



UNIVERSITY OF COPENHAGEN

Concurrency Control: Two-phase locking (2PL), Introduction to Schedules and Serializability

ACS, Dmitriy Traytel

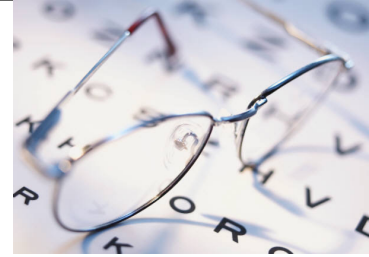
Do-it-yourself Recap: Techniques for Performance

- What is the meaning of the following performance metrics: throughput, latency, overhead, utilization, capacity?
- Why can concurrency improve throughput and latency? How does that related to modern hardware characteristics?

latency
throughput
scalability
overhead
utilization
capacity



What should we learn today?



- Identify the multiple interpretations of the property of atomicity
- Implement methods to ensure before-or-after atomicity, and argue for their correctness
- Explain the variants of the two-phase locking (2PL) protocol, in particular the widely-used Strict 2PL
- Explain situations where predicate locking is required
- Discuss the definition of serializability and the notion of anomalies

Read-Write Systems

- On-Line Transaction Processing (**OLTP**)
 - Process multiple, but relatively simple, application functions
- **Examples**
 - Order processing, e.g., Amazon
 - Item buy/sell in computer games, e.g., EVE Online
 - High-performance trading
 - Updates on social networks, e.g., Twitter





Abstractions vs. Performance is the central trade-off

- Last week's property:
Strong Modularity
- This week:
Atomicity (before-or-after flavor)



Transaction



- Reliable unit of work against memory abstraction
 - In the next lectures, we will use “memory state” and “database” interchangeably!
- **ACID Properties**
 - **Atomicity**: transactions are all-or-nothing
 - **Consistency**: transaction takes database from one consistent state to another 
 - **Isolation**: transaction executes as if it were the only one in the system (aka before-or-after atomicity) 
 - **Durability**: once transaction is done (“committed”), results are persistent in the database

Examples of Transactions in SQL

Transaction T1: TRANSFER



```
BEGIN
  UPDATE account
  SET bal = bal + 100
  WHERE account_id = 'A';
  --
  UPDATE account
  SET bal = bal - 100
  WHERE account_id = 'B';
COMMIT
```

Transaction T2: INTEREST

```
BEGIN
  UPDATE account
  SET bal = bal * 1.06;
COMMIT
```



Examples of Transactions in SQL

Transaction T1: TRANSFER

```
BEGIN
  UPDATE account
  SET bal = bal + 100
  WHERE account_id = 'A';
  --
  UPDATE account
  SET bal = bal - 100
  WHERE account_id = 'B';
COMMIT
```

