MVCC: MULTIVERSION CONCURRENCY CONTROL

Multiversion Schemes

- Provide consistent version of database to transaction
  - Multiversion Timestamp Ordering
  - Multiversion Two-Phase Locking
- Use timestamps to label versions
- **Write**: Create new version of the data item
- **Read(*A*)**: Return appropriate version of *A* based on transaction timestamp
- **Read**s never have to wait!

Multiversion Timestamp Ordering

- Data item A has a sequence of versions $<A_1, A_2, ..., A_m>$
- Each version A_k contains three data fields:
  - Content: the value of version A_k.
  - W-timestamp(A_k): TS of transaction that created (wrote) version A_k
  - R-timestamp(A_k): Largest TS of transaction that read version A_k

85

85

Multiversion Timestamp Ordering

- When T_i creates a new version A_k of A
  - W-timestamp(A_k) := TS(T_i)
  - R-timestamp(A_k) := TS(T_i)
- When T_j reads A_k:
  - if TS(T_j) > R-timestamp(A_k)
    R-timestamp(A_k) := TS(T_j)

86

86

Multiversion Timestamp Ordering

- Suppose *T* issues read(*A*)
- Versions A_1,…, A_k,…, A_m ordered by W-TS(A_i)
- Pick A_k for some k such that
  - W-TS(A_k)<=TS(*T*)
  - W-TS(A_{k+1})>TS(*T*)
- Then return A_k

87

87

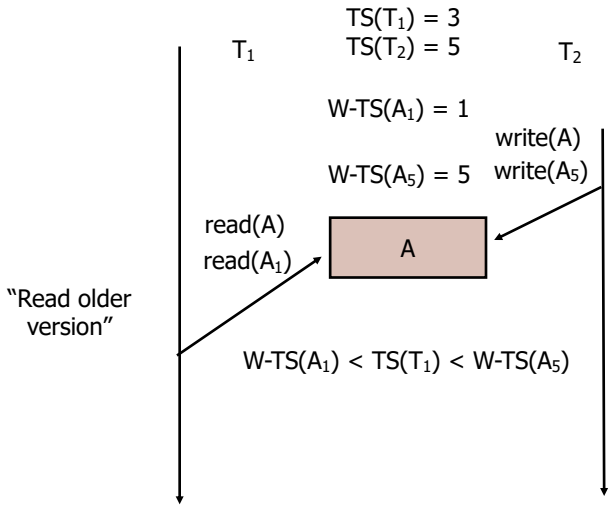
Multiversion Timestamp Ordering

- Suppose *T* issues write(*A*)
- Versions A_1,…, A_k,…, A_m ordered by W-TS(A_i)
- Pick A_k for some k such that
  - W-TS(A_k)<=TS(*T*)
  - W-TS(A_{k+1})>TS(*T*)
- If TS(*T*) < R-timestamp(A_k), then *T* is rolled back
- If TS(*T*) = W-timestamp(A_k), then update A_k
- Otherwise create new version of *A*

