Multiversion Concurrency Control

Multiversion Schemes

- Multiversion schemes keep old versions of data item to increase concurrency. Several variants:
  - **Multiversion Timestamp Ordering**
  - **Multiversion Two-Phase Locking**
  - **Snapshot isolation**
- Key ideas:
  - Each successful **write** results in the creation of a new version of the data item written.
  - Use timestamps to label versions.
  - When a **read**(*Q*) operation is issued, select an appropriate version of *Q* based on the timestamp of the transaction issuing the read request, and return the value of the selected version.
- **read**s never have to wait as an appropriate version is returned immediately.

Multiversion Timestamp Ordering

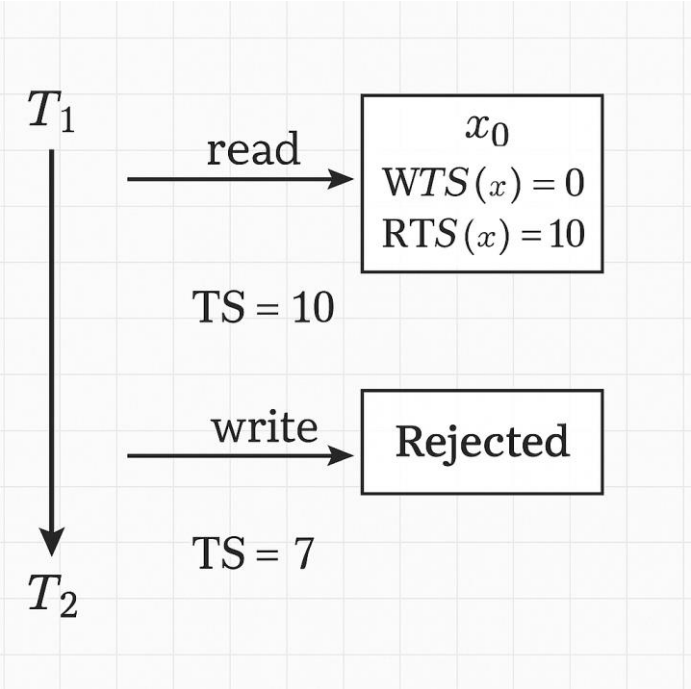
- Each data item Q has a sequence of versions $<Q_1, Q_2, ..., Q_m>$. Each version Q_k contains three data fields:
  - **Content** -- the value of version Q_k.
  - **W-timestamp**(Q_k) -- timestamp of the transaction that created (wrote) version Q_k
  - **R-timestamp**(Q_k) -- largest timestamp of a transaction that successfully read version Q_k

Multiversion Timestamp Ordering (Cont)

- Suppose that transaction T_i issues a **read**(Q) or **write**(Q) operation.
- Let Q_k denote the version of Q whose write timestamp is the largest write timestamp less than or equal to TS(T_i).
  1. If **transaction T_i** issues a **read**(Q), then
     - the value returned is the content of version Q_k
     - If R-timestamp(Q_k) < TS(T_i), set R-timestamp(Q_k) = TS(T_i),
  2. If transaction T_i issues a **write**(Q)
     1. if **TS(T_i) < R-timestamp(Q_k),** then transaction T_i is rolled back.
     2. if **TS(T_i) = W-timestamp(Q_k),** the contents of Q_k are overwritten
     3. Otherwise, a new version Q_i of Q is created
        - W-timestamp(Q_i) and R-timestamp(Q_i) are initialized to TS(T_i).

Multiversion Timestamp Ordering (Cont)