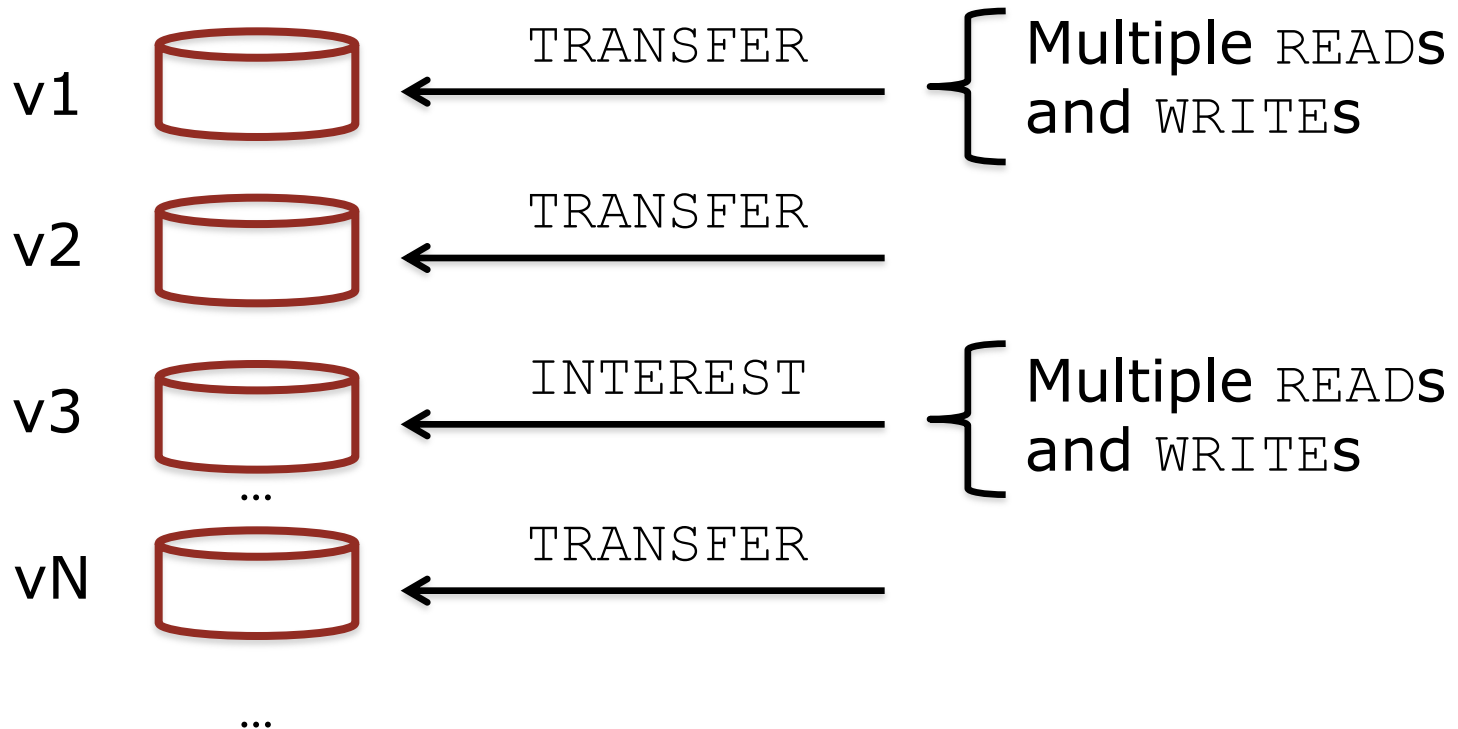
Transaction T2: INTEREST

```
BEGIN
  UPDATE account
  SET bal = bal * 1.06;
COMMIT
```

Under the hood, we know it all translates to calls to READ and WRITE



Conceptual Model: Version Histories



The many faces of atomicity

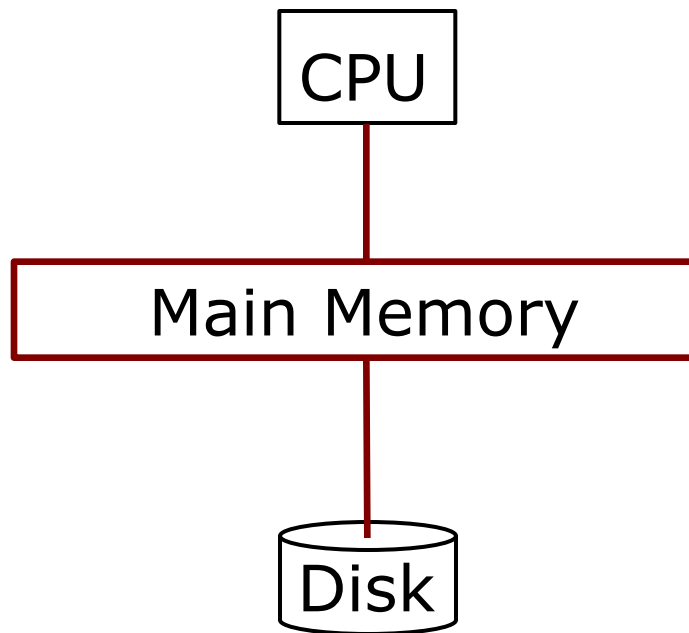
- **Atomicity** is strong modularity mechanism!
 - Hides that one high-level action is actually made of many sub-actions
- **Before-or-after** atomicity
 - == Isolation
 - Cannot have effects that would only arise by interleaving of parts of transactions
- **All-or-nothing** atomicity
 - == Atomicity (+ Durability)
 - Cannot have partially executed transactions
 - Once executed and confirmed, transaction effects are visible and not forgotten



