiple actions of the application are bundled into one unit called a *Transaction*
- Turing awards to database researchers
 - Charles Bachman 1973 for CODASYL
 - Edgar Codd 1981 for relational databases
 - **Jim Gray 1998 for transactions**

Review: TXNs in SQL

BEGIN TRANSACTION
[SQL statements]
COMMIT or
ROLLBACK (=ABORT)

[single SQL statement]

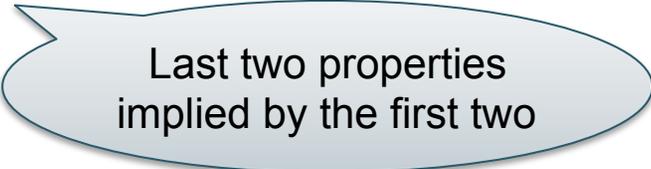
If BEGIN... missing,
then TXN consists
of a single instruction

Review: ACID

- **Atomic**
 - State shows either all the effects of txn, or none of them
- **Consistent**
 - Txn moves from a state where integrity holds, to another where integrity holds
- **Isolated**
 - Effect of txns is the same as txns running one after another (i.e., looks like batch mode)
- **Durable**
 - Once a txn has committed, its effects remain in the database

Implementing ACID Properties

- **Isolation:**
 - Achieved by the concurrency control manager (or scheduler)
 - Discussed briefly in 344 today and in the next lecture
 - Discussed more extensively in 444
- **Atomicity**
 - Achieved using a log and a recovery manager
 - Discussed in 444
- **Durability**
 - Implicitly achieved by writing back to disk
- **Consistency**
 - Implicitly guaranteed by A and I



Last two properties
implied by the first two

Isolation: The Problem

- Multiple transactions are running concurrently
 T_1, T_2, \dots
- They read/write some common elements
 A_1, A_2, \dots
- How can we prevent unwanted interference ?
- The SCHEDULER is responsible for that

Schedules

A *schedule* is a sequence of interleaved actions from all transactions

Serial Schedule

- A serial schedule is one in which transactions are executed one after the other, in some sequential order
- Fact: nothing can go wrong if the system executes transactions serially
 - But database systems don't do that because we need better performance

Example

A and B are elements
in the database
t and s are variables
in txn source code

T1	T2
READ(A, t)	READ(A, s)
t := t+100	s := s*2
WRITE(A, t)	WRITE(A,s)
READ(B, t)	READ(B,s)
t := t+100	s := s*2
WRITE(B,t)	WRITE(B,s)

A Serial Schedule

T1

T2

READ(A, t)

t := t+100

WRITE(A, t)

READ(B, t)

t := t+100

WRITE(B,t)

READ(A,s)

s := s*2

WRITE(A,s)

READ(B,s)

s := s*2

WRITE(B,s)

Serializable Schedule

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

A Serializable Schedule

T1

READ(A, t)
t := t+100
WRITE(A, t)

READ(B, t)
t := t+100
WRITE(B,t)

T2

READ(A,s)
s := s*2
WRITE(A,s)

READ(B,s)
s := s*2
WRITE(B,s)

This is a **serializable** schedule.
This is NOT a serial schedule

A Non-Serializable Schedule

T1	T2
READ(A, t)	
t := t+100	
WRITE(A, t)	
	READ(A,s)
	s := s*2
	WRITE(A,s)
	READ(B,s)
	s := s*2
	WRITE(B,s)
READ(B, t)	
t := t+100	
WRITE(B,t)	

How do We Know if a Schedule is Serializable?

Notation

$T_1: r_1(A); w_1(A); r_1(B); w_1(B)$
 $T_2: r_2(A); w_2(A); r_2(B); w_2(B)$

Key Idea: Focus on *conflicting* operations

Conflicts

- Write-Read – WR
- Read-Write – RW
- Write-Write – WW

Conflict Serializability

Conflicts:

Two actions by same transaction T_i :

$r_i(X); w_i(Y)$

Two writes by T_i, T_j to same element

$w_i(X); w_j(X)$

Read/write by T_i, T_j to same element

$w_i(X); r_j(X)$

$r_i(X); w_j(X)$

Conflict Serializability

- A schedule is *conflict serializable* if it can be transformed into a serial schedule by a series of swappings of adjacent non-conflicting actions
- Every conflict-serializable schedule is serializable
- A serializable schedule may not necessarily be conflict-serializable

Conflict Serializability

Example:

$r_1(A); w_1(A); r_2(A); w_2(A); r_1(B); w_1(B); r_2(B); w_2(B)$

Conflict Serializability

Example:

$r_1(A); w_1(A); r_2(A); w_2(A); r_1(B); w_1(B); r_2(B); w_2(B)$



$r_1(A); w_1(A); r_1(B); w_1(B); r_2(A); w_2(A); r_2(B); w_2(B)$

Conflict Serializability

Example:

$r_1(A); w_1(A); r_2(A); w_2(A); r_1(B); w_1(B); r_2(B); w_2(B)$



$r_1(A); w_1(A); r_1(B); w_1(B); r_2(A); w_2(A); r_2(B); w_2(B)$

Conflict Serializability

Example:

$r_1(A); w_1(A); r_2(A); w_2(A); r_1(B); w_1(B); r_2(B); w_2(B)$



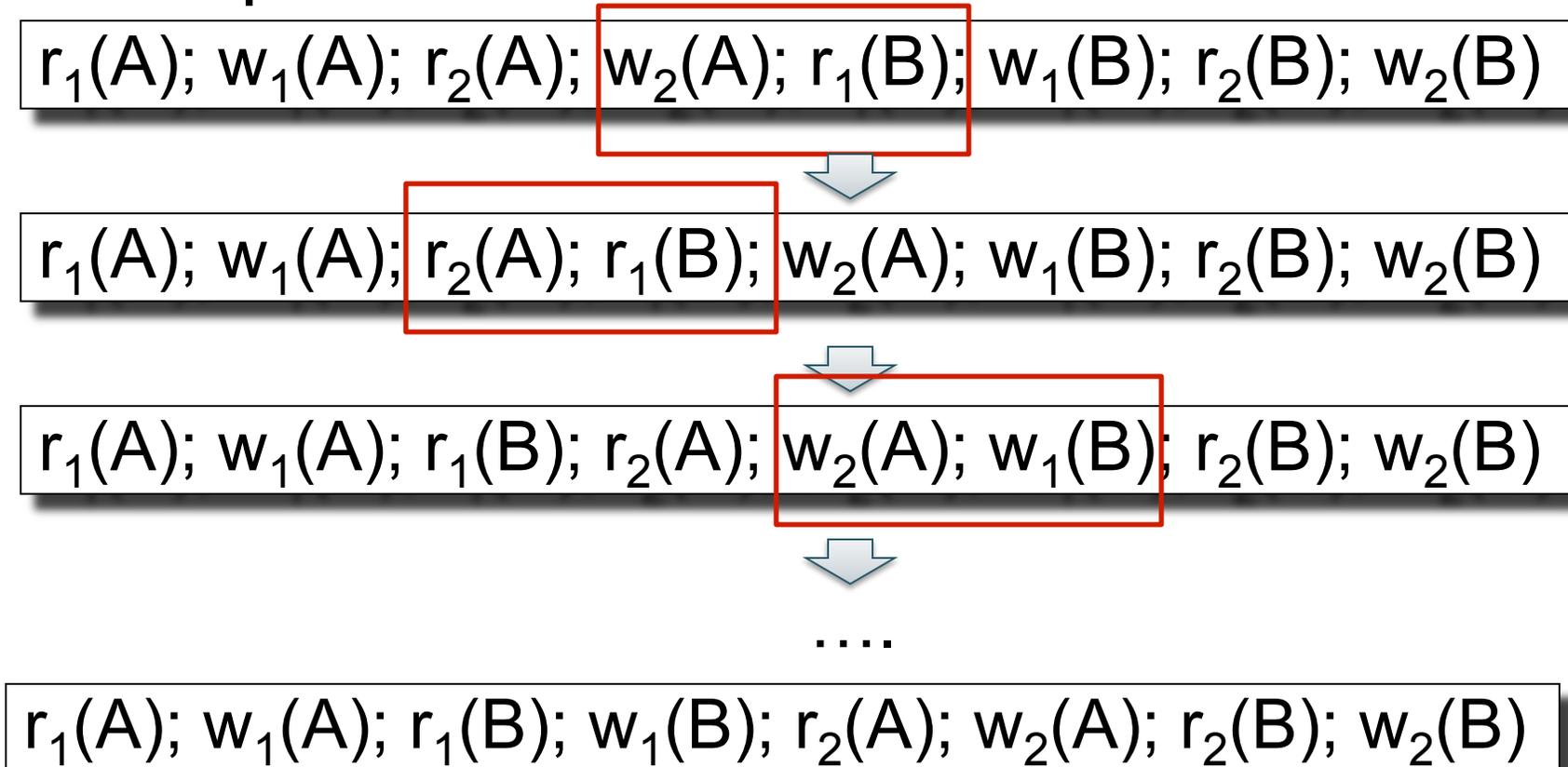
$r_1(A); w_1(A); r_2(A); r_1(B); w_2(A); w_1(B); r_2(B); w_2(B)$



