Database Processing
CS 451 / 551

Lecture 14:
Isolation Levels and
Multi-Version Concurrency Control

Suyash Gupta
Assistant Professor
Distopia Labs and ONRG
Dept. of Computer Science
(E) suyash@uoregon.edu
(W) gupta-suyash.github.io

UNIVERSITY OF OREGON

Assignment 3 is Out!
Deadline: Nov 30, 2025 at 11:59pm

Presentation Slots are Out.

Final Exam: Dec 8, 2025 at 8-10am

Syllabus → Main focus on course not covered in Midterm, but you should understand indexes and storage.

Last Class

- We discussed Timestamp Ordering and Forward validation.

- In **Forward Validation**, at the time of commit, each transaction checks for conflicts with ongoing transactions.

Backward Validation

Backward Validation

- At the time of commit, each transaction checks if it conflicts with other **already committed transactions** (transactions which were concurrent and have committed).

- Each going to commit transaction (at the validation step), checks the timestamps and read/write sets of other committed transactions.

OCC Disadvantages

OCC Disadvantages

- There is an overhead of copying data to private workspace.
  - More data to copy, more expensive!
- Validation/Write phase creates bottlenecks due to locking.
- Aborting a transaction is more expensive in OCC than in 2PL because it occurs after a transaction has already executed.

Queries Isolation

T1:	T2:
Begin	**Begin**
select count(*) as cnt **from** cs_cmployees **where** age > 30	
	insert into cs_employees **values** (anakin, 70, 500)
select count(*) as cnt **from** cs_cmployees **where** age > 30	**Commit**
Commit	

Time

```
create table cs_employees
        (       name    varchar(20),
                age     int,
                salary  int
        );
```

Is there any problem with this schedule?

Queries Isolation

Time

T1:	T2:
Begin	**Begin**
select count(*) as cnt **from** cs_cmployees **where** age > 30	
	insert into cs_employees **values** (anakin, 70, 500)
select count(*) as cnt **from** cs_cmployees **where** age > 30	**Commit**
Commit	

```
create table cs_employees
    (   name    varchar(20),
        age     int,
        salary  int
    );
```

Queries Isolation

	T1:	T2:
	Begin	**Begin**
	select count(*) as cnt **from** cs_cmployees **where** age > 30	
Time		**insert into** cs_employees **values** (anakin, 70, 500)
	select count(*) as cnt **from** cs_cmployees **where** age > 30	**Commit**
	Commit	

```
create table cs_employees
    (    name    varchar(20),
         age     int,
         salary  int
    );
```

Say, output = 10

Queries Isolation

Time

T1:	T2:
Begin	**Begin**
select count(*) as cnt **from** cs_cmployees **where** age > 30	
	insert into cs_employees **values** (anakin, 70, 500)
select count(*) as cnt **from** cs_cmployees **where** age > 30	**Commit**
Commit	

```
create table cs_employees
	(	name	varchar(20),
		age	int,
		salary	int
	);
```

Added a new record.

Queries Isolation

Time	T1:	T2:
	Begin	Begin
	select count(*) as cnt from cs_cmployees where age > 30	
		insert into cs_employees values (anakin, 70, 500)
	select count(*) as cnt from cs_cmployees where age > 30	Commit
	Commit	

```
create table cs_employees
    (   name    varchar(20),
        age     int,
        salary  int
    );
```

Now, output = 11

Queries Isolation

Time	T1:	T2:
	Begin	**Begin**
	select count(*) as cnt **from** cs_cmployees **where** age > 30	
		insert into cs_employees **values** (anakin, 70, 500)
	select count(*) as cnt **from** cs_cmployees **where** age > 30	**Commit**
	Commit	

create table cs_employees
(
name **varchar**(20),
age **int,**
salary **int**
);

The output of the two queries changed!

Phantom Problem

Time

T1:	T2:
Begin	**Begin**
select count(*) as cnt **from** cs_cmployees **where** age > 30	
	insert into cs_employees **values** (anakin, 70, 500)
select count(*) as cnt **from** cs_cmployees **where** age > 30	**Commit**
Commit	

```
create table cs_employees
        (       name    varchar(20),
                age     int,
                salary  int
        );
```

This is also termed as a phantom problem.

Phantom Problem

Time	T1:	T2:
	Begin	**Begin**
	select count(*) as cnt **from** cs_cmployees **where** age > 30	
		insert into cs_employees **values** (anakin, 70, 500)
	select count(*) as cnt **from** cs_cmployees **where** age > 30	**Commit**
	Commit	

```
create table cs_employees
    (   name    varchar(20),
        age     int,
        salary  int
    );
```

Violates our traditional definition of 2PL?
T1 cannot take a lock on something that does not exist!

Why Phantom Problem?

Why Phantom Problem?

- We took read/write locks on existing records, and our locking scheme assumed a static system.

- But real-world databases are dynamic.

- Concurrent transactions can add new records, and our locking scheme does not consider insertions, deletions, and updates.

Solutions to Phantom Problem

Index Locking Schemes

- Index locking schemes can help eliminate phantom problem.

- Four key mechanisms in index locking schemes:
  - Key-Value Locks
  - Gap Locks
  - Key-Range Locks
  - Hierarchical Locking

Key-Value Locks

- Locks that cover a single key-value pair in an index → Standard Locks.
- For non-existent key-value pairs, we would need **virtual keys**.