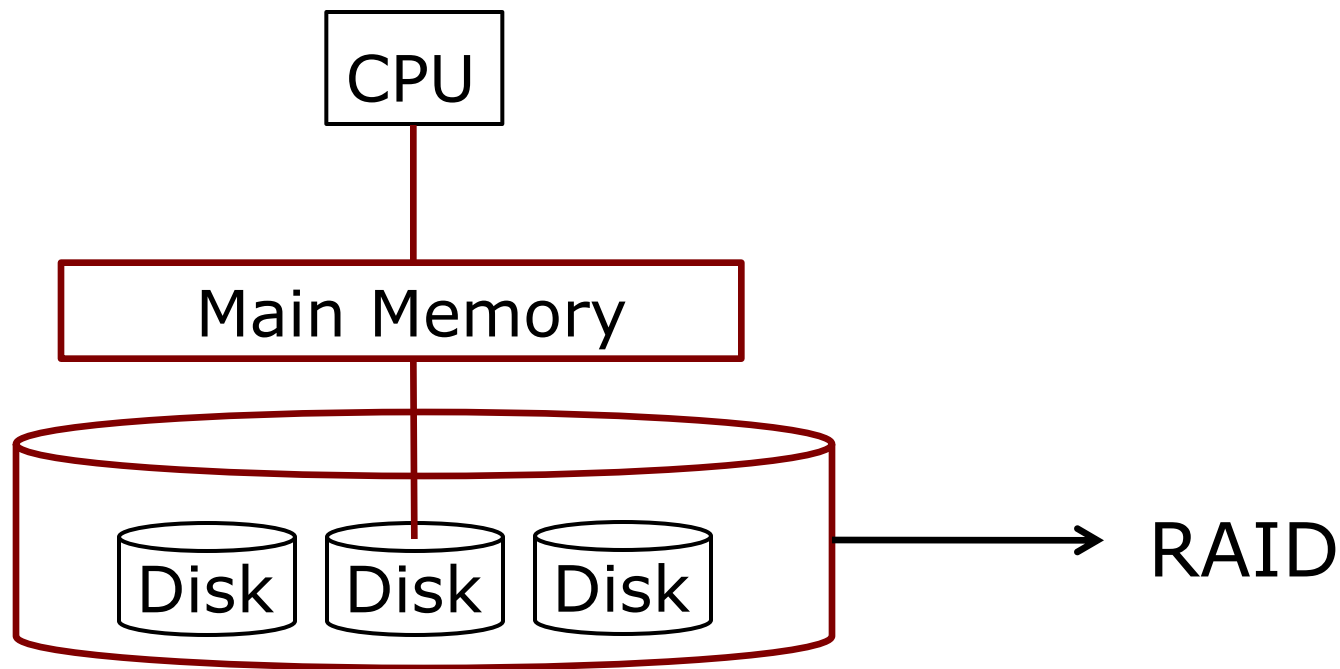
The many faces of atomicity

- **Atomicity** is strong modularity mechanism!
 - Hides that one high-level action is actually made of many sub-actions
- **Before-or-after** atomicity
 - == Isolation
 - Cannot have effects that would only arise by interleaving of parts of transactions
- **All-or-nothing** atomicity
 - == Atomicity (+ Durability)
 - Cannot have partially executed transactions
 - Once executed and confirmed, transaction effects are visible and not forgotten