88

88

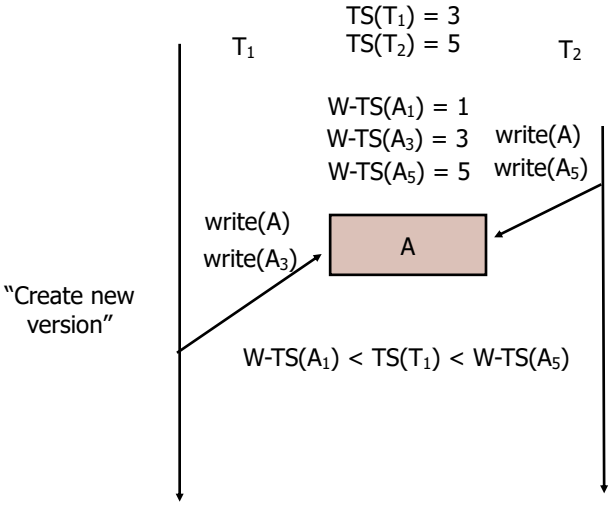
Multiversion Timestamp Ordering



89

89

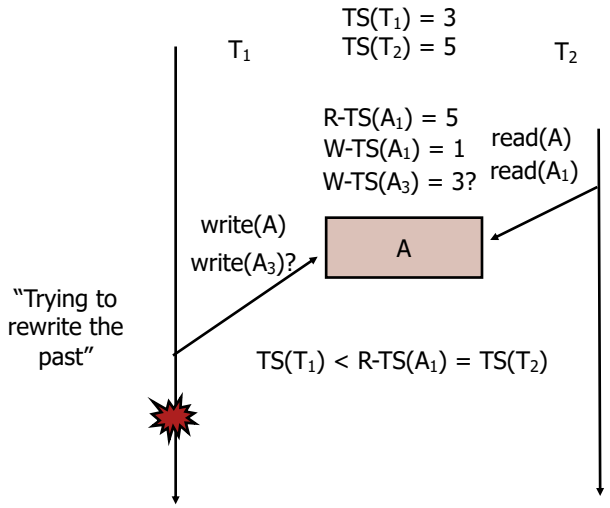
Multiversion Timestamp Ordering



90

90

Multiversion Timestamp Ordering



Multiversion Timestamp Ordering

- Reads always succeed!
- Write by *T* is rejected if already read by "future" transaction T_j
  - Future in serialization order
  - Written by *T* or "older" transaction
- Protocol guarantees serializability

Multiversion Two-Phase Locking

- Global version counter VC for commits
- Type 1: Update transactions
  - Acquire read and write locks
  - Keep locks until commit
  - Write: Create new version of *A*
  - Data item has single timestamp TS(*A*)
- Type 2: Read-only transactions
  - TS(*T*) = current VC when it starts
  - *T* reads "newest" A_k with TS(A_k) <= TS(*T*)

93

93

Multiversion 2PL: Update Transaction

- Read(Q)
  - Acquire read lock
  - Read latest version
- Write(Q)
  - Acquire write lock
  - Create new version A_{m+1} with TS(A_{m+1}) = ∞
- Commit Transaction
  - TS(A_k) := VC for every variable written
  - VC := VC + 1

94

94

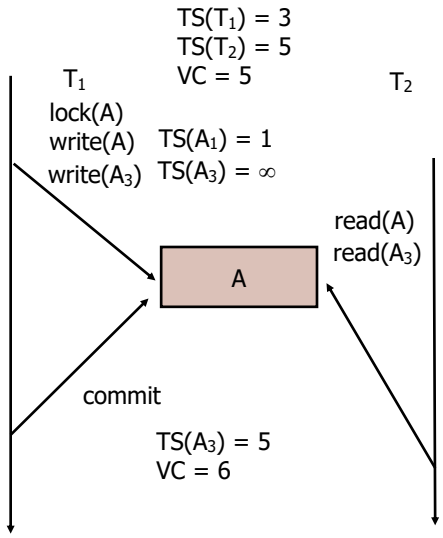
Multiversion 2PL: Read-Only Transaction

- Read-only transaction T_j
  - T_j reads "newest" A_k with TS(A_k) <= TS(T_j)
  - Start before *T*: see the value before *T* updates
  - Start after *T*: see the values updated by *T*
- Only serializable schedules are produced

95

95

Multiversion 2PL



96

96

MVCC: Implementation Issues

- Creation of multiple versions increases storage overhead
  - Extra records
  - Version information
- Versions can be garbage collected

