Database Processing
CS 451 / 551

Lecture 13:
Two-Phase Locking and Time-Stamp Ordering

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

Assignment 3 is Out!

Deadline: Nov 30, 2025 at 11:59pm

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

Syllabus → Focus on course not covered in either Quiz but you should remember indexing and storage.

Presentation

- Time slots to be released today around 11am PST.
- Each group gets a 15min time slot.
- 8 minutes to present + 7 minutes for Q/A
  - If 4 group members → 2 min for each member.
- Don't present your code.
- Present your idea.
  - How did you design?
  - Why did you select such a design?
  - Is there anything cool about your design.
  - How did each member contribute?

Last Class

- We discussed possibility of deadlocks in 2PL.
- There are two ways to manage deadlocks:
- **Deadlock Detection** → When deadlock occurs, detect and solve.
- **Deadlock Prevention** → Prevent deadlock from occurring in the first place.

Deadlock Detection

- Create a **waits-for graph**.
- Waits-for graph keep track of what locks each transaction is waiting to acquire.
- In the wait-for graph:
  - **Nodes** are transactions
  - **Add an Edge** from transaction **Ti** to **Tj** if **Ti** is waiting for Tj to release a lock.
  - The system periodically **checks for cycles** in waits- for graph and then decides **how to break it.**

Deadlock Detection

Time

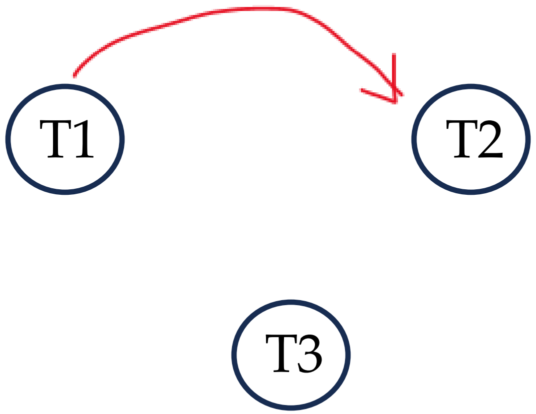
T1:	T2:	T3:
Begin	**Begin**	**Begin**
S-Lock(**A**)		
	X-Lock(**B**)	
		S-Lock(**C**)
S-Lock(**B**)		
	X-Lock(**C**)	
		X-Lock(**A**)

Three transactions and three data-items

Deadlock Detection

Time

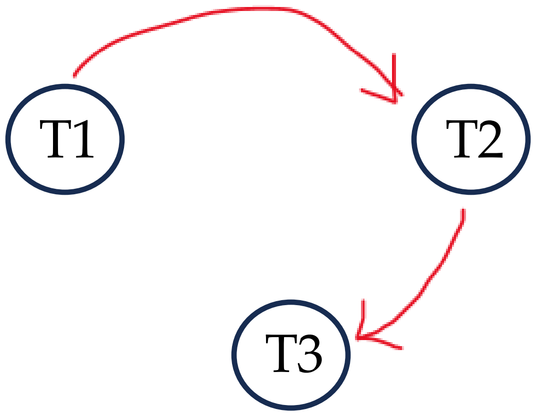
T1:	**T2:**	**T3:**
Begin	**Begin**	**Begin**
S-Lock(**A**)		
	X-Lock(**B**)	
		S-Lock(**C**)
S-Lock(**B**)		
	X-Lock(**C**)	
		X-Lock(**A**)



Deadlock Detection

Time

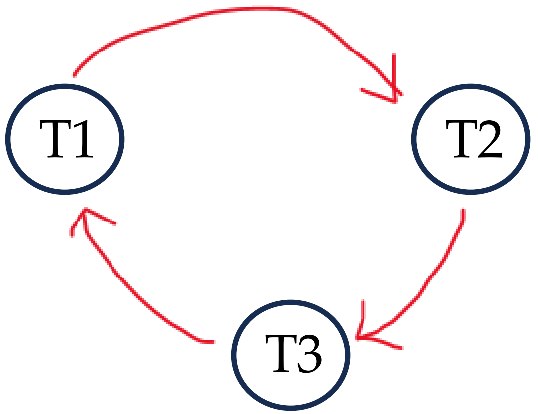
T1:	T2:	T3:
Begin	**Begin**	**Begin**
S-Lock(**A**)		
	X-Lock(**B**)	
		S-Lock(**C**)
S-Lock(**B**)		
	X-Lock(**C**)	
		X-Lock(**A**)



Deadlock Detection

Time

T1:	**T2:**	**T3:**
Begin	**Begin**	**Begin**
S-Lock(**A**)		
	X-Lock(**B**)	
		S-Lock(**C**)
S-Lock(**B**)		
	X-Lock(**C**)	
		X-Lock(**A**)



Deadlock Handling

- When the DBMS detects a deadlock, it will select a **victim**.
- The victim transaction is **rollbacked** to break the cycle.
- The victim transaction is either **restarted in the future** or **aborted**.
- Performance trade-off between the frequency of checking for deadlocks and the time transactions wait before deadlocks are broken.

Deadlock Handling: Victim Selection

- Selecting a victim depends on several factors:
  - Age (lowest timestamp).
  - Progress (least/most queries executed)
  - The number of items already locked.
  - The number of transactions that need to be rollbacked along with it.

- Note: if your DBMS plans to restart the rollbacked transaction, do take into account **starvation**.
  - The number of times a transactions has been restarted in the past.

Deadlock Prevention

Deadlock Prevention

- If a transaction **tries to acquire a lock** that is held by another transaction, **kill** one of them to prevent a deadlock.

- No need for a waits-for graph or detection algorithm.

Deadlock Prevention

- So how to achieve deadlock prevention?

Deadlock Prevention

- So how to achieve deadlock prevention?

- Assign a **timestamp** (time of arrival in the system) to each transaction.

- **Prioritize** transactions based on the value of timestamps.
    - For example: Higher timestamp, lower the priority.

Deadlock Prevention

- So how to achieve deadlock prevention?
- Assign a **timestamp** (time of arrival in the system) to each transaction.
- **Prioritize** transactions based on the value of timestamps.
  - For example: Higher timestamp, lower the priority
- Two Designs:
  - **Wait-Die**
  - **Wound-Wait**

Deadlock Prevention: Wait-Die

- **Old Waits for Young**
- If requesting transaction has higher priority than holding transaction, then requesting transaction waits for holding transaction.
- Otherwise requesting transaction aborts.

Deadlock Prevention: Wait-Die

Time

T1:	T2:	T3:
Begin	**Begin**	**Begin**
S-Lock(**A**)		
	X-Lock(**B**)	
		S-Lock(**C**)
S-Lock(**B**)		
	X-Lock(**C**)	
		X-Lock(**A**)

Three transactions and three data-items

Deadlock Prevention: Wait-Die

Time

T1:	**T2:**	**T3:**
Begin	**Begin**	**Begin**
S-Lock(**A**)		
	X-Lock(**B**)	
		S-Lock(**C**)
S-Lock(**B**)		
	X-Lock(**C**)	
		X-Lock(**A**)

T1 waits!

Deadlock Prevention: Wait-Die

Time

T1:	T2:	T3:
Begin	**Begin**	**Begin**
S-Lock(**A**)		
	X-Lock(**B**)	
		S-Lock(**C**)
S-Lock(**B**)		
	X-Lock(**C**)	
		X-Lock(**A**)

T2 waits!

Deadlock Prevention: Wait-Die

Time

T1:	**T2:**	**T3:**
Begin	**Begin**	**Begin**
S-Lock(**A**)		
	X-Lock(**B**)	
		S-Lock(**C**)
S-Lock(**B**)		
	X-Lock(**C**)	
		X-Lock(**A**)

T3 aborts, which allows T2 to finish and thus T1!

Deadlock Prevention: Wound-Wait

- **Young Waits for Old**

- If requesting transaction has higher priority than holding transaction, then holding transaction aborts and releases lock.

- Otherwise requesting transaction waits.