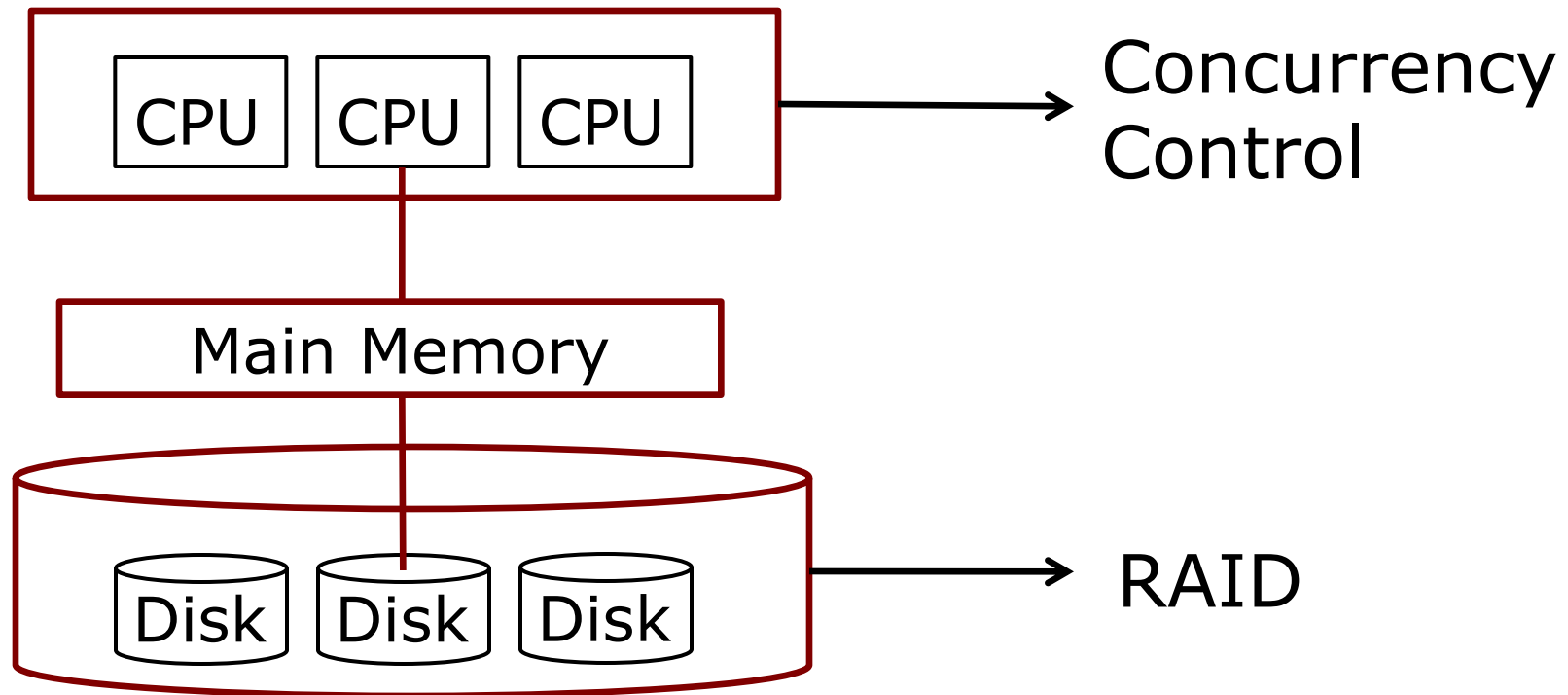
Scaling Up



Scaling Up



Scaling Up



- **Problem:** Ensure automatically that all interactions leave data consistent



Goal of Concurrency Control

- Transactions should be executed so that it is *as though* they executed in some serial order
 - Also called **Isolation** or **Serializability** or **Before-or-after atomicity**
- Weaker variants also possible
 - Lower “degrees of isolation”



Example

- Consider again our two transactions (*Xacts*):

T1:	BEGIN	A=A+100,	B=B-100	END
T2:	BEGIN	A=1.06*A,	B=1.06*B	END

- T1** transfers \$100 from B's account to A's account
- T2** credits both accounts with 6% interest
- If submitted concurrently, net effect should be equivalent to *Xacts* running in some serial order
 - No guarantee that **T1** “logically” occurs before **T2** (or vice-versa) – but one of them is true