B+-tree Leaf Nodes 6 8 10 12

Key-Value Locks

- Locks that cover a single key-value pair in an index → Standard Locks.
- For non-existent key-value pairs, we would need **virtual keys**.

B^+-tree Leaf Nodes 6 8 $\boxed{10}$ 12

Update 10 → Lock 10.

Gap Locks

- Locks acquired on empty slots or gaps in the index.
- Gaps are like missing possible keys in the index.

B^+-tree Leaf Nodes	6	8	10	12

Gap Locks

- Locks acquired on empty slots or gaps in the index.
- Gaps are like missing possible keys in the index.

B+-tree Leaf Nodes	6	8	10	12

Say, we want to take a lock on gap between 10-12.

Gap Locks

- Locks acquired on empty slots or gaps in the index.
- Gaps are like missing possible keys in the index.
- Once a gap lock is taken, only the locking transaction can modify the gap.

Say, we want to take a lock on gap between 10-12.

Key-Range Locks

- Locks that cover a key and a gap → Key-lock + Gap-lock.

B+-tree Leaf Nodes 6 8 10 12

Key-Range Locks

- Locks that cover a key and a gap → Key-lock + Gap-lock.

B^+-tree Leaf Nodes 6 8 10 12

Say, we want to take a lock from 10-12.

Key-Range Locks

- Locks that cover a key and a gap → Key-lock + Gap-lock.

B^+-tree Leaf Nodes 6 8 $\boxed{10 \quad 12}$

Say, we want to take a lock from 10-12.

Hierarchical Locks

- Allow a transaction to acquire key-range locks in a wider variety of modes.
- Remember the locking granularity matrix.

B$^+$-tree Leaf Nodes	6	8	10	12

Hierarchical Locks

- Allow a transaction to acquire key-range locks in a wider variety of modes.
- Remember the locking granularity matrix.

B+-tree Leaf Nodes	6	8	10	12

Say, T1 wants to read all numbers from 6 to 12 (excluding 12) and update 10 to 12.

Hierarchical Locks

- Allow a transaction to acquire key-range locks in a wider variety of modes.
- Remember the locking granularity matrix.

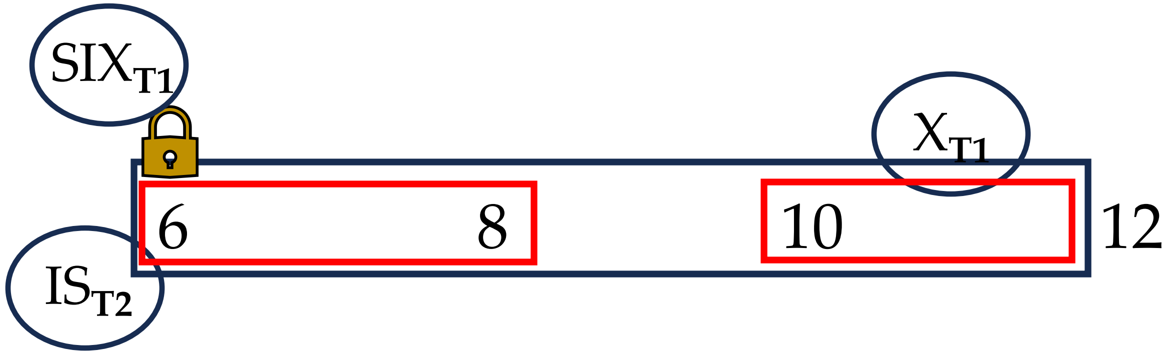
B^+-tree Leaf Nodes

Say, T1 wants to read all numbers from 6 to 12 (excluding 12) and update 10 to 12.

Hierarchical Locks

- Allow a transaction to acquire key-range locks in a wider variety of modes.
- Remember the locking granularity matrix.

B⁺-tree Leaf Nodes



Say, T1 wants to read all numbers from 6 to 12 (excluding 12) and update 10 to 12.

Say, T2 wants to read all numbers from 6 to 8.

Weaker Levels of Isolation

Weaker Levels of Isolation

- Serializability permits programmers to ignore concurrency issues.
- But enforcing serializability restricts opportunities for concurrency and limits performance.
- **Solution**? → Use a weaker level of consistency to improve scalability.

Weaker Levels of Isolation

- Isolation Levels control the extent to which a transaction is exposed to the actions of other concurrent transactions.

- Providing greater concurrency leads to several challenges:
  - Dirty Reads (W-R)
  - Unrepeatable Reads (R-W)
  - Lost Updates (W-W)
  - Phantom Reads

Weaker Levels of Isolation

Isolation
(High → Low)

- **Serializable:** no phantoms, all reads repeatable, no dirty reads.

Weaker Levels of Isolation

Isolation
(High → Low)

- **Serializable:** no phantoms, all reads repeatable, no dirty reads.
- **Repeatable Reads:** phantoms may happen.

Weaker Levels of Isolation

Isolation
(High → Low)

- **Serializable:** no phantoms, all reads repeatable, no dirty reads.
- **Repeatable Reads:** phantoms may happen.
- **Read Committed:** phantoms, unrepeatable reads, and lost updates may happen.

Weaker Levels of Isolation

Isolation
(High → Low)

- **Serializable:** no phantoms, all reads repeatable, no dirty reads.
- **Repeatable Reads:** phantoms may happen.
- **Read Committed:** phantoms, unrepeatable reads, and lost updates may happen.
- **Read Uncommitted:** all anomalies may happen.