SNAPSHOT ISOLATION

Snapshot Isolation

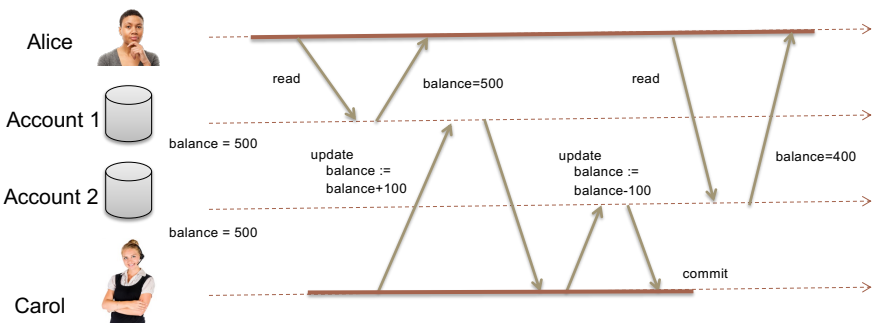
- Motivation: Large data query conflicts with OLTP transactions
- 2PL: Read locks block OLTP
- MVCC: Reads force rollback of OLTP

99

99

Weaker Isolation Levels

- Read Committed:
  - Read Skew: Reading different versions



100

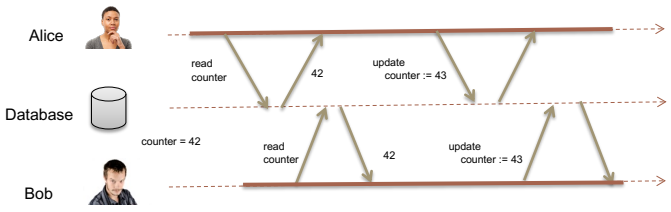
100

Snapshot Isolation

- Motivation: Large data query conflicts with OLTP transactions
- Solution 1: Give logical "snapshot" of database state to read only transactions
  - RW transactions use normal locking
  - Multiversion 2PL
  - How does system know a transaction is read only?

Snapshot Isolation

- Solution 2: Give snapshot of database state to **every** transaction
  - Updates alone use 2PL
  - Problem: anomalies such as lost update



  - Partial solution: snapshot isolation level

Snapshot Isolation

- Transaction *T*
  - takes snapshot of committed data at start
  - reads/modifies data in its own snapshot
  - updates of *concurrent* transactions not visible to *T*
  - writes of *T* complete when it commits
- First-committer-wins rule:
  - Commit if no other concurrent transaction has already committed

103

Snapshot Read

- $X_0 = 100, Y_0 = 0$

T_1: Deposit 50 in Y	T_2: Withdraw 50 from X
rd X_0 → 100	
rd Y_0 → 0	
	rd Y_0 → 0
	rd X_0 → 100
	wr X_2 := 50
wr Y_1 := 50	
rd X_0 → 100	
rd Y_1 → 50	

- $X_2 = 50, Y_1 = 50$

104

Snapshot Write

- $X_0 = 100$

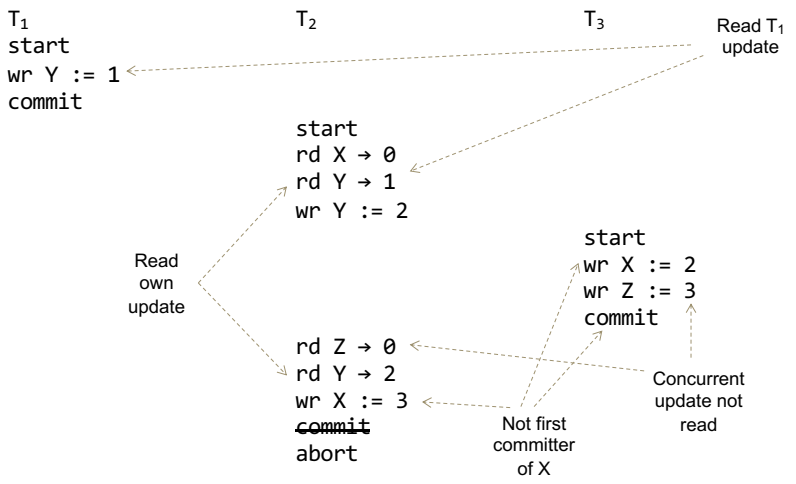
T_1: Deposit 50 in X	T_2: Withdraw 50 from X
rd X_0 → 100	
	rd X_0 → 100
	wr X_2 := 50
wr X_1 := 150	
commit	
	~~commit~~
	abort