Deadlock Prevention: Wound-Wait

Time

T1:	T2:	T3:
Begin	**Begin**	**Begin**
S-Lock(**A**)		
	X-Lock(**B**)	
		S-Lock(**C**)
S-Lock(**B**)		
	X-Lock(**C**)	
		X-Lock(**A**)

Three transactions and three data-items

Deadlock Prevention: Wound-Wait

Time

T1:	T2:	T3:
Begin	**Begin**	**Begin**
S-Lock(**A**)		
	X-Lock(**B**)	
		S-Lock(**C**)
S-Lock(**B**)		
	X-Lock(**C**)	
		X-Lock(**A**)

T1 needs a lock held by T2, so we abort T2.

This allows T1 to finish and release locks later.

Deadlock Prevention: Wound-Wait

Time

T1:	**T2:**	**T3:**
Begin	**Begin**	**Begin**
S-Lock(**A**)		
	X-Lock(**B**)	
		S-Lock(**C**)
S-Lock(**B**)		
	X-Lock(**C**)	
		X-Lock(**A**)

T3 waits for T1 as young waits for old.

Deadlock Prevention

- **Why do these schemes guarantee no deadlocks?**

Deadlock Prevention

- **Why do these schemes guarantee no deadlocks?**
- Ensure that waiting for locks occur in only one.
- **When a transaction restarts, what is its priority?**

Deadlock Prevention

- **Why do these schemes guarantee no deadlocks?**
- Ensure that waiting for locks occur in only one.
- **When a transaction restarts, what is its priority?**
- Its original timestamp to prevent the transaction from starving.

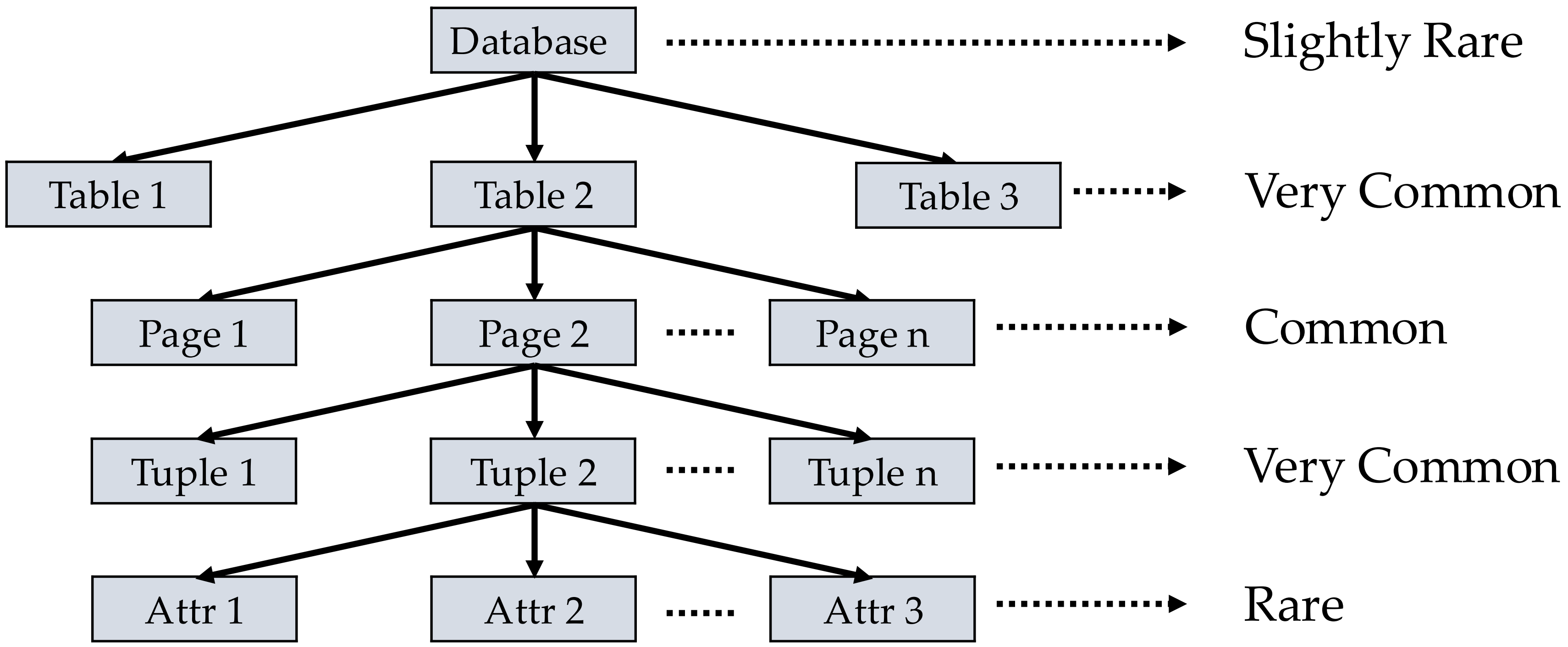
Lock Granularities

- What is the right granularity of acquiring a lock?

Lock Granularities

- What is the right granularity of acquiring a lock?
- The DBMS needs to decide the lock granularity: page, tuple, or attribute?
- Finer the lock granularity, better the performance and harder to guarantee code correctness.
- Finer the lock granularity, frequent the need to request/acquire locks.

Lock Granularities



Lock Granularities

Say a transaction T1 has locked some attribute in Table 2/ Page 1 and another transaction T2 wants to lock the full database, how to check if T2's request can / cannot be satisfied?

Intention Locks

- An **intention lock** allows locking a higher-level node in shared or exclusive mode without checking all the descendent nodes.

- If a node is locked in an **intention mode**, then some transaction has acquired a lock at the lower level in the tree.

Intention Locks

- **Intention-Shared (IS)**
  - Indicates explicit locking at lower level with S-Locks.
  - Intent to get S-Lock(s) at finer granularity.

- **Intention-Exclusive (IX)**
  - Indicates explicit locking at lower level with X-Locks.
  - Intent to get X-Lock(s) at finer granularity.

- **Shared+Intention-Exclusive (SIX)**
  - The subtree rooted by that node is locked explicitly in S mode and at a further lower level explicit locking with X-Locks.

Lock Compatibility Matrix

If a transaction Ti holds a lock, can another transaction Tj acquire a lock.

Tj Wants

Ti Holds		IS	IX	S	SIX	X
	IS	✓	✓	✓	✓	✗
	IX	✓	✓	✗	✗	✗
	S	✓	✗	✓	✗	✗
	SIX	✓	✗	✗	✗	✗
	X	✗	✗	✗	✗	✗

Locking Protocol

- A transaction tries to fetch an appropriate lock at highest level of the database hierarchy.

- To get **S** or **IS** lock on a node, the transaction must have at least **IS** lock on the parent node.

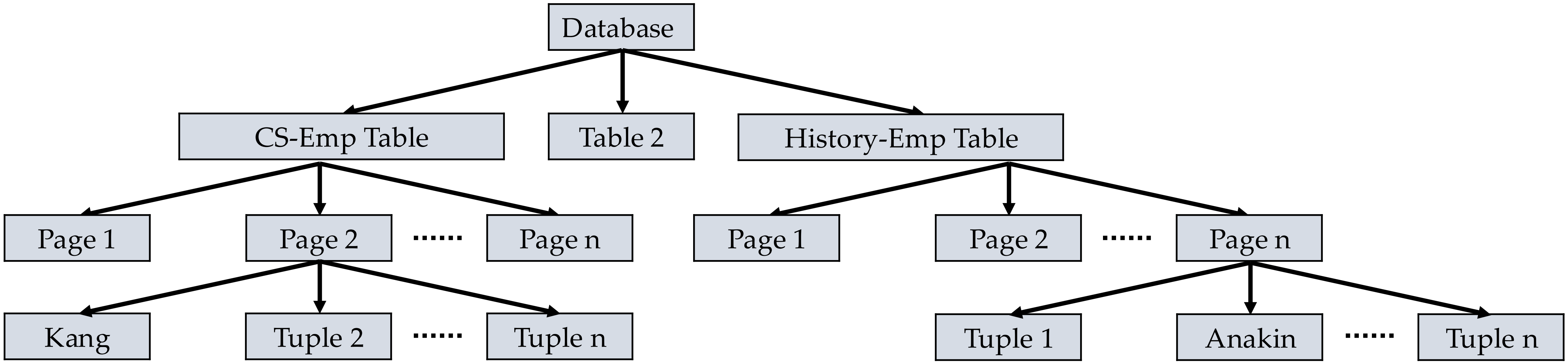
- To get **X**, **IX**, or **SIX** on a node, the transaction must have at least **IX** lock on the parent node.