Weaker Levels of Isolation

Isolation
(High → Low)

- **Serializable:** Strong Strict 2PL with phantom protection (example through use of index locks)
- **Repeatable Reads:** Same as above, but without phantom protection.
- **Read Committed:** Same as above, but S-Locks are released immediately.
- **Read Uncommitted:** Same as above but allows dirty reads (no S-Locks).

Multi-Version Concurrency Control

Multi-Version Concurrency Control

- The DBMS maintains multiple physical versions of each record in the database.
- When a transaction reads a record, it reads the newest version that existed when the transaction started.
- When a transaction writes/updates a record, the DBMS creates a new version of that record.

Multi-Version Concurrency Control

- In MVCC,
  - Writers do not block readers.
  - Readers do not block writers.

- Read-only transactions can read from a **consistent snapshot** without acquiring locks.

- MVCC uses timestamps to determine visibility.

- MVCC provides support for **time-travel queries** if you skip doing garbage collection.
  - Run this query on the database state 2 weeks ago.

Multi-Version Concurrency Control

- How does MVCC work?

Multi-Version Concurrency Control

- How does MVCC work?

- For each transaction:
  - Create a new version on write.
  - Assign a begin timestamp and end timestamp.
  - End timestamp of previous version = begin timestamp of new version.

- Remember, we will still try to maintain isolation.
  - A concurrent transaction should not see uncommitted versions.
  - Concurrent transactions should read only committed versions.
  - View uncommitted versions as written in the local/logical space.

Example 1

Time

T1:	T2:
Begin	
read(**A**)	
	Begin
read(**A**)	write(**A**)
Commit	
	Commit

Database

Object	Value	Begin-TS	End-TS
A_0	100		

Example 1

Time

T1:	T2:
Begin	
read(**A**)	
	Begin
read(**A**)	write(**A**)
Commit	
	Commit

$TS(T_1) = 1$

Database

Object	Value	Begin-TS	End-TS
A_0	100	0	

At start of the transaction get a begin timestamp.

Example 1

Time

T1:	T2:
Begin	
read(A)	
	Begin
read(A)	write(A)
Commit	
	Commit

$TS(T_1) = 1$

Database

Object	Value	Begin-TS	End-TS
A_0	100	0	

Next, read the latest version from the database.

Example 1

T1:	T2:
Begin	
read(**A**)	
	Begin
read(**A**)	write(**A**)
Commit	
	Commit

Time

$TS(T_1) = 1$

$TS(T_2) = 2$

Database

Object	Value	Begin-TS	End-TS
A_0	100	0	

Begin timestamp for T2.

Example 1

Time

T1:	T2:
Begin	
read(**A**)	
	Begin
read(**A**)	write(**A**)
Commit	
	Commit

$TS(T_1) = 1$

$TS(T_2) = 2$

Database

Object	Value	Begin-TS	End-TS
A_0	100	0	
A_1	200		

Create a new version for T2.

Example 1

Time	T1:	T2:
	Begin	
	read(**A**)	
		Begin
	read(**A**)	write(**A**)
	Commit	
		Commit

$TS(T_1) = 1$

$TS(T_2) = 2$

Database

Object	Value	Begin-TS	End-TS
A_0	100	0	
A_1	200	2	

Begin Timestamp for the new version of A is T2's begin timestamp.

Example 1

Time

T1:		T2:
Begin		
read(**A**)		
		Begin
read(**A**)	➡	write(**A**)
Commit		
		Commit

$TS(T_1) = 1$

$TS(T_2) = 2$

Database

Object	Value	Begin-TS	End-TS
A_0	100	0	2
A_1	200	2	

Set end Timestamp for the previous.

Example 1

T1:	T2:
Begin	
read(**A**)	
	Begin
read(**A**)	write(**A**)
Commit	
	Commit

Time

$TS(T_1) = 1$

$TS(T_2) = 2$

Database

Object	Value	Begin-TS	End-TS
A_0	100	0	2
A_1	200	2	

But how would a future transaction know which version should it read or which is the committed version?

Example 1

Time

T1:		T2:
Begin		
read(**A**)		
		Begin
read(**A**)	⟹	write(**A**)
Commit		
		Commit

$TS(T_1) = 1$

$TS(T_2) = 2$

Database

Object	Value	Begin-TS	End-TS
A_0	100	0	2
A_1	200	2	

Transaction Status

Object	Timestamp	Status
T_1	1	Active
T_2	2	Active

Maintain transaction status table!

Example 1

Time	T1:		T2:
	Begin		
	read(**A**)		
			Begin
	read(**A**)	➡	write(**A**)
	Commit		
			Commit

$TS(T_1) = 1$

$TS(T_2) = 2$

Database

Object	Value	Begin-TS	End-TS
A_0	100	0	2
A_1	200	2	

Transaction Status

Object	Timestamp	Status
T_1	1	Active
T_2	2	Active

Assume a new transaction T3 arrives at this moment and wants to read A, which version should it read?

Example 1

Time

T1:	T2:
Begin	
read(**A**)	
	Begin
read(**A**)	write(**A**)
Commit	
	Commit

$TS(T_1) = 1$

$TS(T_2) = 2$

Database

Object	Value	Begin-TS	End-TS
A_0	100	0	2
A_1	200	2	

Transaction Status

Object	Timestamp	Status
T_1	1	Active
T_2	2	Active

If T3 is allowed to read only committed changes, then version A_0, and if uncommitted changes are allowed then A_1.