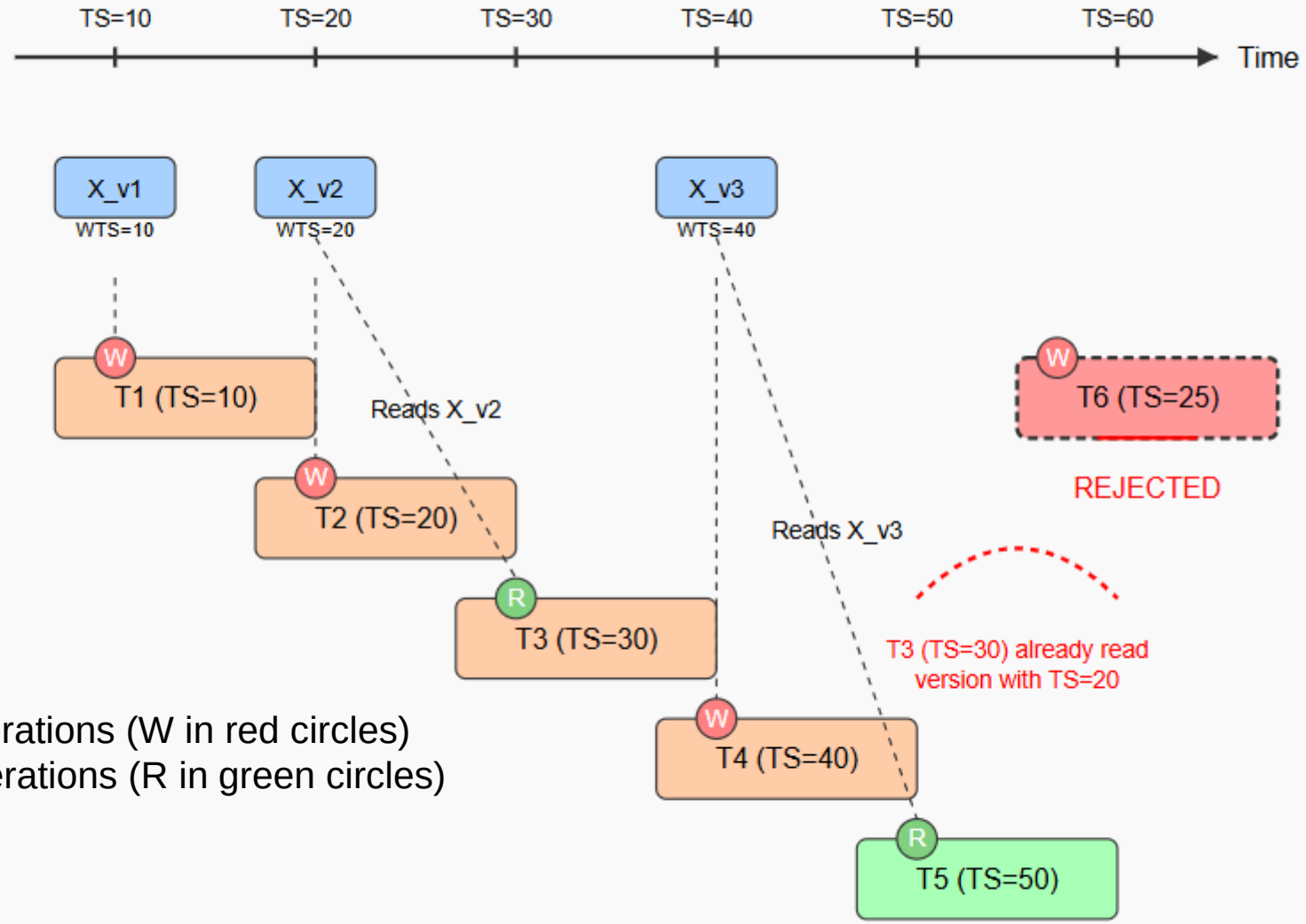
- Observations
  - Reads always succeed
  - A write by T_i is rejected if some other transaction T_j that (in the serialization order defined by the timestamp values) should read T_i's write, has already read a version created by a transaction older than T_i.
- Protocol guarantees serializability



Multiversion Timestamp Ordering (Cont)

```
procedure MVTO_Write(x, T_i):
    // Find the version x_j with the largest
  WTS(x_j) < TS(T_i)
    x_j ← latest_version_of(x) such that
  WTS(x_j) ≤ TS(T_i)
    if RTS(x_j) > TS(T_i) then
        // Conflict: a future-timestamped
  transaction already read the old value
        reject T_i's write on x
    else
        // Safe: install new version
        create version x_i with
            WTS(x_i) ← TS(T_i)
            RTS(x_i) ← TS(T_i)
```

Multiversion Timestamp Ordering



1. Write operations (W in red circles)
2. Read operations (R in green circles)

Multiversion Two-Phase Locking

- **Differentiates** between **read-only transactions** and update transactions
- **Update transactions** acquire read and write locks, **and hold all locks up to the end of the transaction**. That is, **update transactions follow rigorous two-phase locking.**
  - Read of a data item returns the latest version of the item
  - The first **write** of Q by T_i results in the creation of a new version Q_i of the data item Q written
    - W-timestamp(Q_i) set to ∞ **initially to not allow other writes**
  - When **update** transaction T_i **completes**, **commit** processing occurs:
    - Value **ts-counter** stored in the database is used to assign timestamps
      - **ts-counter** is locked in two-phase manner
    - Set **W-timestamp(Q_i)** = (**ts-counter** + 1) for all versions Q_i that it creates
    - **ts-counter** = **ts-counter +** 1
    - Thereby, those transactions that start before $\boldsymbol{T_i}$ commits will see the value before the updates by T_i .