Example 1

- T1 → Read a CS employee Kang's salary.
- T2 → Increase a History employee Anakin's salary by $100.

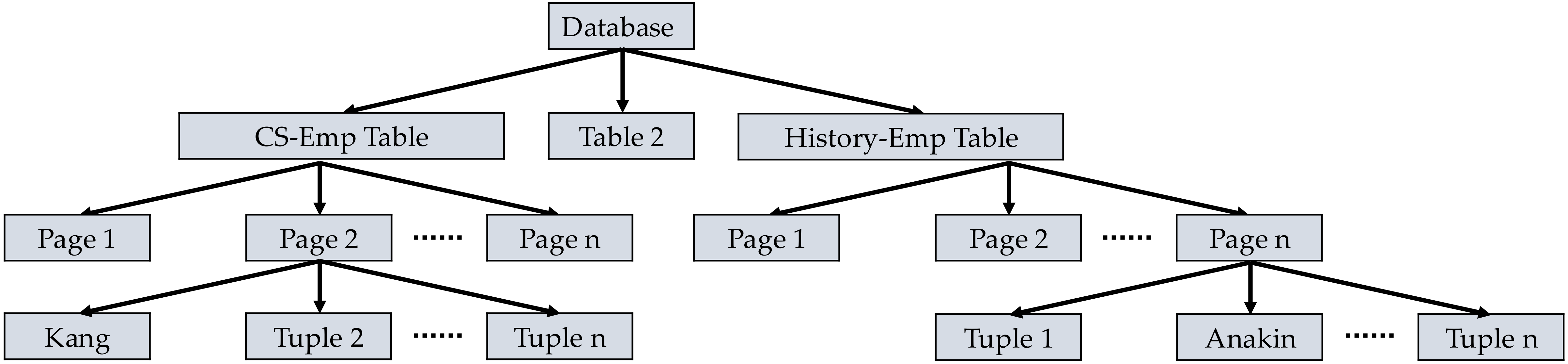
- How can we acquire locks in this example?

Example 1


Database
CS-Emp Table
Table 2
History-Emp Table
Page 1
Page 2
......
Page n
Page 1
Page 2
......
Page n
Kang
Tuple 2
......
Tuple n
Tuple 1
Anakin
......
Tuple n