Example 1

Time	T1:	T2:	
	Begin		$TS(T_1) = 1$
	read(**A**)		
		Begin	$TS(T_2) = 2$
	read(**A**)	write(**A**)	
	Commit		
		Commit	

Database

Object	Value	Begin-TS	End-TS
A_0	100	0	2
A_1	200	2	

Transaction Status

Object	Timestamp	Status
T_2	2	Active

T1 commits

Example 1

Time	T1:	T2:	
	Begin		$TS(T_1) = 1$
	read(**A**)		
		Begin	$TS(T_2) = 2$
	read(**A**)	write(**A**)	
	Commit		
		→ **Commit**	

Database

Object	Value	Begin-TS	End-TS
A_0	100	0	2
A_1	200	2	

Transaction Status

Object	Timestamp	Status

T2 commits.

Example 2

Time

T1:	T2:
Begin	
read(**A**)	
write(A)	
	Begin
	read(**A**)
	write(**A**)
read(**A**)	
Commit	
	Commit

$TS(T_1) = 1$

Database

Object	Value	Begin-TS	End-TS
A_0	100	0	

Transaction Status

Object	Timestamp	Status
T_1	1	Active

Example 2

Time

T1:	T2:
Begin	
read(**A**)	
write(A)	
	Begin
	read(**A**)
	write(**A**)
read(**A**)	
Commit	
	Commit

$TS(T_1) = 1$

Database

Object	Value	Begin-TS	End-TS
A_0	100	0	

Transaction Status

Object	Timestamp	Status
T_1	1	Active

Example 2

Time

T1:	T2:
Begin	
read(**A**)	
write(A)	
	Begin
	read(**A**)
	write(**A**)
read(**A**)	
Commit	
	Commit

$TS(T_1) = 1$

Database

Object	Value	Begin-TS	End-TS
A_0	100	0	1
A_1	200	1	

Transaction Status

Object	Timestamp	Status
T_1	1	Active

New version due to write operation.

Example 2

Time

T1:

Begin
read(**A**)
write(A)

read(**A**)
Commit

T2:

Begin
read(**A**)
write(**A**)

Commit

$TS(T_1) = 1$

$TS(T_2) = 2$

Database

Object	Value	Begin-TS	End-TS
A_0	100	0	1
A_1	200	1	

Transaction Status

Object	Timestamp	Status
T_1	1	Active
T_2	2	Active

Example 2

Time

T1:	T2:
Begin	
read(**A**)	
write(A)	**Begin**
	➡ read(**A**)
	write(**A**)
read(**A**)	
Commit	
	Commit

$TS(T_1) = 1$

$TS(T_2) = 2$

Database

Object	Value	Begin-TS	End-TS
A_0	100	0	1
A_1	200	1	

Transaction Status

Object	Timestamp	Status
T_1	1	Active
T_2	2	Active

What version will T2 read?

Example 2

T1:

Begin
read(**A**)
write(A)

read(**A**)
Commit

T2:

Begin
read(**A**)
write(**A**)

Commit

$TS(T_1) = 1$

$TS(T_2) = 2$

Time

Database

Object	Value	Begin-TS	End-TS
A_0	100	0	1
A_1	200	1	

Transaction Status

Object	Timestamp	Status
T_1	1	Active
T_2	2	Active

T2 will read version A_0 because A_1 is not committed yet as T_1 is still Active!

Example 2

Time	T1:	T2:
	Begin	
	read(**A**)	
	write(A)	**Begin**
		read(**A**)
		write(**A**)
	read(**A**)	
	Commit	
		Commit

$TS(T_1) = 1$

$TS(T_2) = 2$

Database

Object	Value	Begin-TS	End-TS
A_0	100	0	1
A_1	200	1	

Transaction Status

Object	Timestamp	Status
T_1	1	Active
T_2	2	Active

What happens now?

Example 2

Time

T1:	T2:
Begin	
read(**A**)	
write(A)	
	Begin
	read(**A**)
	write(**A**)
read(**A**)	
Commit	
	Commit

$TS(T_1) = 1$

$TS(T_2) = 2$

Database

Object	Value	Begin-TS	End-TS
A_0	100	0	1
A_1	200	1	

Transaction Status

Object	Timestamp	Status
T_1	1	Active
T_2	2	Active

This write operation should wait otherwise T2 will create a conflicting version.

Example 2

Time

T1:	T2:
Begin	
read(**A**)	
write(A)	
	Begin
	read(**A**)
	⟶ write(**A**)
read(**A**)	
Commit	
	Commit

$TS(T_1) = 1$

$TS(T_2) = 2$

Database

Object	Value	Begin-TS	End-TS
A_0	100	0	1
A_1	200	1	

Transaction Status

Object	Timestamp	Status
T_1	1	Active
T_2	2	Active

Essentially, T2 is not allowed to create any version until T1 commits!

Example 2

T1:	T2:	
Begin		$TS(T_1) = 1$
read(**A**)		
write(A)		
	Begin	$TS(T_2) = 2$
	read(**A**)	
	write(**A**)	
read(**A**)		
Commit		
	Commit	

Time

Database

Object	Value	Begin-TS	End-TS
A_0	100	0	1
A_1	200	1	

Transaction Status

Object	Timestamp	Status
T_1	1	Active
T_2	2	Active

T1 reads version A_1 as it is local to T1.