Multiversion Two-Phase Locking

Imagine an online store has a product **P** with a price history. Transactions update or read the price.

Initial State:

- The product **P** has an initial price: **$100,** The **timestamp counter (ts-counter) = 10**.

Transactions:

Transaction T1 (Update Transaction)

- **Starts** at **ts = 11**. **Reads** the latest price (**$100**).
- **First Write**: Creates a **new version P1** with price **$120**.
  - The **write timestamp W-ts(P1) is set to ∞** (blocking other writes).
- **Commit Process:**
- Locks **ts-counter**. Sets **W-ts(P1) = ts-counter + 1 → W-ts(P1) = 11**. Updates **ts-counter = 11**.

Transaction T2 (Read-Only Transaction)

- **Starts at ts = 10** (before T1 commits).
- **Reads product P**, but since **T1 hasn't committed**, it **sees the older price ($100)**.

Multiversion Two-Phase Locking (Cont.)

- **Read-only transactions**
  - are assigned a **timestamp** = **ts-counter** when they start execution
  - follow the multiversion timestamp-ordering protocol for performing reads
    - Do not obtain any locks
- Read-only transactions that start after T_i increments **ts-counter** will see the values updated by T_i.
- Read-only transactions that start before T_i increments the **ts-counter** will see the value before the updates by T_i.
- Only serializable schedules are produced.

Multiversion Two-Phase Locking (Cont.)

- Example
- One product P with initial price: **$100**
- A **global timestamp counter (ts-counter) = 10**

Transaction T1 (Update Transaction)

- **Starts**: ts(T1) = 11
- **Reads**: version P0 (price = $100, W-ts = 10)
- **Tentatively creates** a new version: P1 with price = $120
  - W-ts(P1) = ∞ (not committed yet)
- **At this moment:**
  - `P0`: price = $100, `W-ts = 10`
  - `P1`: price = $120, `W-ts = ∞` (invisible to others)

Transaction	Start Time	Sees Version	W-ts of Version	Observed Price
T2 (read-only)	**10**	P0	10	$100
T1 (update)	**11**	—	—	— (writes P1)
T3 (read-only)	**11**	P1	11	$120

MVCC: Implementation Issues

- Creation of multiple versions increases **storage overhead**
  - **Extra tuples**
  - **Extra space** in each tuple for storing version information
- Versions can, however, be **garbage collected**
  - E.g., if Q has two versions Q5 and Q9, and the oldest active transaction has timestamp > 9, than Q5 will never be required again
- Issues with
  - **primary** key and **foreign key constraint** checking
  - **Indexing** of records with multiple versions

  See textbook for details

Snapshot Isolation

- In databases that serve both **OLTP transactions** (which update a few rows) and **decision support queries** (which read large volumes of data), there is a **concurrency conflict**:
- OLTP transactions require **locking** to ensure consistency during updates.
- Decision support queries scan **large portions of data**, which can hold locks for a long time or be blocked by locks, leading to:
  - **Poor performance** for both types of transactions.
  - **Lock contention**, increasing response times or causing deadlocks.

- What is solution?

Snapshot Isolation

- **Solution 1: Multiversion 2-Phase Locking (MV2PL)**
- The idea is to **separate the read and write workloads** logically:
  - **Read-only transactions** (like decision support queries) get a **"snapshot"** of the database — a consistent view of the data as it was at a point in time.
  - These reads are performed on **that snapshot**, so they **don't interfere with writes**.
  - **Update transactions** (read-write) follow **normal 2-phase locking** protocols.
- ☑ **Advantage**: Eliminates interference between read and write transactions.
- ? **Challenge**: The system must **know in advance** which transactions are **read-only**, which is not always explicit or easy to enforce.

Snapshot Isolation

- **Solution 2 (Partial Fix): Give a Snapshot to Every Transaction**
- **Every transaction**, whether read-only or read-write, reads from a snapshot.
- **Writes** are still done using **2-phase locking** on actual data items.
- ! **Problem**: This can lead to anomalies:
  - **Lost updates**: Two transactions may read the same snapshot and update based on outdated data, overwriting each other's changes.
  - **Write skew** and **phantoms** can also occur.
- **Better Solution: Snapshot Isolation**
  - Snapshot Isolation (discussed on the next slide) is an advanced **concurrency control** technique.
  - It **preserves a consistent snapshot** for readers and **prevents write conflicts** by checking for overlaps in write sets before committing.