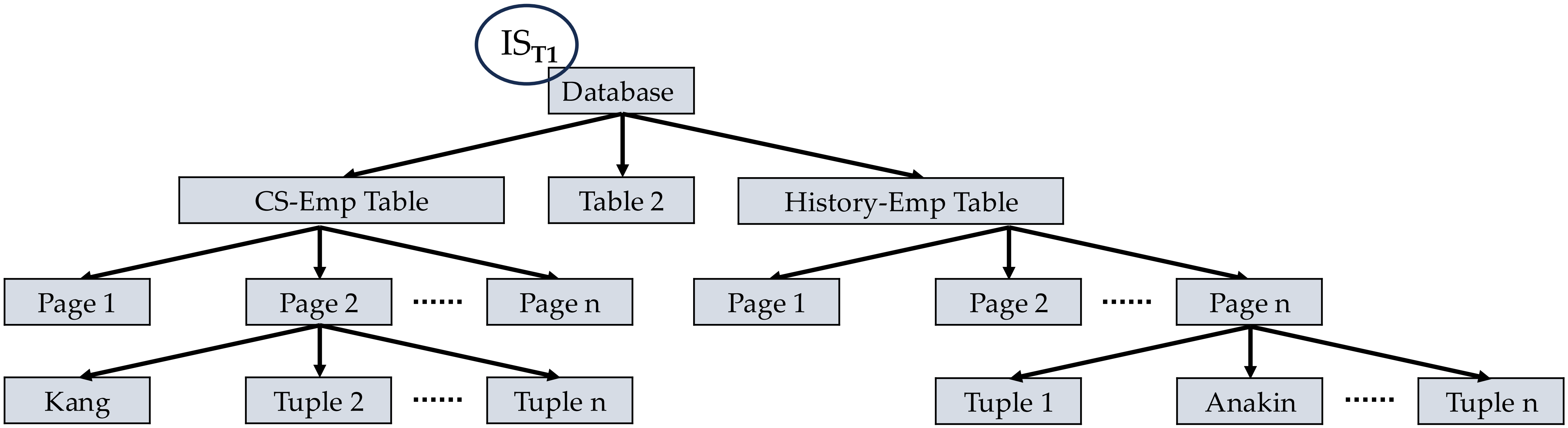
Example 1

T1 starts locking



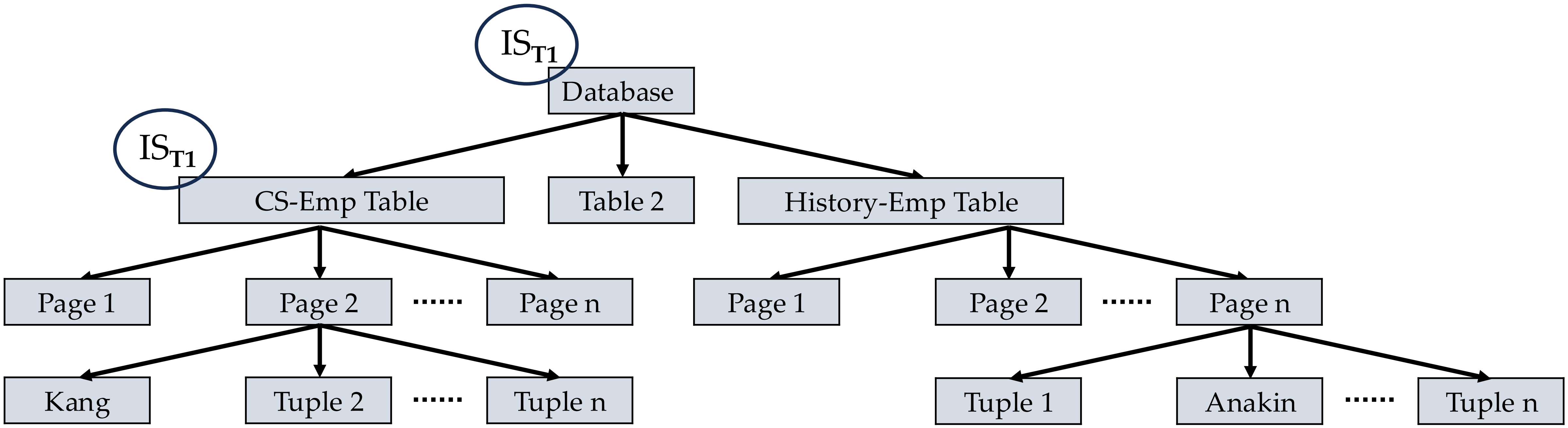
Example 1

T1 starts locking



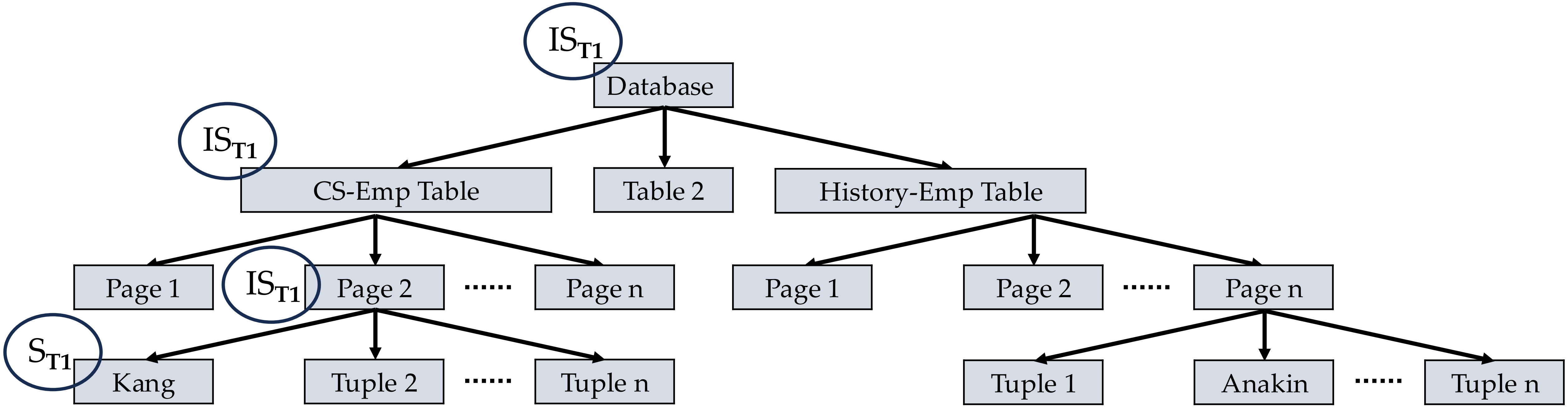
Example 1

T1 starts locking



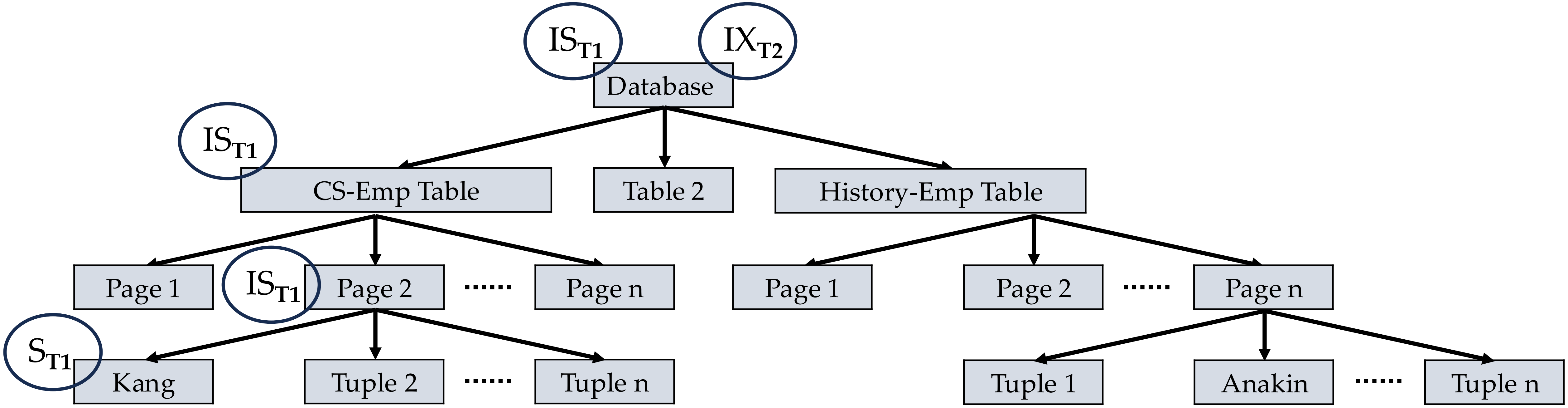
Example 1

T1 starts locking



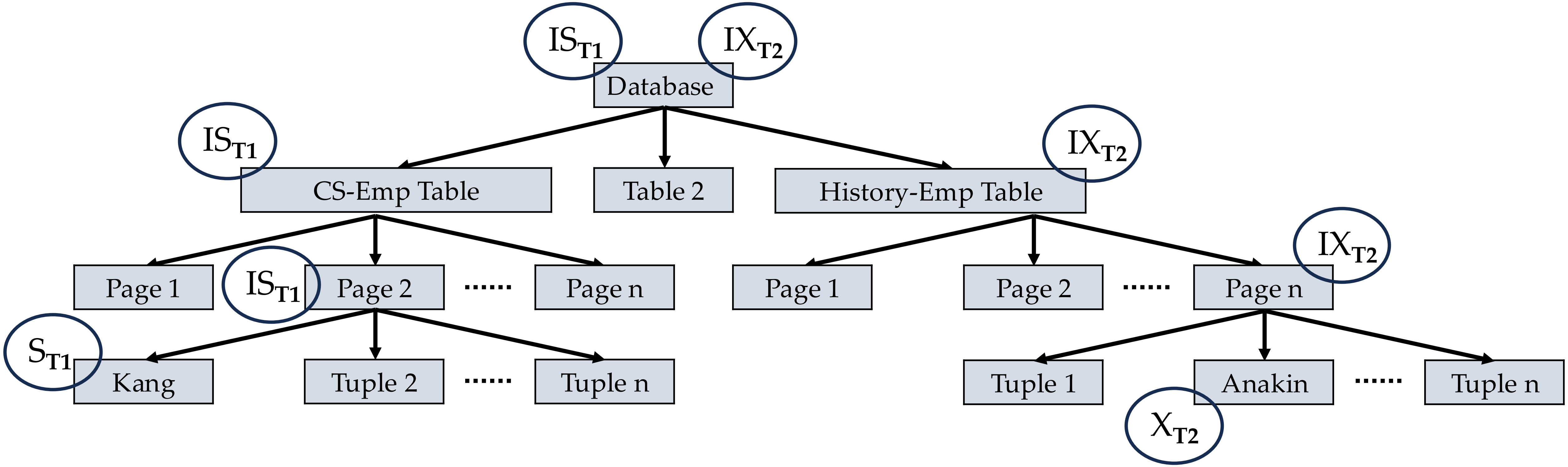
Example 1

T2 starts locking and IS and IX are compatible



Example 1

T2 starts locking

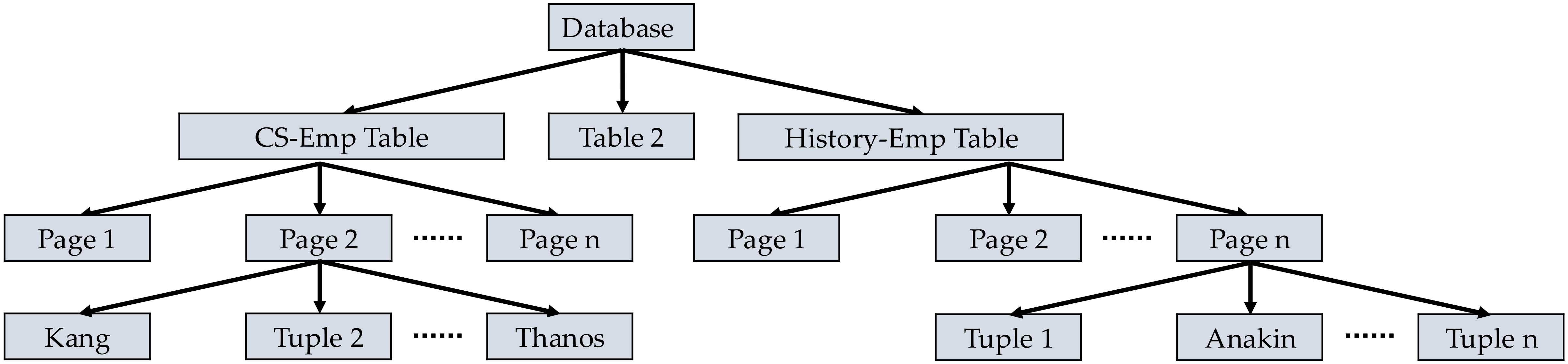


Example 2

- T1 → Read all CS employees salary and increase Kang's salary by $100.
- T2 → Read salary of CS employee Thanos.
- T3 → Read all CS employees salary .