Example 2

	T1:	T2:	
Time	**Begin**		$TS(T_1) = 1$
	read(**A**)		
	write(A)	**Begin**	$TS(T_2) = 2$
		read(**A**)	
		write(**A**)	
	read(**A**)		
	Commit		
		Commit	

Database

Object	Value	Begin-TS	End-TS
A_0	100	0	1
A_1	200	1	

Transaction Status

Object	Timestamp	Status
T_1	1	Committed
T_2	2	Active

T1 commits and now we allow T2 to continue running.

Example 2

Time ↓	T1:	T2:
	Begin	
	read(**A**)	
	write(A)	**Begin**
		read(**A**)
		→ write(**A**)
	read(**A**)	
	Commit	
		Commit

$TS(T_1) = 1$

$TS(T_2) = 2$

Database

Object	Value	Begin-TS	End-TS
A_0	100	0	1
A_1	200	1	2
A_2	200	2	

Transaction Status

Object	Timestamp	Status
T_1	1	Committed
T_2	2	Active

T1 commits and now we allow T2 to continue running.

Example 2

Time

T1:	T2:
Begin	
read(**A**)	
write(A)	
	Begin
	read(**A**)
	write(**A**)
read(**A**)	
Commit	
	Commit

$TS(T_1) = 1$

$TS(T_2) = 2$

Database

Object	Value	Begin-TS	End-TS
A_0	100	0	1
A_1	200	1	2
A_2	200	2	

Transaction Status

Object	Timestamp	Status
T_1	1	Committed
T_2	2	Active

Is this serializable?

Example 2

	T1:	T2:
Time ↓		
	Begin	
	read(**A**)	
	write(A)	**Begin**
		read(**A**)
		→ write(**A**)
	read(**A**)	
	Commit	
		Commit

$TS(T_1) = 1$

$TS(T_2) = 2$

Database

Object	Value	Begin-TS	End-TS
A_0	100	0	1
A_1	200	1	2
A_2	200	2	

Transaction Status

Object	Timestamp	Status
T_1	1	Committed
T_2	2	Active

Is this serializable? No, because T2 reads an older version.

Snapshot Isolation

- When a transaction starts, it sees a consistent snapshot of the database.
  - Snapshot of the database that existed when that the transaction started.

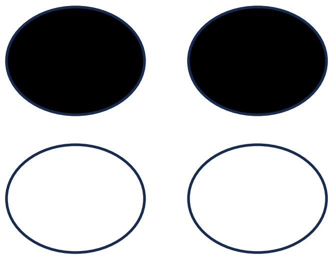
- No uncommitted writes from active transactions are visible.

- If two transactions update the same object, then the first writer does not wait.

- SI sometimes faces the **Write Skew Anomaly**.