- $X_1 = 150$

105

105

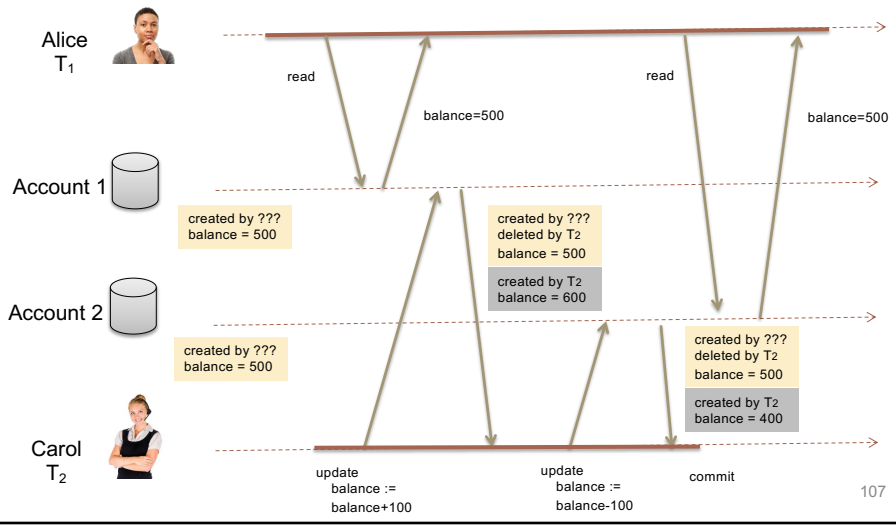
Snapshot Isolation



106

106

Implementing SI



107

107

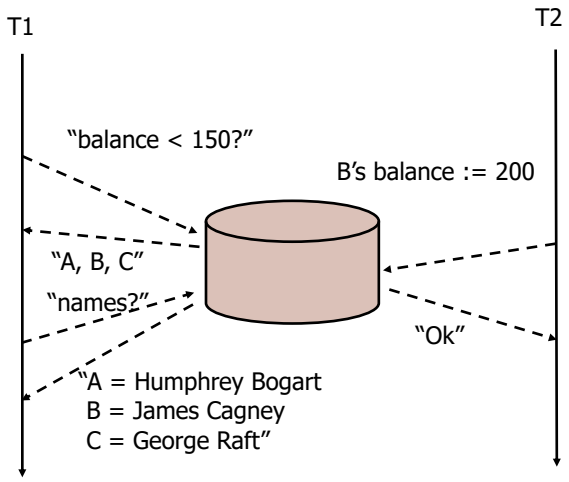
Benefits of SI

- Reading is never blocked
- Performance similar to Read Committed
- Avoids the usual anomalies
  - No dirty read
  - No lost update
  - No non-repeatable read
  - Predicate based selects are repeatable (no phantoms)