- How can we acquire locks in this example?

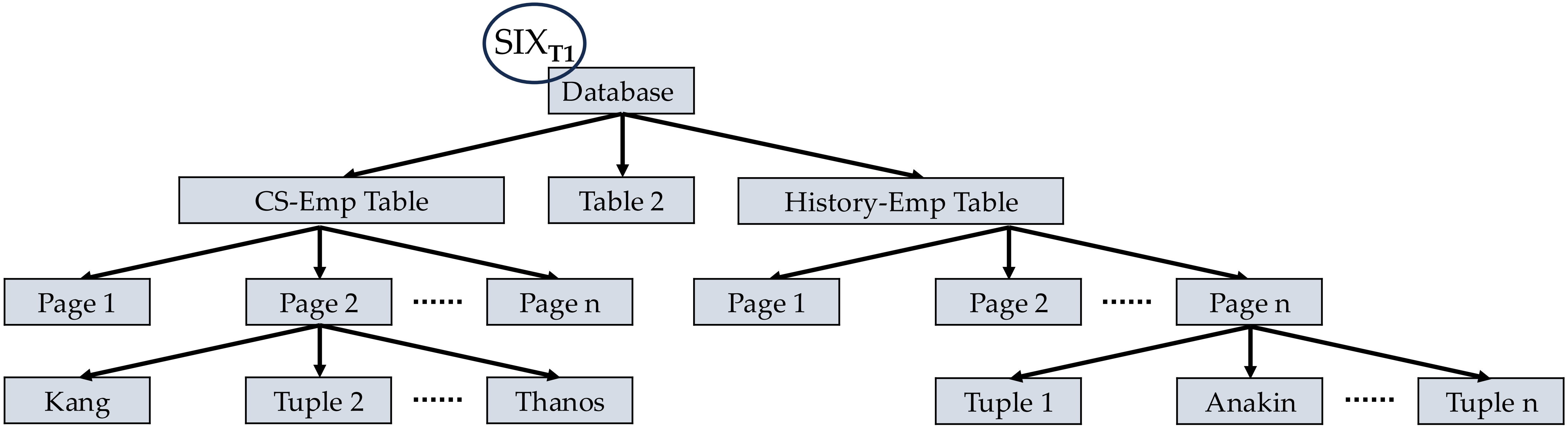
Example 2



Database
CS-Emp Table
Table 2
History-Emp Table
Page 1
Page 2
......
Page n
Page 1
Page 2
......
Page n
Kang
Tuple 2
......
Thanos
Tuple 1
Anakin
......
Tuple n