Write Skew Anomaly

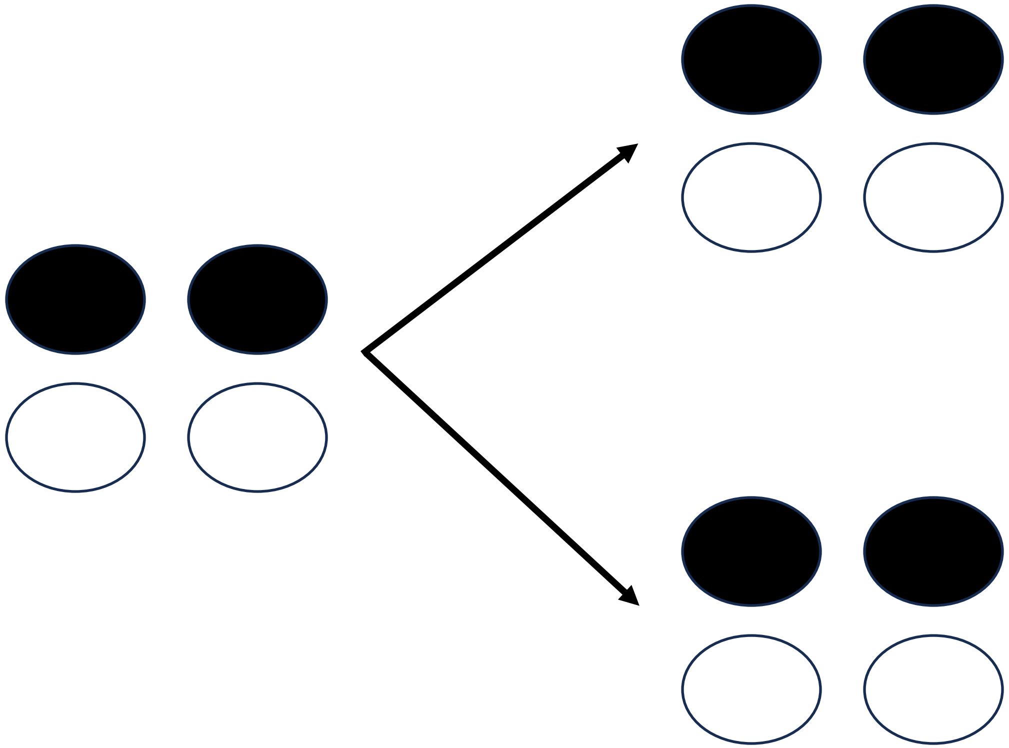
T1 → Turn all balls Black



T2 → Turn all balls White

Write Skew Anomaly

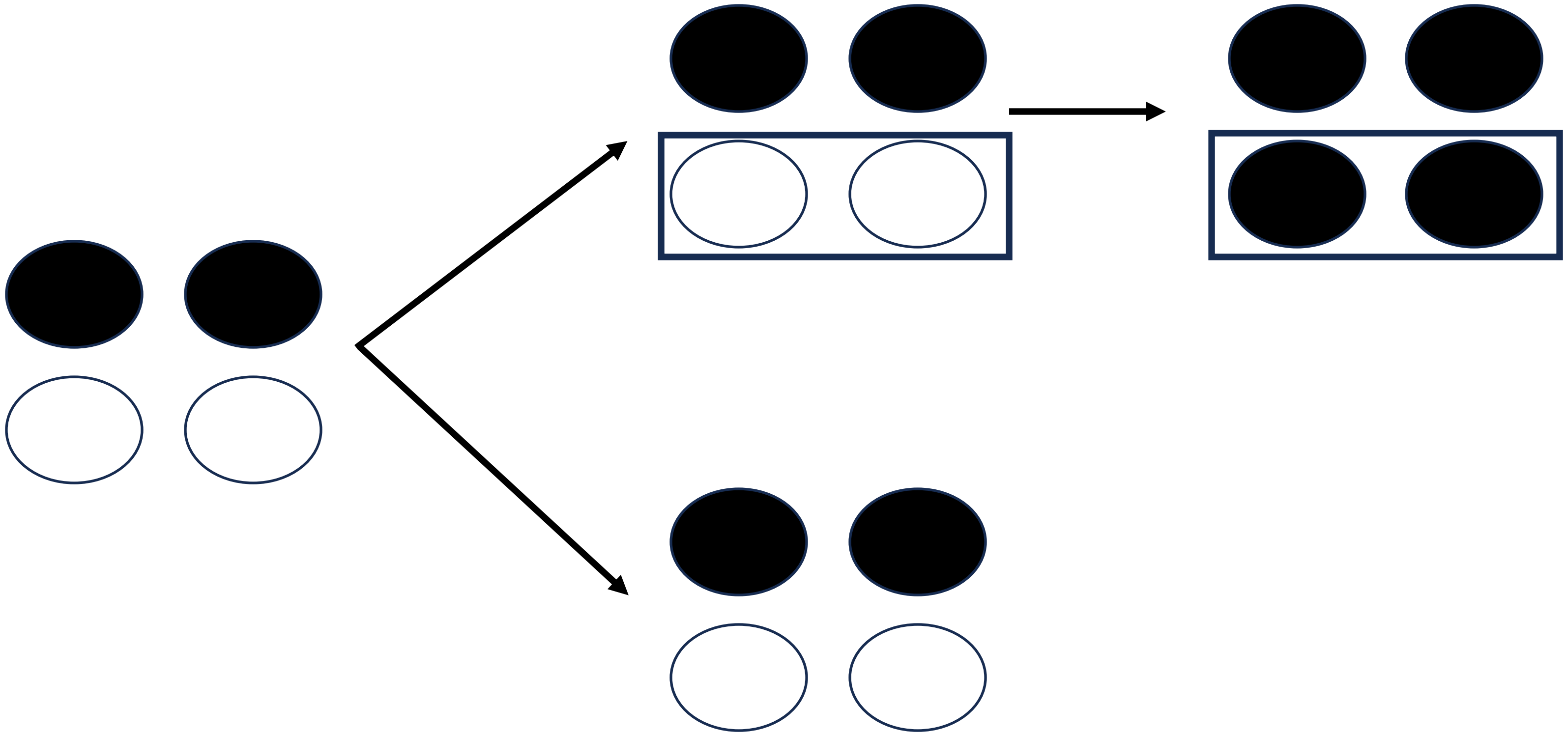
T1 → Turn all balls Black



T2 → Turn all balls White

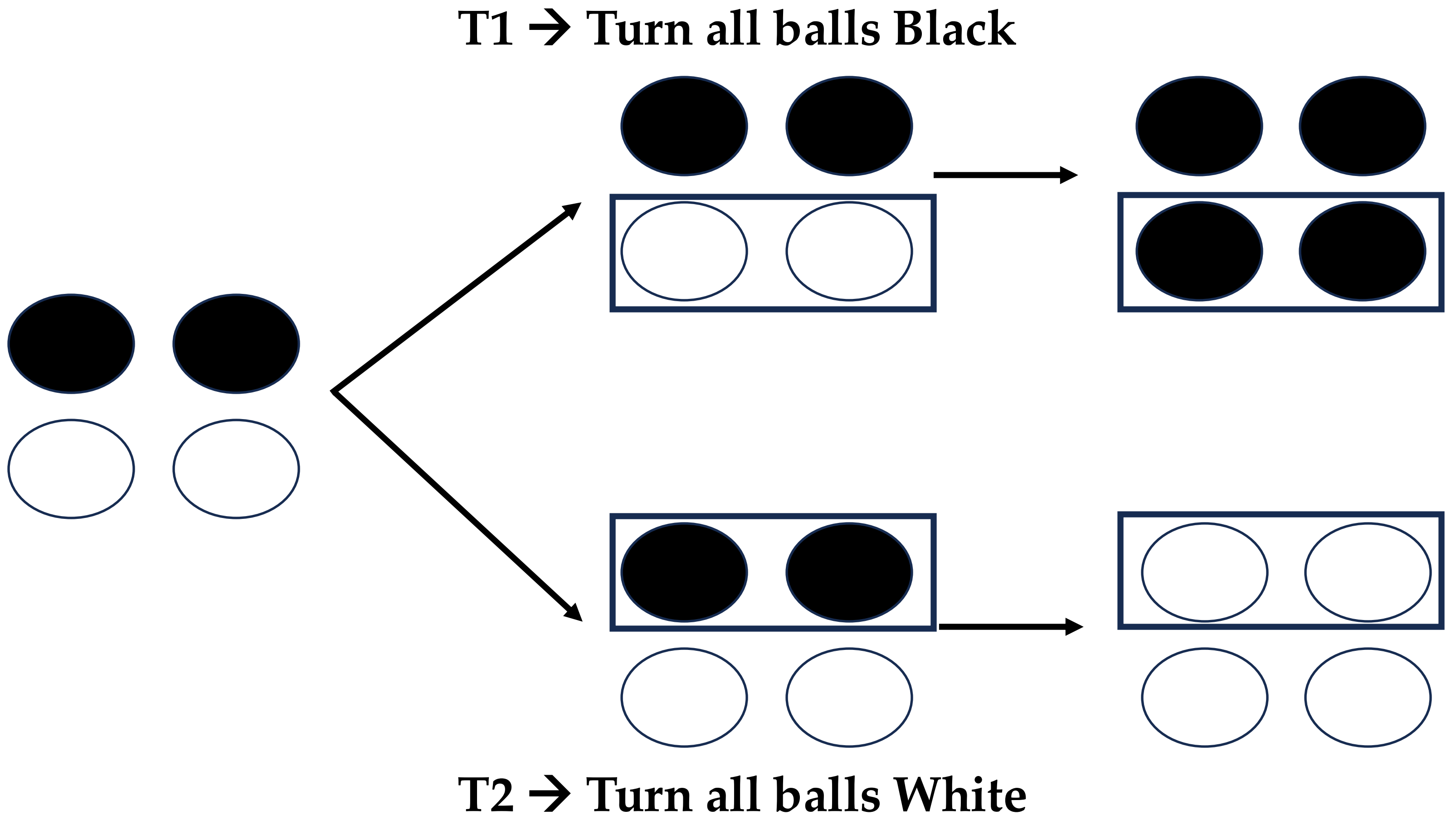
Write Skew Anomaly

T1 → Turn all balls Black



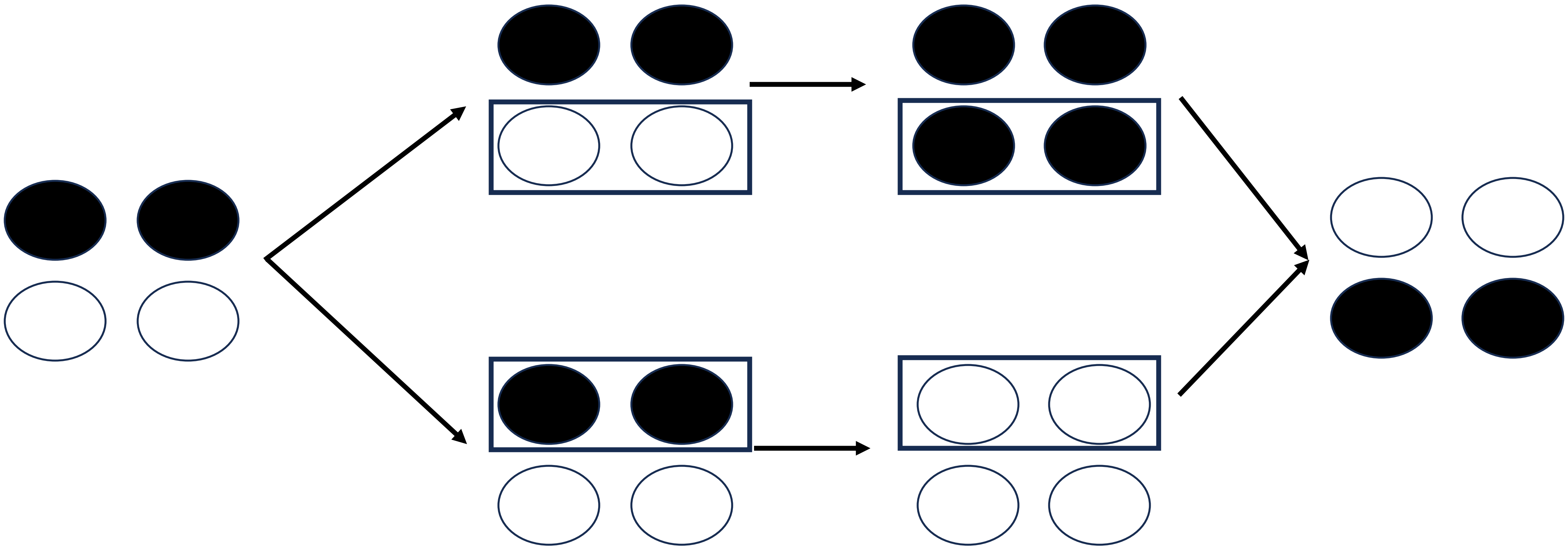
T2 → Turn all balls White

Write Skew Anomaly



Write Skew Anomaly

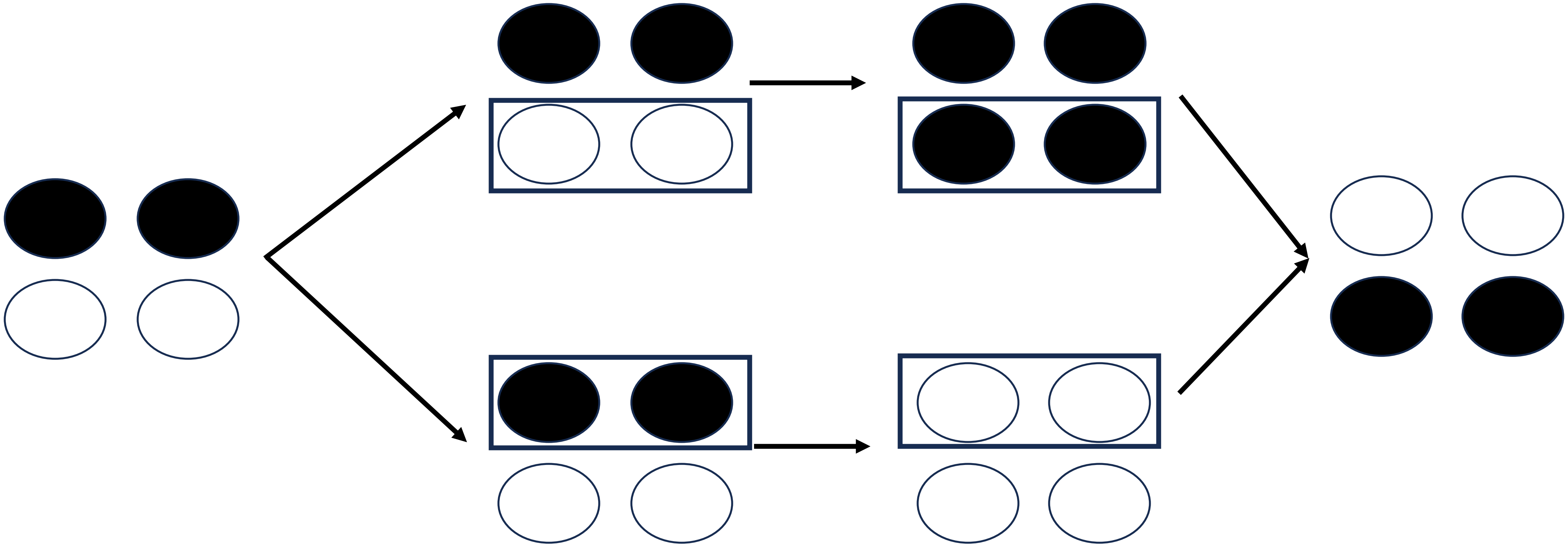
T1 → Turn all balls Black



T2 → Turn all balls White

Write Skew Anomaly

T1 → Turn all balls Black

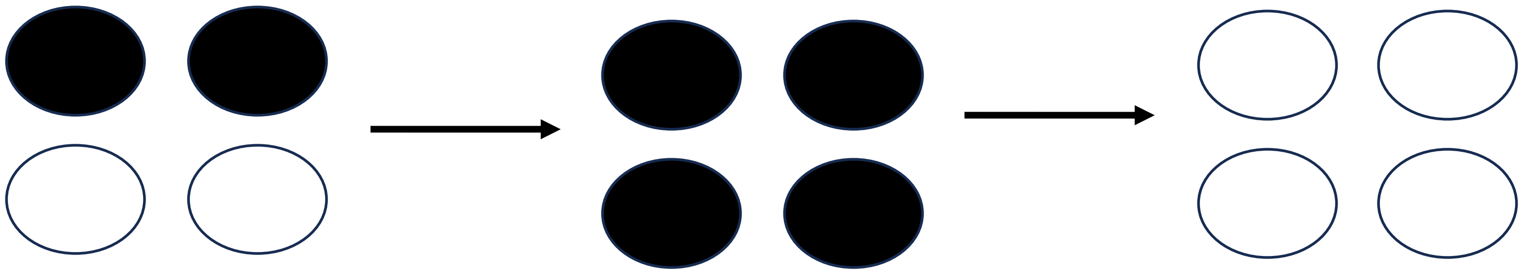


T2 → Turn all balls White

But this is not serializable.

Write Skew Anomaly

T1 → Turn all balls Black



T2 → Turn all balls White

This is serializable.

MVCC Design Decisions