$r_1(A); w_1(A); r_1(B); w_1(B); r_2(A); w_2(A); r_2(B); w_2(B)$

Conflict Serializability

Example:



Testing for Conflict-Serializability

Precedence graph:

- A node for each transaction T_i ,
- An edge from T_i to T_j whenever an action in T_i conflicts with, and comes before an action in T_j
- The schedule is serializable iff the precedence graph is acyclic

Example 1

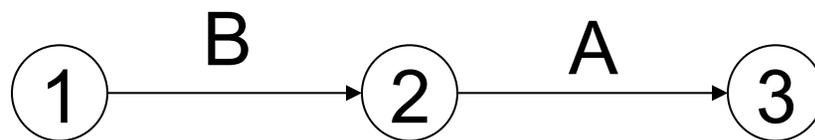
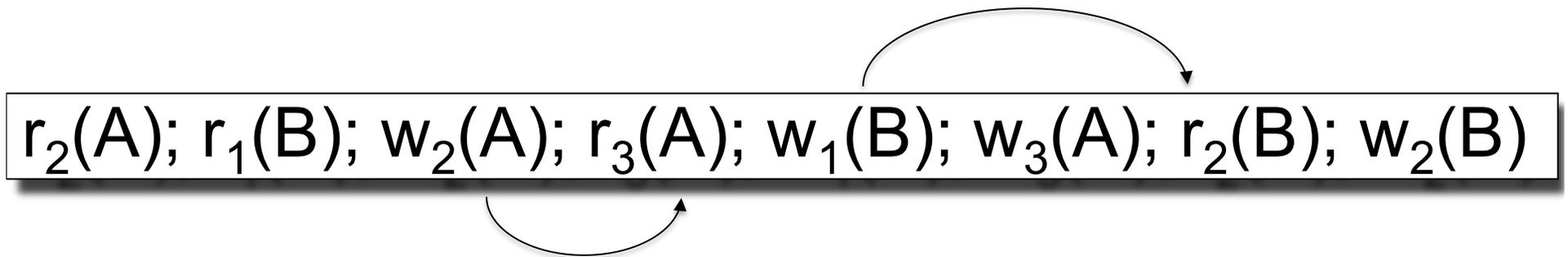
$r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B)$

①

②

③

Example 1



This schedule is **conflict-serializable**

Example 2

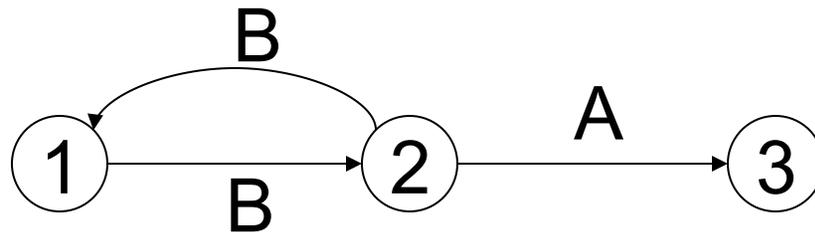
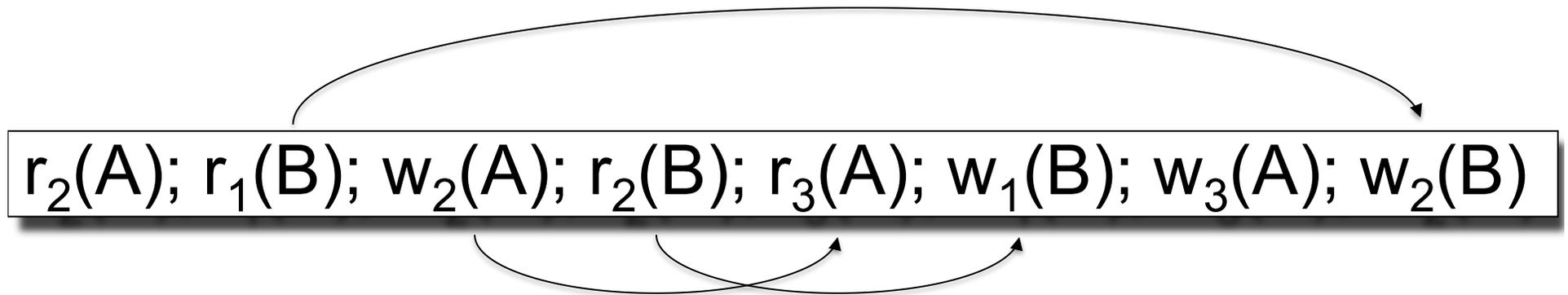
$r_2(A); r_1(B); w_2(A); r_2(B); r_3(A); w_1(B); w_3(A); w_2(B)$

