TIMESTAMP-BASED CONCURRENCY CONTROL

Pessimistic Timestamp-Based Concurrency Control

- Transactions get timestamps based on when they start running
- Database records have timestamps for reads, updates
- Abort transaction if:
  - Read future values (written by “future transactions”)
  - Rewrite the past (written by “past transactions”)

Pessimistic Timestamp-Based Concurrency Control

- Each transaction assigned a timestamp $TS(T_i)$ when it starts
- If $TS(T_i) < TS(T_j)$ then T_i is serialized to run before $TS(T_j)$
- Each data item has 2 timestamps:
  - W-timestamp(A): timestamp of transaction that did last write
  - R-timestamp(A): timestamp of transaction that did last read

50

50

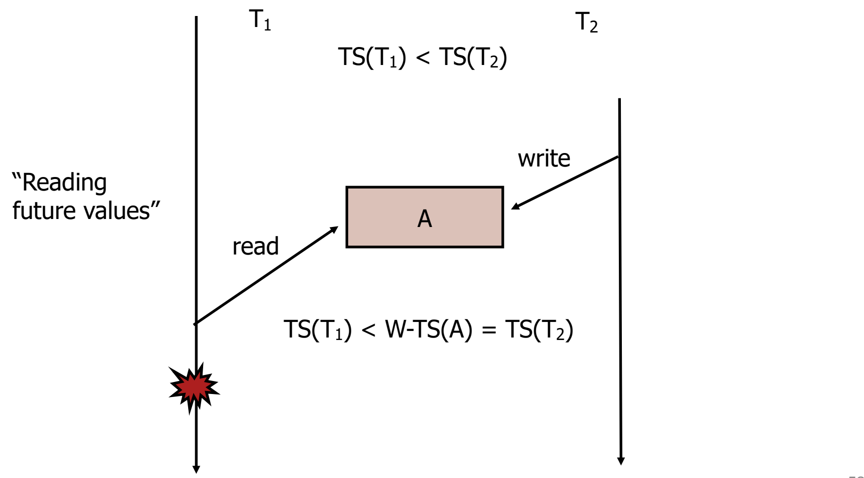
Pessimistic Timestamp-Based Concurrency Control

- Suppose transaction T_i executes "**read A**":
  - if $TS(T_i) <$ W-timestamp(A)
    then read fails and T_i is aborted/restarted
  - if $TS(T_i) >$ W-timestamp(A)
    then read succeeds;
    R-timestamp(A) := max(R-timestamp(A), $TS(T_i)$)

51

51

Pessimistic Timestamp-Based Concurrency Control



52

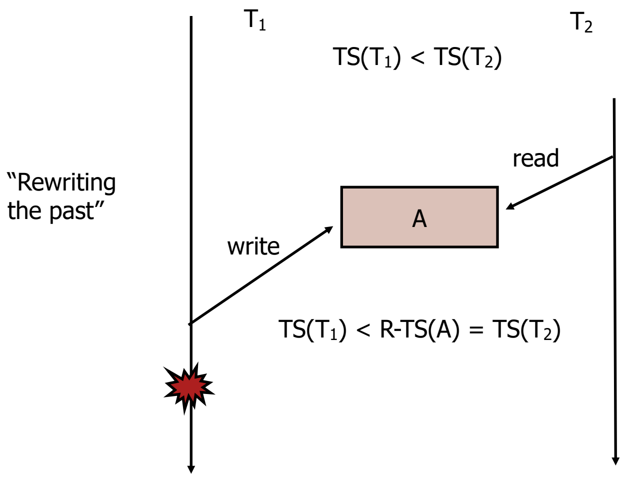
Pessimistic Timestamp-Based Concurrency Control

- Suppose transaction Ti executes "**write A**":
  - if TS(Ti) < R-timestamp(A)
    then write fails and Ti is aborted/restarted
  - if TS(Ti) < W-timestamp(A)
    then write fails and Ti is aborted/restarted
  - (Thomas' Write rule: ignore this write;
    - may produce schedules that are not CS)
  - otherwise write succeeds;
    - W-timestamp(A) := TS(Ti)

53

53

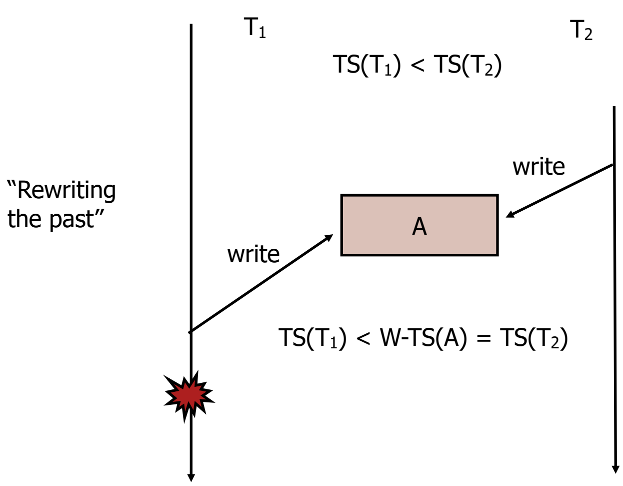
Pessimistic Timestamp-Based Concurrency Control