Solution 1

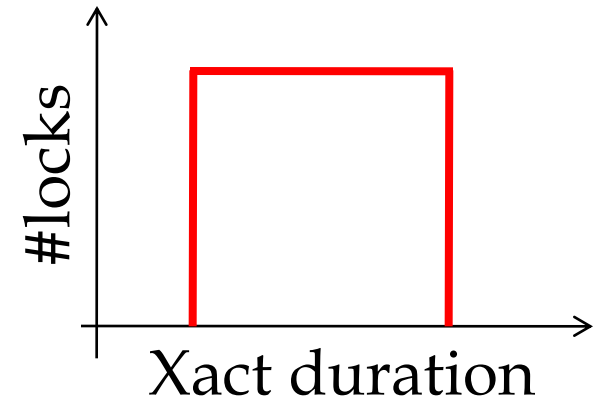
- 1) Get exclusive lock on entire database
 - 2) Execute transaction
 - 3) Release exclusive lock
- Transactions execute in *critical section*
 - Serializability guaranteed because execution is serial!
 - Problems?



Solution 2

- 1) Get exclusive locks on *accessed* data items
- 2) Execute transaction
- 3) Release exclusive locks

- Greater concurrency
- Problems?

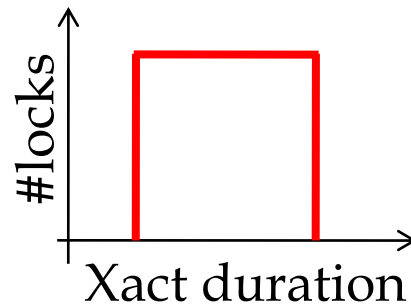


Solution 3

- 1) Get exclusive locks on data items that are *modified*; get shared locks on data items that are only *read*
- 2) Execute transaction
- 3) Release all locks

- Greater concurrency
- Conservative Strict Two Phase Locking (CS2PL)

- Problems?



	S	X
S	Yes	No
X	No	No

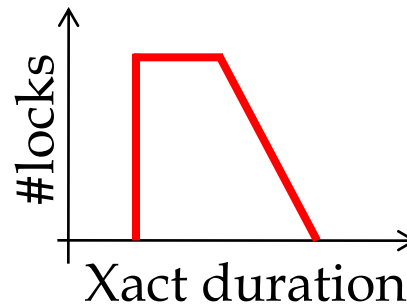


Solution 4

- 1) Get exclusive locks on data items that are modified and get shared locks on data items that are read
- 2) Execute transaction and release locks on objects no longer needed *during execution*

- Greater concurrency
- Conservative Two Phase Locking (C2PL)

- Problems?



	S	X
S	Yes	No
X	No	No

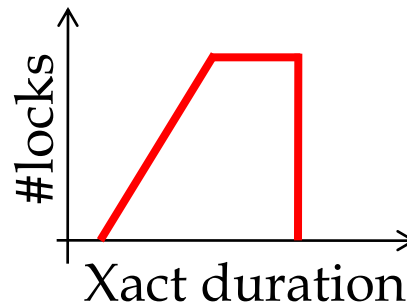


Solution 5

- 1) Get exclusive locks on data items that are modified and get shared locks on data items that are read, but do this *during execution* of transaction (as needed)
- 2) Release all locks

- Greater concurrency
- Strict Two Phase Locking (S2PL)

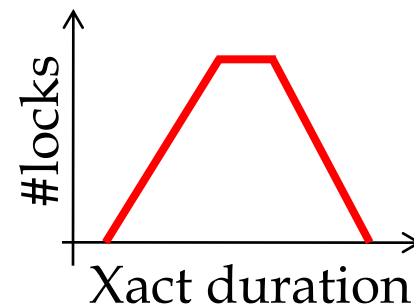
- Problems?



	S	X
S	Yes	No
X	No	No

Solution 6

- 1) Get exclusive locks on data items that are modified and get shared locks on data items that are read, but do this during execution of transaction (as needed)
 - 2) Release locks on objects no longer needed during execution of transaction
 - 3) ***Cannot acquire locks once any lock has been released***
 - ***Hence two-phase (acquiring phase and releasing phase)***
- Greater concurrency
 - Two Phase Locking (2PL)
 - Problems?