108

108

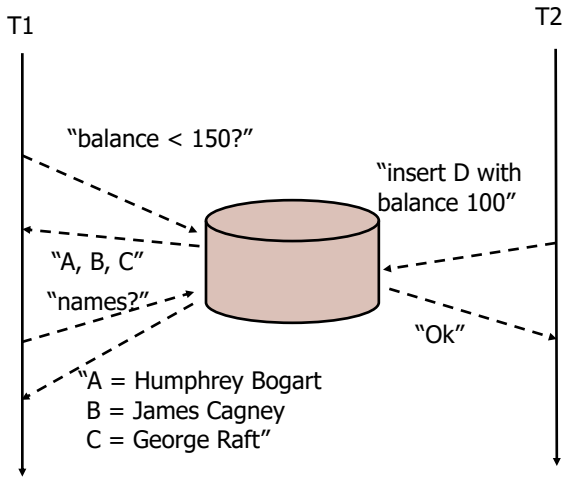
Unrepeatable Read



109

109

Phantom Read

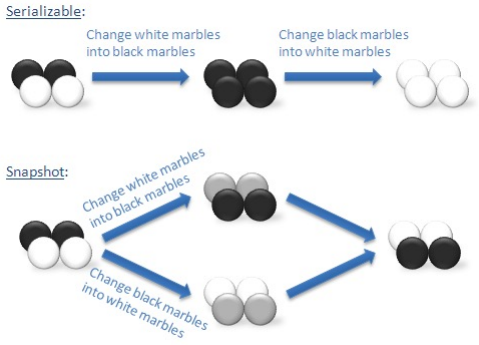


110

110

Problems with SI

- SI does not always give serializable executions



- Result: Integrity constraints can be violated

111

111

Write Skew

- Example
  - T1: x:=y
  - T2: y:= x
  - Initially x = 3 and y = 17
  - Serial execution: x = ??, y = ??
  - Snapshot isolation: x = ? , y = ??
- Example:
  - Find max order number among all purchase orders
  - Create a new purchase order with order number = previous max + 1

112

112

Write Skew and Phantoms

Alice 

name	on_call
Alice	true
Bob	false
Carol	true

Carol 

```
currently_on_call := (
  select count(*) from doctors
  where on_call = true
)
currently_on_call = 2
if (on_call > 1) {
  update doctors
  set on_call=false
  where name = "Alice"
}
commit
```

name	on_call
Alice	false
Bob	false
Carol	true

name	on_call
Alice	false
Bob	false
Carol	false

```
currently_on_call := (
  select count(*) from doctors
  where on_call = true
)
currently_on_call = 2
if (on_call > 1) {
  update doctors
  set on_call=false
  where name = "Carol"
}
commit
```


Write Skew and Phantoms

- Example: Meeting room booking system

```
BEGIN TRANSACTION;
-- Check for any existing bookings noon-1pm
SELECT COUNT(*) FROM bookings
  WHERE room_id = 123
    AND end_time > '2022-01-01 12:00'
    AND start_time < '2022-01-01 13:00';
-- If the previous query returned zero:
INSERT INTO bookings
  (room_id, start_time, end_time, user_id)
  VALUES (123, '2015-01-01 12:00', '2015-01-01 13:00', 666);
COMMIT;
```

Write Skew and Phantoms

- Pattern causing write skew:
  - Query based on some condition
  - INSERT, UPDATE, DELETE based on query
  - Phantom: Operation changes the result of query in another transaction
- What if query tests for absence in database, and operation does INSERT?
- Materializing conflicts
  - Locks just for query results

115

115

Snapshot Isolation and Correctness

- Execution may not be serializable but still correct!
- Example: reserving seats for a concert
  - Integrity constraint: a seat cannot be reserved by more than one person

116

116

Snapshot Isolation and Correctness

- Example: reserving seat X or Y for a concert

T_1:	T_2:
rd X → free	
rd Y → free	
	rd X → free
	rd Y → free
	wr Y := reserved
	commit
wr X := reserved	
commit	

117

117

Snapshot Isolation and Correctness

- Example: reserving seat X or Y for a concert

T_1:	T_2:
rd X → free	
rd Y → free	
	rd X → free
	rd Y → free
	wr X := reserved
	commit
wr X := reserved	
~~commit~~	
abort	

118

118

Preventing Lost Updates

- Atomic Writes

```
UPDATE counters SET value = value + 1 WHERE key = 'foo';
```

- Explicit Locking (SFU)

```
BEGIN TRANSACTION;
SELECT * FROM figures
WHERE name = 'robot' AND game_id = 222
FOR UPDATE;
-- Check whether move is valid, then update the position of a
piece
UPDATE figures SET position = 'c4' WHERE id = 1234;
COMMIT;
```