①

②

③

Example 2



This schedule **is NOT conflict-serializable**

Scheduler

- **Scheduler** = is the module that schedules the transaction's actions, ensuring serializability
- Also called **Concurrency Control Manager**
- We discuss next how a scheduler may be implemented

Implementing a Scheduler

Major differences between database vendors

- **Locking Scheduler**
 - Aka “pessimistic concurrency control”
 - SQLite, SQL Server, DB2
- **Multiversion Concurrency Control (MVCC)**
 - Aka “optimistic concurrency control”
 - Postgres, Oracle

We discuss only locking in 344

Locking Scheduler

Simple idea:

- Each element has a unique **lock**
- Each transaction must first **acquire** the lock before reading/writing that element
- If the lock is taken by another transaction, then wait
- The transaction must **release** the lock(s)

By using locks scheduler ensures conflict-serializability

What Data Elements are Locked?

Major differences between vendors:

- Lock on the entire database
 - SQLite
- Lock on individual records
 - SQL Server, DB2, etc

Let's Study SQLite First

- SQLite is very simple
- More info: <http://www.sqlite.org/atomiccommit.html>
- Lock types
 - READ LOCK (to read)
 - RESERVED LOCK (to write)
 - PENDING LOCK (wants to commit)
 - EXCLUSIVE LOCK (to commit)

SQLite

Step 1: when a transaction begins

- Acquire a **READ LOCK** (aka "SHARED" lock)
- All these transactions may read happily
- They all read data from the database file
- If the transaction commits without writing anything, then it simply releases the lock

SQLite

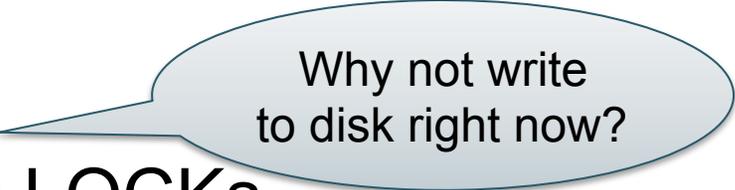
Step 2: when one transaction wants to write

- Acquire a **RESERVED LOCK**
- May coexists with many READ LOCKs
- Writer TXN may write; these updates are only in main memory; others don't see the updates
- Reader TXN continue to read from the file
- New readers accepted
- No other TXN is allowed a RESERVED LOCK

SQLite

Step 3: when writer transaction wants to commit, it needs *exclusive lock*, which can't coexists with *read locks*

- Acquire a **PENDING LOCK**
- May coexists with old READ LOCKs
- No new READ LOCKS are accepted
- Wait for all read locks to be released



Why not write to disk right now?