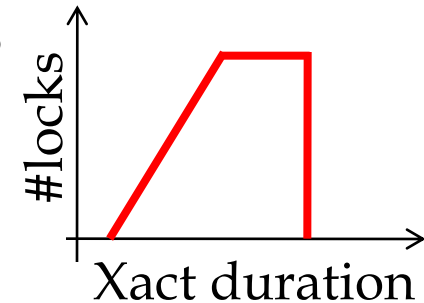
Summary of Alternatives

- Conservative Strict 2PL
 - No deadlocks, no cascading aborts
 - **But** need to know objects a priori, least concurrency
- Conservative 2PL
 - No deadlocks, more concurrency than Conservative Strict 2PL
 - **But** need to know objects a priori, when to release locks, cascading aborts
- Strict 2PL
 - No cascading aborts, no need to know objects a priori or when to release locks, more concurrency than Conservative Strict 2PL
 - **But** deadlocks
- 2PL
 - Most concurrency, no need to know objects a priori
 - **But** need to know when to release locks, cascading aborts, deadlocks



Method of Choice

- Strict 2PL
 - No cascading aborts, no need to know objects a priori or when to release locks, more concurrency than Conservative Strict 2PL
 - But deadlocks
- Reason for choice
 - Cannot know objects a priori, so no Conservative options → only if you would know something about application!
 - Thus only 2PL and Strict 2PL left
 - 2PL needs to know when to release locks (main problem), and has cascading aborts
 - Hence Strict 2PL
- Implication:
 - Need to deal with deadlocks!



	S	X
S	Yes	No
X	No	No



Lock Management

- Lock/unlock requests handled by lock manager
- Lock table entry:
 - Number of transactions currently holding a lock
 - Type of lock held (shared or exclusive)
 - Pointer to queue of lock requests
- Locking and unlocking have to be atomic operations
- **Lock upgrade:** transaction that holds a shared lock can be upgraded to hold an exclusive lock



Questions so far?



Dynamic Databases: Locking the objects that exist now in the database is not enough!

- If we relax the assumption that the DB is a fixed collection of objects, even Strict 2PL will not work correctly:
 - T1 locks all pages containing sailor records with *rating* = 1, and finds oldest sailor (say, *age* = 71).
 - Next, T2 inserts a new sailor; *rating* = 1, *age* = 96.
 - T2 also deletes oldest sailor with *rating* = 2 (and, say, *age* = 80), and commits.
 - T1 now locks all pages containing sailor records with *rating* = 2, and finds oldest (say, *age* = 63).
- No consistent DB state where T1 is “correct”!



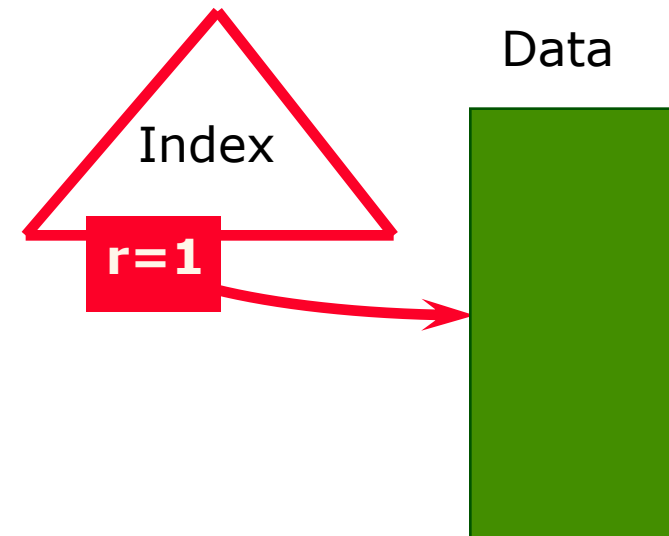
The Problem

- T1 implicitly assumes that it has locked the set of all sailor records with *rating* = 1.
 - Assumption only holds if no sailor records are added while T1 is executing!
 - Need some mechanism to enforce this assumption.
(Index locking and predicate locking.)
- Example shows that correctness is guaranteed for locking on individual objects only if the set of objects is fixed!