Snapshot Isolation

- A transaction T1 executing with Snapshot Isolation
  - Takes **snapshot** of **committed** data at **start**
  - Always **reads/modifies data** in its **own snapshot**
  - **Updates** of **concurrent transactions** are not **visible** to **T1**
  - **Writes** of T1 **complete** when it **commits**
  - **First-committer-wins rule**:
    - Commits only if no other concurrent transaction has already written data that T1 intends to write.

T1	T2	T3
W(Y := 1) Commit		
	Start R(X) → 0 R(Y)→ 1	
		W(X:=2) W(Z:=3) Commit
	R(Z) → 0 R(Y) → 1 W(X:=3) Commit-Req Abort	

Concurrent updates not visible
Own updates are visible
Not first-committer of X
Serialization error, T2 is rolled back

Snapshot Read

- Concurrent updates invisible to snapshot read

$X_0 = 100,\ Y_0 = 0$

T_1 deposits 50 in Y	T_2 withdraws 50 from X
$r_1(X_0, 100)$	
$r_1(Y_0, 0)$	
	$r_2(Y_0, 0)$
	$r_2(X_0, 100)$
	$w_2(X_2, 50)$
$w_1(Y_1, 50)$	
$r_1(X_0, 100)$ (update by T_2 not seen)	
$r_1(Y_1, 50)$ (can see its own updates)	
	$r_2(Y_0, 0)$ (update by T_1 not seen)

$X_2 = 50,\ Y_1 = 50$

Snapshot Write: First Committer Wins

$X_0 = 100$

T_1 deposits 50 in X	T_2 withdraws 50 from X
$r_1(X_0, 100)$	
	$r_2(X_0, 100)$
	$w_2(X_2, 50)$
$w_1(X_1, 150)$	
$commit_1$	
	$commit_2$ (Serialization Error T_2 is rolled back)

$X_1 = 150$

- Variant: "**First-updater-wins**"
  - Check for concurrent updates when write occurs by locking item
    - But lock should be held till all concurrent transactions have finished
  - (Oracle uses **this plus some extra features**)
  - Differs **only in when abort occurs**, otherwise equivalent

Benefits of SI

- **Reads are *never* blocked**,
  - and also don't block other transactions activities
- Performance similar to **Read Committed**
- Avoids **several anomalies**
  - **No** dirty read, i.e. no read of **uncommitted data**
  - **No lost** update
    - I.e., update made by a transaction is overwritten by another transaction that did not see the update)
  - No non-repeatable read
    - I.e., if read is executed again, it will see the same value
- Problems with SI
  - SI does not always give **serializable** executions
    - Serializable: among two concurrent transactions, one sees the effects of the other
    - In SI: neither sees the effects of the other
  - Result: Integrity constraints can be violated

Snapshot Isolation

- Example of problem with SI
  - Initially A = 3 and B = 17
    - Serial execution: A = ??, B = ??
    - if both transactions start at the same time, with snapshot isolation: A = ?? , B = ??
- Called **skew write**
- Skew also occurs with inserts
  - E.g:
    - Find max order number among all orders
    - Create a new order with order number = previous max + 1
    - Two transaction can both create order with same number
      - Is an example of phantom phenomenon

T_i	T_j
read(A)	
read(B)	
	read(A)
	read(B)
A=B	
	B=A
write(A)	
	write(B)

Snapshot Isolation Anomalies

- **SI is not fully serializable**: It can break serializability when transactions modify **different items**, each based on **outdated data** modified by the other (known as **write skew**).
- **Not common in practice**: Most real-world workloads (e.g., **TPC-C benchmark**) behave correctly under SI because:
  - Conflicting transactions often also modify a **shared item**, causing SI to detect the conflict and **abort one**.
- **Key Issues with SI:**
- **Write skew**: Two transactions read consistent data, then **write to disjoint items**, creating an inconsistent combined state.
- **Read-only anomaly**: Even read-only transactions can see an **inconsistent snapshot**, despite all update transactions being serializable.
- **Integrity constraint issues**: Using snapshots to check **primary/foreign key** constraints can lead to **inconsistencies**, as constraints are usually enforced **outside the snapshot mechanism**.