- What do we need to consider while designing an MVCC scheme?

MVCC Design Decisions

- What do we need to consider while designing an MVCC scheme?

- Preventing Write Skew
- Version Storage
- Garbage Collection
- Index Management
- Deletes

Concurrency Control Protocol

- **Approach 1: Timestamp Ordering**
  - Assign transactions timestamps that determine serial order.

- **Approach 2: Optimistic Concurrency Control**
  - Three-phase protocol that we learnt in T/O lecture.
  - Use private workspace for new versions.

- **Approach 3: Two-Phase Locking**
  - Transactions acquire lock on physical version before they can read/write a logical tuple.

Version Storage

- **How to store versions?**

Version Storage

- **How to store versions?**

- The DBMS uses the record's pointer field to create a version chain per logical tuple.

- This allows the DBMS to find the version that is visible to a particular transaction at runtime.

- Indexes always point to the **head** of the chain.

- Different storage schemes determine where/what to store for each version.

Garbage Collection

- **How to garbage collect old versions?**

Garbage Collection

- **How to garbage collect old versions?**

- The DBMS needs to remove reclaimable physical versions from the database over time.

- No active transaction in the DBMS should be able to see a version going to be garbage collected.
  - For example: A version was created by an aborted transaction should be garbage collected.

- Two additional design decisions:
  - How to look for expired versions?
  - How to decide when it is safe to reclaim memory?