Preventing Lost Updates

- Atomic Writes
- Explicit Locking
- Automatically detect lost updates
  - Necessary for snapshot isolation?
- Compare-and-set

```
-- May or may not be safe, depending on DB implementation
-- (what if WHERE clause reads from snapshot)
UPDATE wiki_pages SET content = 'new content'
WHERE id = 1234 AND content = 'old content';
```

Preventing Lost Updates

- Conflict resolution and replication
  - Perform operation independently and resolve conflicts
  - Requires commutative operations
  - Otherwise Last Writer Wins (LWW) loses updates

Preventing Write Skew

- Atomic single-object operations
  - Multiple objects involved
- Automatic detection of lost updates
  - Not in practice
- Constraints
  - Requires multi-object constraints
- Explicitly lock rows
- Serializable isolation

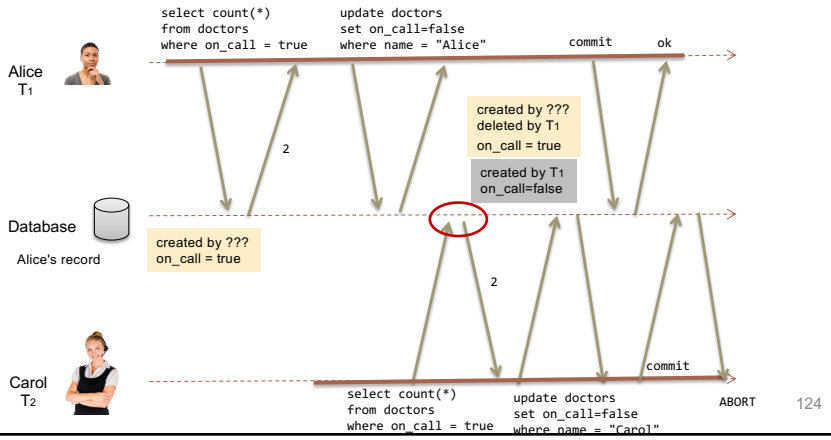
Serializable Snapshot Isolation (SSI)

- Optimistic concurrency control
- Detect read of stale MVCC object
  - Have uncommitted writes committed by transaction end?
- Detect writes that affect prior reads
  - Use index locks to detect prior reads, but don't block

Serializability

name	on_call
Alice	true
Bob	false
Carol	true

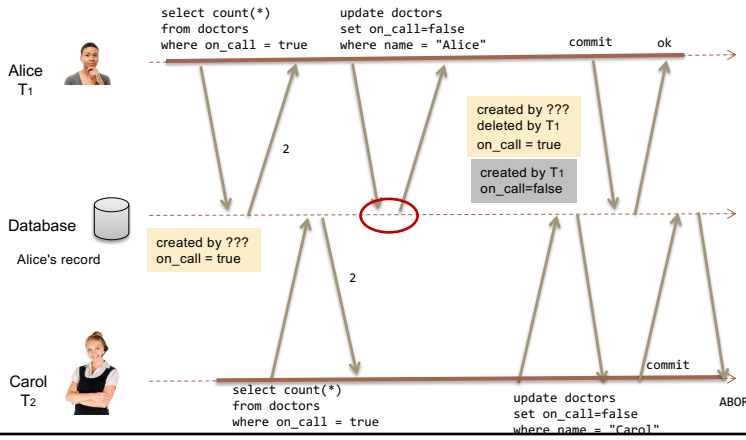
- Serializable Snapshot Isolation: Detecting stale MVCC reads



Serialization

name	on_call
Alice	true
Bob	false
Carol	true

- Serializable Snapshot Isolation: Detecting writes that affect prior reads



SI In Oracle and PostgreSQL

- Oracle
  - "Serializable" implemented by Snapshot Isolation!
  - "First updater wins"
- Postgresql:
  - <9.1: "Serializable" implemented by Snapshot Isolation (!)
  - >=9.1: “Serializable Snapshot Isolation” (SSI)