Snapshot Isolation Anomalies

- ◇ **Read-Only Transaction Anomalies under Snapshot Isolation (SI):**
- Under **Snapshot Isolation (SI)**, each transaction reads from a **consistent snapshot** of the database taken at the start of the transaction. However, this can lead to **anomalies for read-only transactions**, even if all **updater transactions** are individually serializable.
- ❗ **Issue:**
- A **read-only transaction** may observe a **state that never existed in any serial execution** of the update transactions. This happens because:
- SI allows multiple transactions to **commit independently** as long as their **write sets do not overlap**, even if their **read sets** do.
- A read-only transaction can read from a **mix of states**, reflecting effects of some transactions but not others, leading to an **inconsistent view**.

Snapshot Isolation Anomalies

- ◇ **Primary/Foreign Key Inconsistency under SI:**
- Using SI to verify **referential integrity** (e.g., foreign key constraints) can be problematic.
- ❗ **Example Problem:**
- Transaction A deletes a parent row.
- Transaction B concurrently inserts a child row referencing the deleted parent.
- If both operate on disjoint data and do not see each other's updates (due to SI), they may both commit.
- A read-only transaction or constraint checker may then see the **child without its parent**, **violating foreign key integrity**.
- ⚠ **Why This Happens:**
- **Integrity constraint checking** is typically done **outside the snapshot** context, often at commit time.
- SI does not coordinate these checks across transactions unless explicitly programmed.

Serializable Snapshot Isolation

- **Serializable snapshot isolation (SSI)**: extension of snapshot isolation that ensures serializability
- Snapshot isolation tracks write-write conflicts, but does not track read-write conflicts
  - Where T_i writes a data item Q, T_j reads an earlier version of Q, but T_j is serialized after T_i
- Idea: track read-write dependencies separately, and roll-back transactions where cycles can occur
  - Ensures serializability
  - Details in book
- Implemented in PostgreSQL from version 9.1 onwards
  - PostgreSQL implementation of SSI also uses index locking to detect phantom conflicts, thus ensuring true serializability

SI Implementations

- Snapshot isolation supported by many databases
  - Including Oracle, PostgreSQL, SQL Server, IBM DB2, etc
  - Isolation level can be set to snapshot isolation
- Oracle implements "first updater wins" rule (variant of "first committer wins")
  - Concurrent writer check is done at time of write, not at commit time
  - Allows transactions to be rolled back earlier: The **second updater is blocked or aborted immediately** when it tries to write, rather than waiting until commit.
- **Warning**: *even if isolation level is set to serializable, Oracle actually uses snapshot isolation*
  - Old versions of PostgreSQL prior to 9.1 did this too
  - Oracle and PostgreSQL < 9.1 do not support true serializable execution

Working Around SI Anomalies

In **Snapshot Isolation (SI)**, certain anomalies like **write skew** may occur due to the lack of coordination between concurrent transactions that **read and write disjoint data**. However, some of these anomalies can be avoided by explicitly **locking the rows** involved in the query using the SELECT ... FOR UPDATE clause.

- ***The SELECT ... FOR UPDATE statement:***
  - Retrieves rows **just like a regular SELECT**, **but** also **locks** those rows for writing.
  - Treats the **read operation as if it were an update**, thus preventing **concurrent transactions** from reading or writing the same rows until the lock is released (usually at commit time).

Working Around SI Anomalies

- -- Transaction T1:
  - **SELECT MAX(orderno) FROM orders FOR UPDATE;**
    - -- Suppose it returns 100
- -- T1 assigns 101 to a new order
  - **INSERT INTO orders (orderno, ...) VALUES (101, ...);**

- The FOR UPDATE clause locks the row containing the maximum order number (or the index if applicable).
- This prevents **another concurrent transaction** (e.g., T2) from reading the same MAX(orderno) value and inserting the same or conflicting value.
- As a result, **only one transaction** can safely generate the next order number at a time, thus preserving **consistency**.
- It does **not prevent all anomalies**, especially those involving **phantom reads** or **predicate-based queries**.

Working Around SI Anomalies

- **SELECT * FROM accounts WHERE balance > 500 FOR UPDATE;**
  - This locks only the currently matching rows.
  - If a concurrent transaction inserts a new row that also satisfies the predicate, this is a phantom, and SI (and FOR UPDATE) alone cannot prevent it.
  - Preventing phantoms typically requires predicate locking or true serializable isolation (e.g., via Serializable Snapshot Isolation in PostgreSQL ≥ 9.1).