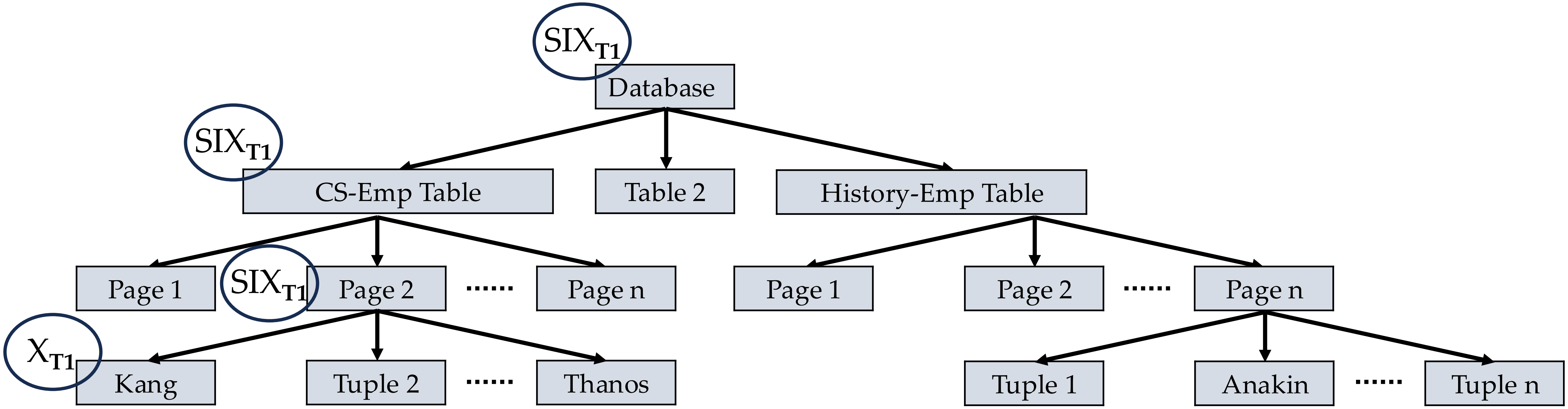
Example 2

T1 starts locking



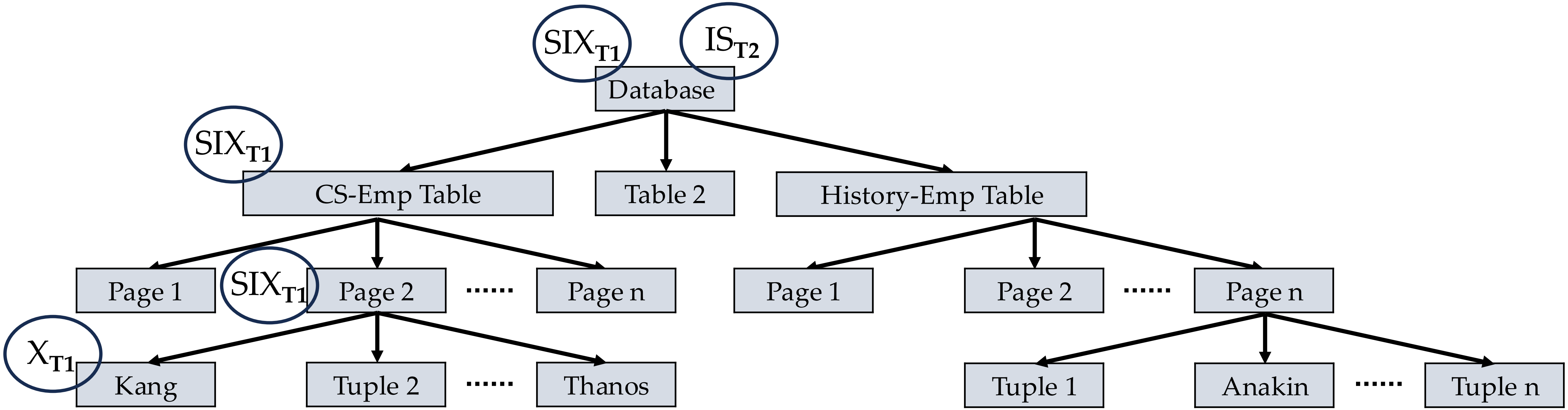
Example 2

T1 starts locking



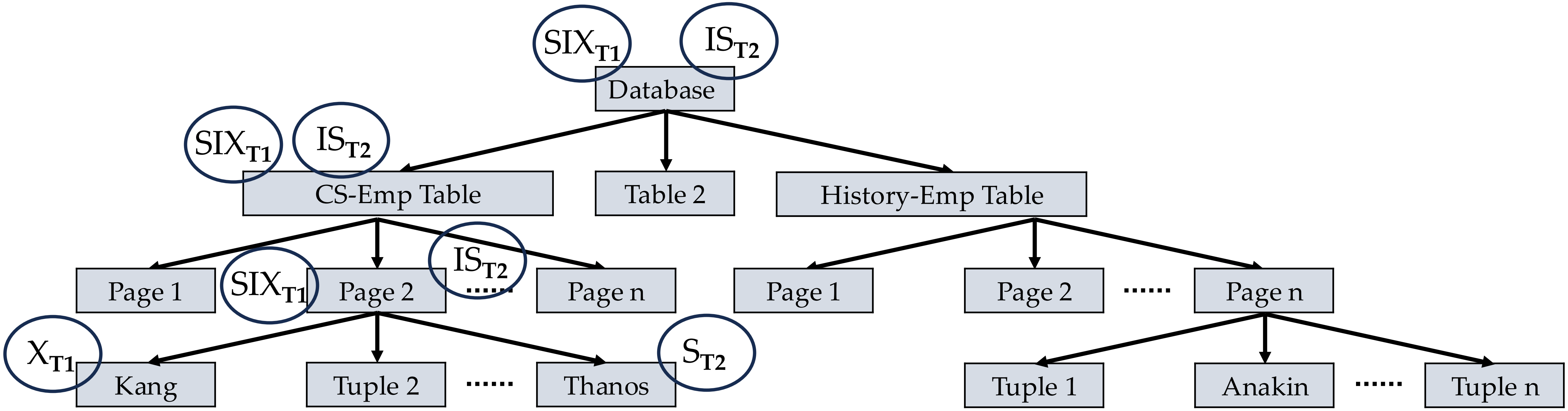
Example 2

T2 starts locking



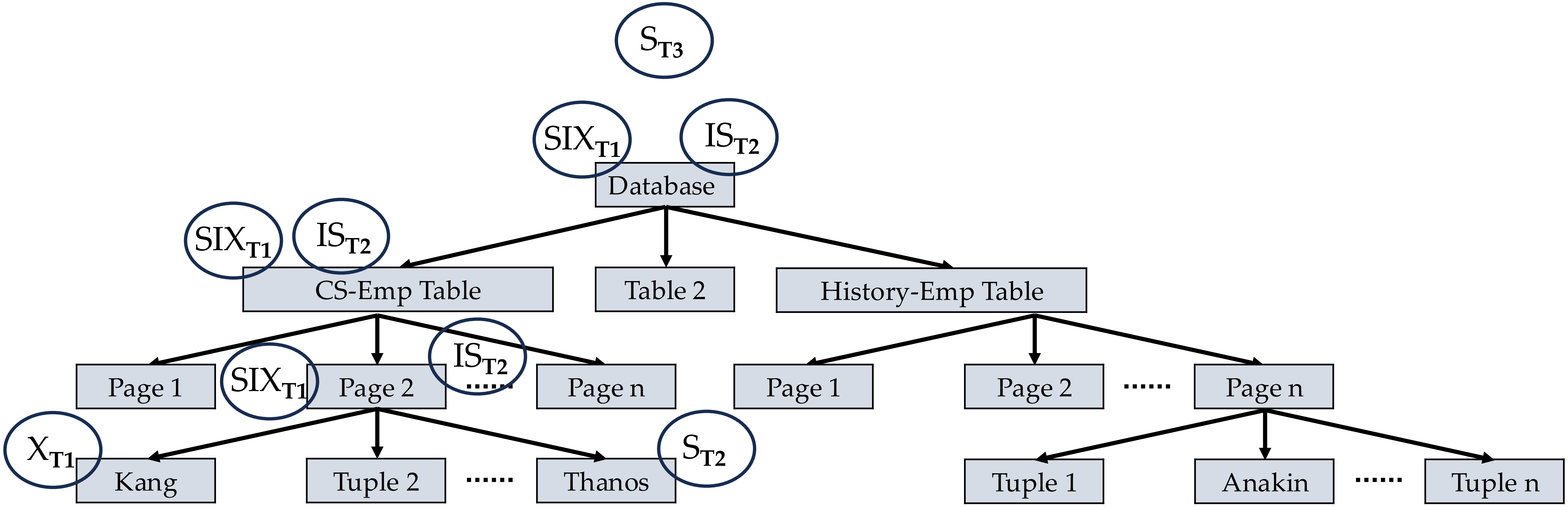
Example 2

T2 starts locking



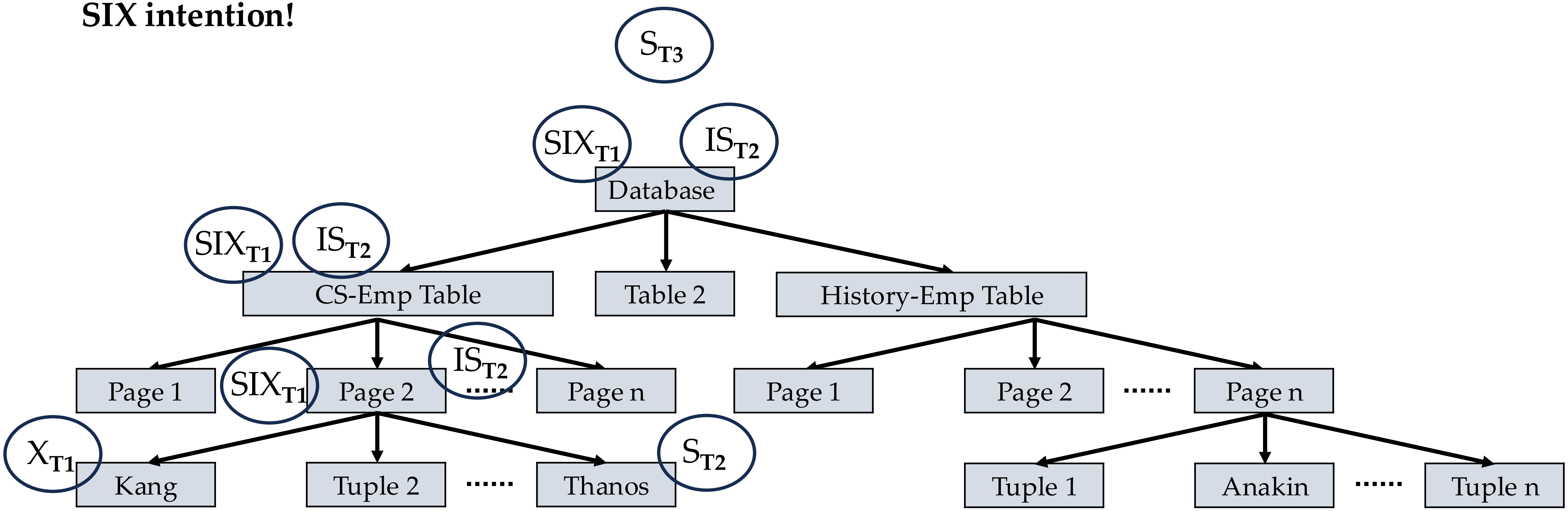
Example 2

T3 starts locking



Example 2

T3 is not allowed to lock until T1 finishes because T3 wants to read something which has SIX intention!



Lock Escalation

- The DBMS automatically switches to coarser-grained locks when a transaction acquires too many finer-grained locks.

- Reduces the number of requests that the lock manager needs to process.

Discussion

- 2PL forces transactions to acquire locks!
- Strong Strict 2PL forces transactions to acquire locks early to prevent cascade aborts!
- These protocols take a pessimistic approach and assume that conflicts are common and transactions access a lot of data items!
- Can we do better?

Timestamp Ordering Concurrency Control

Timestamp Ordering Concurrency Control

- An optimistic concurrency control protocol.

- Assumption:
  - Conflicts between transactions are rare.
  - Transactions are short-lived.

- Optimized for the no-conflict cases.

Assigning Timestamps

- Each transaction Ti is assigned a **unique monotonically increasing** timestamp.
- Let **TS(Ti)** be the timestamp allocated to transaction **Ti**.
- When to assign the timestamp → Depends on the design.
- How to generate a timestamp?

Assigning Timestamps

- Each transaction Ti is assigned a **unique monotonically increasing** timestamp.

- Let **TS(Ti)** be the timestamp allocated to transaction **Ti**.

- When to assign the timestamp → Depends on the design.

- How to generate a timestamp?
  - Wall clock time / System time
  - Logical counter
  - Hybrid

Timestamp Ordering Concurrency Control

- Timestamps are used to determine the **serializability order of transactions**.

- For two transactions Ti and Tj, if TS(Ti) < TS(Tj), then
  - The DBMS must ensure that the execution schedule for these transactions is equivalent to the serial schedule where Ti appears before Tj .