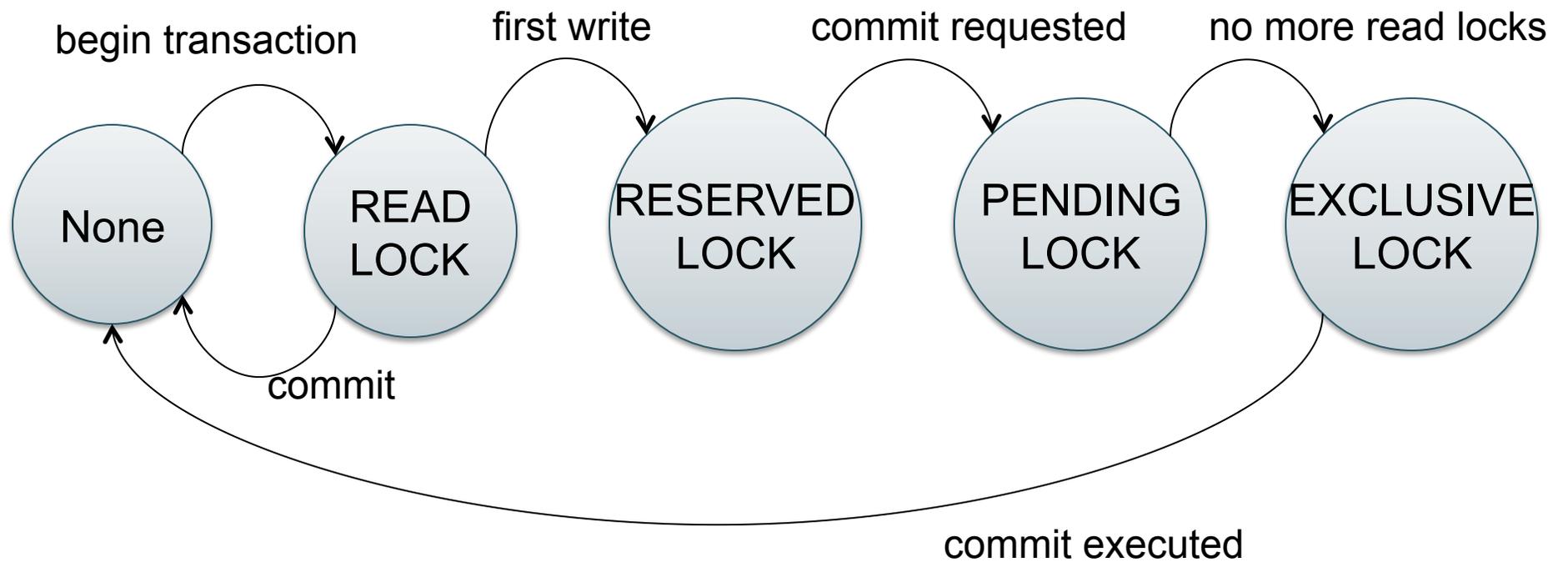
SQLite

Step 4: when all read locks have been released

- Acquire the **EXCLUSIVE LOCK**
- Nobody can touch the database now
- All updates are written permanently to the database file

- Release the lock and **COMMIT**

SQLite



SQLite Demo

```
create table r(a int, b int);  
insert into r values (1,10);  
insert into r values (2,20);  
insert into r values (3,30);
```

Demonstrating Locking in SQLite

T1:

```
begin transaction;  
select * from r;  
-- T1 has a READ LOCK
```

T2:

```
begin transaction;  
select * from r;  
-- T2 has a READ LOCK
```

Demonstrating Locking in SQLite

T1:

```
update r set b=11 where a=1;  
-- T1 has a RESERVED LOCK
```

T2:

```
update r set b=21 where a=2;  
-- T2 asked for a RESERVED LOCK: DENIED
```

Demonstrating Locking in SQLite

T3:

```
begin transaction;
```

```
select * from r;
```

```
commit;
```

```
-- everything works fine, could obtain READ LOCK
```

Demonstrating Locking in SQLite

T1:

commit;

-- SQL error: database is locked

-- T1 asked for PENDING LOCK -- GRANTED

-- T1 asked for EXCLUSIVE LOCK -- DENIED

Demonstrating Locking in SQLite

T3':

```
begin transaction;
```

```
select * from r;
```

```
-- T3 asked for READ LOCK-- DENIED (due to T1)
```

T2:

```
commit;
```

```
-- releases the last READ LOCK; T1 can commit
```

Appendix / Review: Some Famous Anomalies

- What could go wrong if we didn't have concurrency control:
 - Dirty reads (including inconsistent reads)
 - Unrepeatable reads
 - Lost updates

Many other things can go wrong too

Dirty Reads

Write-Read Conflict

T_1 : WRITE(A)

T_1 : ABORT

T_2 : READ(A)

Inconsistent Read

Write-Read Conflict

```
T1: A := 20; B := 20;  
T1: WRITE(A)
```

```
T1: WRITE(B)
```

```
T2: READ(A);  
T2: READ(B);
```

Unrepeatable Read

Read-Write Conflict

T_1 : WRITE(A)

T_2 : READ(A);

T_2 : READ(A);

Lost Update

Write-Write Conflict

T_1 : READ(A)

T_1 : $A := A + 5$

T_1 : WRITE(A)

T_2 : READ(A);

T_2 : $A := A * 1.3$

T_2 : WRITE(A);