54

54

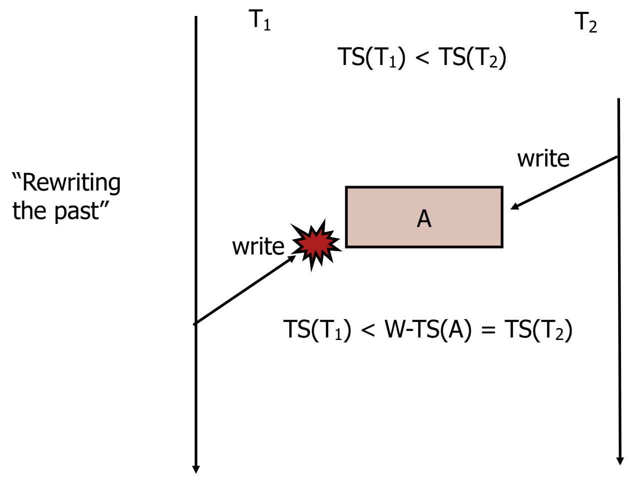
Pessimistic Timestamp-Based Concurrency Control



55

55

Pessimistic Timestamp-Based Concurrency Control



56

56

Pessimistic Timestamp-Based Concurrency Control

T_1	T_2	T_3	T_4	T_5
				rd X
	rd Y			
rd Y				
		wr Y		
		wr Z		
				rd Z
	rd Z			
	abort			
rd X				
			wr Z	
			abort	
				wr X
				wr Z

57

57

Recoverability and Cascade Freedom

- Suppose T_i reads data item A written by T_j
  - R-timestamp(A) = TS(Ti) > TS(Tj)
    = W-timestamp(A)
  - T_i must delay commit until T_j commits
  - Possibility of cascading rollbacks
- Delay all writes until end of transaction, and perform all writes as single atomic
  - Avoid cascading rollbacks

58

58

OPTIMISTIC CONCURRENCY CONTROL

59

59

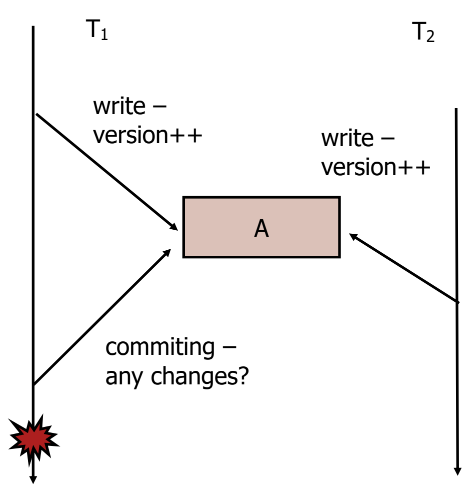
Optimistic Version Ordering

- Execute operations without synchronization
  - but keep track of timestamps
- Commit: check to see if any data items used by the transaction have been changed
  - abort if something has been changed
  - commit otherwise
- Optimistic concurrency control allows maximum parallelism, but at a price…

60

60

Optimistic Version Ordering



61

61

Optimistic Locking in Jakarta EE

- Entity: object represented by record in database
- Entity Manager (`em`): object representing database manager
  - `em.persist(entity)`: add entity to database
- Transaction (`tx`)
  - `tx.begin()`: start transaction
  - `tx.commit()`: end transaction

62

62

Optimistic Locking in Jakarta EE

- Designated version stamp on entity object
- Incremented on every database update

```
@Entity
public class FlowSheet {
  @Id @GeneratedValue
  private Long id;
  @Version
  private Integer version;
  ...
}
```

63

63

Optimistic Locking in Jakarta EE

```
FlowSheet f1;           FlowSheet f2;
tx.begin();
em.persist(f1);
tx.commit();
// Version = 1          tx.begin();
                        f2 = em.find(…);
                        … update f2 …
                        tx.commit();
                        // Version = 2
```

64

64

Optimistic Locking in Jakarta EE

```
FlowSheet f1;           FlowSheet f2;
tx.begin();
f1 = em.find(…);
… update f2 …
tx.commit();            tx.begin();
// Version = N          f2 = em.find(…);
                        … update f2 …
                        tx.commit();
                        // Version = N+1
```

65

65

Optimistic Locking in Jakarta EE

```
FlowSheet f1;              FlowSheet f2;
tx.begin();
f1 = em.find(…);           tx.begin();
… update f1 …              f2 = em.find(…);
                           … update f2 …
                           tx.commit();
tx.commit();               // Version = N
// OptimisticLock
// Exception
```

66

66

Pros and cons of approaches

- Pessimistic schemes work best when conflicts between transactions are common
- Optimistic scheme works best when conflicts are rare

67

67