- Each database object (e.g., tuple) need to track the timestamps of that last accessed/modified them.

Optimistic Concurrency Control

- Timestamp Ordering (**T/O**) can be used to design an OCC protocol.

- In OCC using T/O, DBMS creates a **private workspace** for each transaction.

Optimistic Concurrency Control

- Timestamp Ordering (**T/O**) can be used to design an OCC protocol.

- In OCC using T/O, DBMS creates a **private workspace** for each transaction.
  - Each object **read** is copied into workspace.

Optimistic Concurrency Control

- Timestamp Ordering (**T/O**) can be used to design an OCC protocol.

- In OCC using T/O, DBMS creates a **private workspace** for each transaction.
  - Each object **read** is copied into workspace.
  - Updates/Writes are applied to workspace.

Optimistic Concurrency Control

- Timestamp Ordering (**T/O**) can be used to design an OCC protocol.

- In OCC using T/O, DBMS creates a **private workspace** for each transaction.
  - Each object **read** is copied into workspace.
  - Updates/Writes are applied to workspace.
  - When a transaction commits, the DBMS checks if the workspace writes conflict with other transactions.
  - If there are no conflicts, the workspace write set is copied to the database.

OCC Phases

- How does OCC work?

OCC Phases

- How does OCC work?

- **Three Phases of OCC:**
  - Read Phase
  - Validation Phase
  - Write Phase

Read Phase

- Track the read/write sets of each transaction and store the writes of each transaction in a private workspace.
- DBMS copies every tuple that the transaction accesses from the database to its private workspace.

Validation Phase

- Assign the transaction a unique timestamp (TS) and then check whether it conflicts with other transactions.

Write Phase

- If validation is successful, set the write timestamp (W-TS) for all the modified objects in private workspace to the validation timestamp.
  - Next, update the value and timestamp in the database.

- Otherwise abort transaction.

OCC Example I

Database

Object	Value	W-TS
A	100	0

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

OCC Example I

T1:	T2:
Begin	
Read	
read(**A**)	**Begin**
	Read
	read(**A**)
	Validate
	Write
	Commit
write(**A**)	
read(**A**)	
Validate	
Write	
Commit	

Time

Database

Object	Value	W-TS
A	100	0

T1 Workspace

Object	Value	W-TS

OCC Example I

Time

T1:	T2:
Begin	
Read	
read(**A**)	
	Begin
	Read
	read(**A**)
	Validate
	Write
	Commit
write(**A**)	
read(**A**)	
Validate	
Write	
Commit	

Database

Object	Value	W-TS
A	100	0

T1 Workspace

Object	Value	W-TS
A	100	0

OCC Example I

Time	T1:	T2:
	Begin	
	Read	
	read(A)	→ Begin
		Read
		read(A)
		Validate
		Write
		Commit
	write(A)	
	read(A)	
	Validate	
	Write	
	Commit	

Database

Object	Value	W-TS
A	100	0

T1 Workspace

Object	Value	W-TS
A	100	0

T2 Workspace

Object	Value	W-TS

OCC Example I

Time

T1:	T2:
Begin	
Read	
read(**A**)	
	Begin
	Read
	read(**A**)
	Validate
	Write
	Commit
write(**A**)	
read(**A**)	
Validate	
Write	
Commit	

Database

Object	Value	W-TS
A	100	0

T1 Workspace

Object	Value	W-TS
A	100	0

T2 Workspace

Object	Value	W-TS
A	100	0

OCC Example I

T1:	T2:
Begin	
Read	
read(**A**)	
	Begin
	Read
	read(**A**)
	Validate
	Write
	Commit
write(**A**)	
read(**A**)	
Validate	
Write	
Commit	

Time

Database

Object	Value	W-TS
A	100	0

T1 Workspace

Object	Value	W-TS
A	100	0

T2 Workspace

Object	Value	W-TS
A	100	0

Timestamp for T2:
TS(T2) = 1

OCC Example I

T1:	T2:
Begin	
Read	
read(**A**)	
	Begin
	Read
	read(**A**)
	Validate
	→ **Write**
	Commit
write(**A**)	
read(**A**)	
Validate	
Write	
Commit	

Time

Database

Object	Value	W-TS
A	100	0

T1 Workspace

Object	Value	W-TS
A	100	0

T2 Workspace

Object	Value	W-TS
A	100	0

Nothing written so no change to timestamp.

OCC Example I

Time

T1:	**T2:**
Begin	
Read	
read(**A**)	**Begin**
	Read
	read(**A**)
	Validate
	Write
	Commit
write(**A**)	
read(**A**)	
Validate	
Write	
Commit	

Database

Object	Value	W-TS
A	100	0

T1 Workspace

Object	Value	W-TS
A	100	0

No update to global database.

OCC Example I

Database

Object	Value	W-TS
A	100	0

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

T1 Workspace

Object	Value	W-TS
A	150	0

Update in value.

OCC Example I

T1:	T2:
Begin	
Read	
read(**A**)	**Begin**
	Read
	read(**A**)
	Validate
	Write
	Commit
write(**A**)	
read(**A**)	
Validate	
Write	
Commit	

Time

Database

Object	Value	W-TS
A	100	0

T1 Workspace

Object	Value	W-TS
A	150	0

OCC Example I

Database

Object	Value	W-TS
A	100	0

Time	T1:	T2:
	Begin	
	Read	
	read(**A**)	**Begin**
		Read
		read(**A**)
		Validate
		Write
		Commit
	write(**A**)	
	read(**A**)	
→	**Validate**	
	Write	
	Commit	

T1 Workspace

Object	Value	W-TS
A	150	0

Timestamp for T1:
TS(T1) = 2

OCC Example I

Time

T1:	T2:
Begin	
Read	
read(**A**)	
	Begin
	Read
	read(**A**)
	Validate
	Write
	Commit
write(**A**)	
read(**A**)	
Validate	
Write	
Commit	

Database

Object	Value	W-TS
A	100	0

T1 Workspace

Object	Value	W-TS
A	150	2

Write-Timestamp set for data-item A.

OCC Example I

T1:	T2:
Begin	
Read	
read(**A**)	**Begin**
	Read
	read(**A**)
	Validate
	Write
	Commit
write(**A**)	
read(**A**)	
Validate	
Write	
Commit	

Time

Database

Object	Value	W-TS
A	150	2

T1 Workspace

Object	Value	W-TS
A	150	2

No conflicts so updates written to global database.

Validation Phase

- Assign the transaction a unique timestamp (TS) and then check whether it conflicts with other transactions.
- When transaction Ti invokes **Commit**, the DBMS checks if it conflicts with other transactions.
- Simplest mechanism → Use serial validation.
- How can DBMS guarantee only serializable schedules are permitted?

Validation Phase

- Assign the transaction a unique timestamp (TS) and then check whether it conflicts with other transactions.
- When transaction Ti invokes **Commit**, the DBMS checks if it conflicts with other transactions.
- Simplest mechanism → Use serial validation.
- How can DBMS guarantee only serializable schedules are permitted?
  - Forward Validation
  - Backward Validation

Forward Validation

- At the time of commit, each transaction checks if it conflicts with other **concurrently ongoing transactions** (yet to be committed).

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

- There are three specific cases to satisfy:

Forward Validation: Case I

- For two transactions T1 and T2, say T1 is at the validation step (T1 < T2)
  - Check if T1 completes its Write phase before T2 begins its Read phase.
  - No conflict as all T1 's actions happen before T2 ‘s.
  - Essentially, serial ordering.

Forward Validation: Case I

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

Forward Validation: Case II

- For two transactions T1 and T2, say T1 is at the validation step (T1 < T2)
    - Check if T1 completes its Write phase before T2 starts its Write phase.
    - T1 does not modify to any object read by T2.
    - **WriteSet(T1) ⋂ ReadSet(T2) = 0**

Forward Validation: Case II

T1:	T2:
Begin	
Read	
read(**A**)	
write(**A**)	**Begin**
	Read
	read(**A**)
Validate	
	Validate
	Write
	Commit

Time

Database

Object	Value	W-TS
A	100	0

Forward Validation: Case II

T1:	T2:
Begin	
Read	
read(**A**)	
write(**A**)	**Begin**
	Read
	read(**A**)
Validate	
	Validate
	Write
	Commit

Time

Database

Object	Value	W-TS
A	100	0

T1 Workspace

Object	Value	W-TS

Forward Validation: Case II

Time

T1:	T2:
Begin	
Read	
read(**A**)	
write(**A**)	**Begin**
	Read
	read(**A**)
Validate	
	Validate
	Write
	Commit

Database

Object	Value	W-TS
A	100	0

T1 Workspace

Object	Value	W-TS
A	100	0

Forward Validation: Case II

T1:	T2:
Begin	
Read	
read(**A**)	
write(**A**)	Begin
	Read
	read(**A**)
Validate	
	Validate
	Write
	Commit

Time

Database

Object	Value	W-TS
A	100	0

T1 Workspace

Object	Value	W-TS
A	150	0

Forward Validation: Case II

Time

T1:	T2:
Begin	
Read	
read(**A**)	
write(**A**)	**Begin**
	Read
	read(**A**)
Validate	
	Validate
	Write
	Commit

Database

Object	Value	W-TS
A	100	0

T1 Workspace

Object	Value	W-TS
A	150	0

T2 Workspace

Object	Value	W-TS

Forward Validation: Case II

T1:	T2:
Begin	
Read	
read(**A**)	
write(**A**)	**Begin**
	Read
	→ read(**A**)
Validate	
	Validate
	Write
	Commit

Time

Database

Object	Value	W-TS
A	100	0

T1 Workspace

Object	Value	W-TS
A	150	0

T2 Workspace

Object	Value	W-TS
A	100	0

Forward Validation: Case II

Database

Object	Value	W-TS
A	100	0

Time

T1:	T2:
Begin	
Read	
read(**A**)	
write(**A**)	**Begin**
	Read
	read(**A**)
Validate	
	Validate
	Write
	Commit

T1 Workspace

Object	Value	W-TS
A	150	0

T2 Workspace

Object	Value	W-TS
A	100	0

T1 has to be aborted, fails Case II condition.

Forward Validation: Case II

How about this example?

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

Database

Object	Value	W-TS
A	100	0

Forward Validation: Case II

Time

T1:	T2:
Begin	
Read	
read(**A**)	
write(**A**)	**Begin**
	Read
	read(**A**)
	Validate
Validate	
Write	
Commit	**Write**
	Commit

Database

Object	Value	W-TS
A	100	0

T1 Workspace

Object	Value	W-TS

Forward Validation: Case II

Time

T1:	T2:
Begin	
Read	
read(**A**)	
write(**A**)	**Begin**
	Read
	read(**A**)
	Validate
Validate	
Write	
Commit	**Write**
	Commit

Database

Object	Value	W-TS
A	100	0

T1 Workspace

Object	Value	W-TS
A	100	0

Forward Validation: Case II

T1:	T2:
Begin	
Read	
read(**A**)	
write(**A**)	Begin
	Read
	read(**A**)
	Validate
Validate	
Write	
Commit	Write
	Commit

Time

Database

Object	Value	W-TS
A	100	0

T1 Workspace

Object	Value	W-TS
A	150	0

Forward Validation: Case II

T1:	T2:
Begin	
Read	
read(**A**)	
write(**A**)	Begin
	Read
	read(**A**)
	Validate
Validate	
Write	
Commit	Write
	Commit

Time

Database

Object	Value	W-TS
A	100	0

T1 Workspace

Object	Value	W-TS
A	150	0

T2 Workspace

Object	Value	W-TS

Forward Validation: Case II

T1:	T2:
Begin	
Read	
read(**A**)	
write(**A**)	**Begin**
	Read
	read(**A**)
	Validate
Validate	
Write	
Commit	**Write**
	Commit

Time

Database

Object	Value	W-TS
A	100	0

T1 Workspace

Object	Value	W-TS
A	150	0

T2 Workspace

Object	Value	W-TS
A	100	0

Forward Validation: Case II

Time

T1:	T2:
Begin	
Read	
read(A)	
write(A)	Begin
	Read
	read(A)
	Validate
Validate	
Write	
Commit	Write
	Commit

Database

Object	Value	W-TS
A	100	0

T1 Workspace

Object	Value	W-TS
A	150	0

T2 Workspace

Object	Value	W-TS
A	100	0

Safe to commit and finish T2 as T2 completes before T1.

Forward Validation: Case II

Time

T1:	T2:
Begin	
Read	
read(**A**)	
write(**A**)	Begin
	Read
	read(**A**)
	Validate
Validate	
Write	
Commit	Write
	Commit

Database

Object	Value	W-TS
A	100	0

T1 Workspace

Object	Value	W-TS
A	150	0

T2 Workspace

Object	Value	W-TS
A	100	0

Safe to commit and finish T1 as T2 has committed and it can be observed as logically finishing before T1!

Forward Validation: Case III

- For two transactions T1 and T2, say T1 is at the validation step (T1 < T2)
  - Check if T1 completes its Read phase before T2 completes its Read phase.
  - T1 should not modify any object read or written by T2.
  - **WriteSet(T1) ⋂ ReadSet(T2) = 0**
  - **WriteSet(T1) ⋂ WriteSet(T2) = 0**

Forward Validation: Case III

T1:	T2:
Begin	
Read	
read(**A**)	Begin
write(**A**)	Read
	read(**B**)
Validate	
Write	
Commit	read(**A**)
	Validate
	Write
	Commit

Time

Database

Object	Value	W-TS
A	100	0
B	200	0

Forward Validation: Case III

Time

T1:	T2:
Begin	
Read	
read(**A**)	**Begin**
write(**A**)	**Read**
	read(**B**)
Validate	
Write	
Commit	read(**A**)
	Validate
	Write
	Commit

Database

Object	Value	W-TS
A	100	0
B	200	0

T1 Workspace

Object	Value	W-TS
A	100	0

Forward Validation: Case III

Time	T1:	T2:
	Begin	
	Read	
	read(**A**)	**Begin**
	write(**A**)	**Read**
		read(**B**)
	Validate	
	Write	
	Commit	read(**A**)
		Validate
		Write
		Commit

Database

Object	Value	W-TS
A	100	0
B	200	0

T1 Workspace

Object	Value	W-TS
A	150	0

Forward Validation: Case III

Time

T1:	T2:
Begin	
Read	
read(**A**)	Begin
write(**A**)	Read
	read(**B**)
Validate	
Write	
Commit	read(**A**)
	Validate
	Write
	Commit

Database

Object	Value	W-TS
A	100	0
B	200	0

T1 Workspace

Object	Value	W-TS
A	150	0

T2 Workspace

Object	Value	W-TS
B	200	0

Forward Validation: Case III

Time

T1:	T2:
Begin	
Read	
read(**A**)	**Begin**
write(**A**)	**Read**
	read(**B**)
Validate	
Write	
Commit	read(**A**)
	Validate
	Write
	Commit

Database

Object	Value	W-TS
A	150	0
B	200	0

T1 Workspace

Object	Value	W-TS
A	150	0

T2 Workspace

Object	Value	W-TS
B	200	0

Forward Validation: Case III

Time

T1:	T2:
Begin	
Read	
read(A)	Begin
write(A)	Read
	read(B)
Validate	
Write	
Commit	read(A)
	Validate
	Write
	Commit

Database

Object	Value	W-TS
A	150	0
B	200	0

T1 Workspace

Object	Value	W-TS
A	150	0

T2 Workspace

Object	Value	W-TS
B	200	0
A	150	0

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.

- There are three specific cases to satisfy:

OCC: Write Phase

- Propagate the changes in the transaction's private workspace (write set) to the database.
- The idea is to make the transaction's write-set visible to other transactions.
- **Serial Commits**: Use a global lock to limit a single transaction to be in the Validation/Write phases at a time.

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.