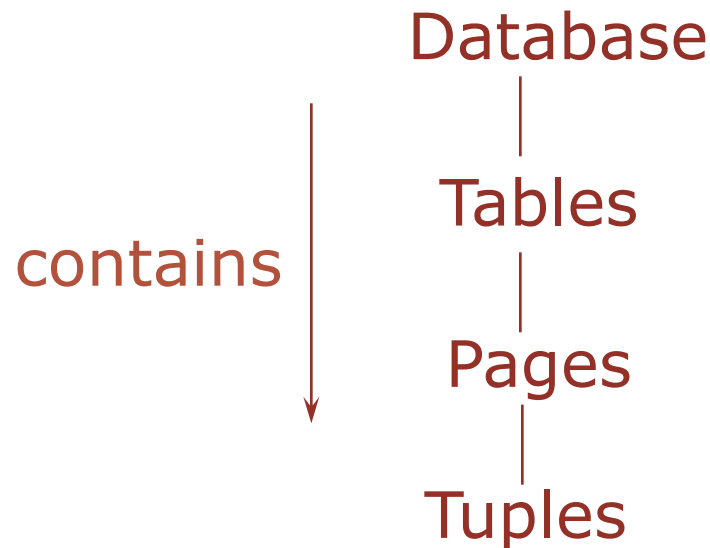
Index Locking

- If data is accessed by an **index** on the *rating* field, T1 should **lock the index page** containing the data entries with *rating* = 1.
 - If there are no records with *rating* = 1, T1 must lock the index page where such a data entry *would* be, if it existed!
- If there is **no suitable index**, T1 must **lock all pages**, and lock the file/table to prevent new pages from being added, to ensure that no new records with *rating* = 1 are added.



Multiple-Granularity Locks

- Hard to decide what granularity to lock (tuples vs. pages vs. tables).
- Shouldn't have to decide!
- Data “containers” are nested:



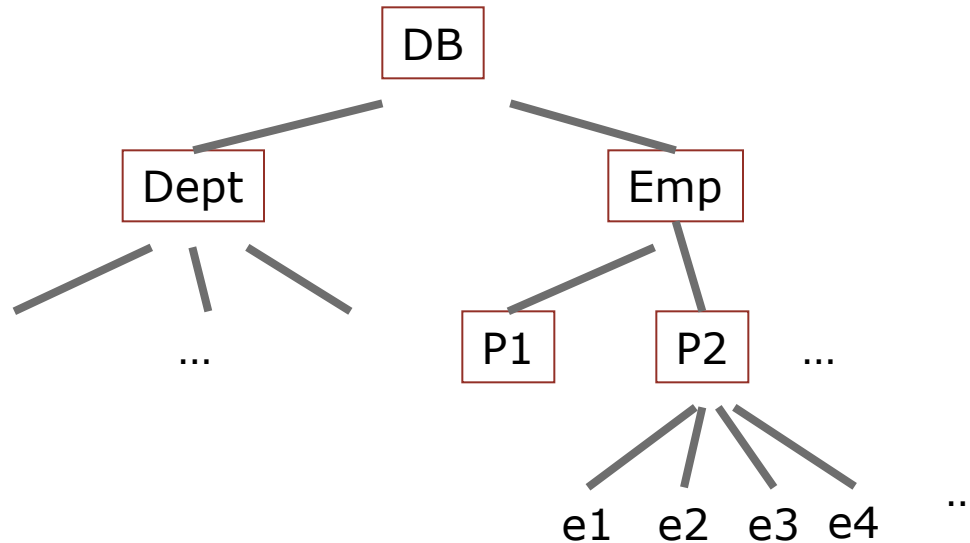
Solution: New Lock Modes, Protocol

- Allow Xacts to lock at each level, but with a special protocol using new **"intention" locks**
- Before locking an item, Xact must set "intention locks" on all its ancestors.
- For unlock, go from specific to general (i.e., bottom-up).
- **SIX mode:** Like S & IX at the same time.

	--	IS	IX	S	X
--	✓	✓	✓	✓	✓
IS	✓	✓	✓	✓	
IX	✓	✓	✓		
S	✓	✓		✓	
X	✓				



Examples: Multiple-Granularity Locks



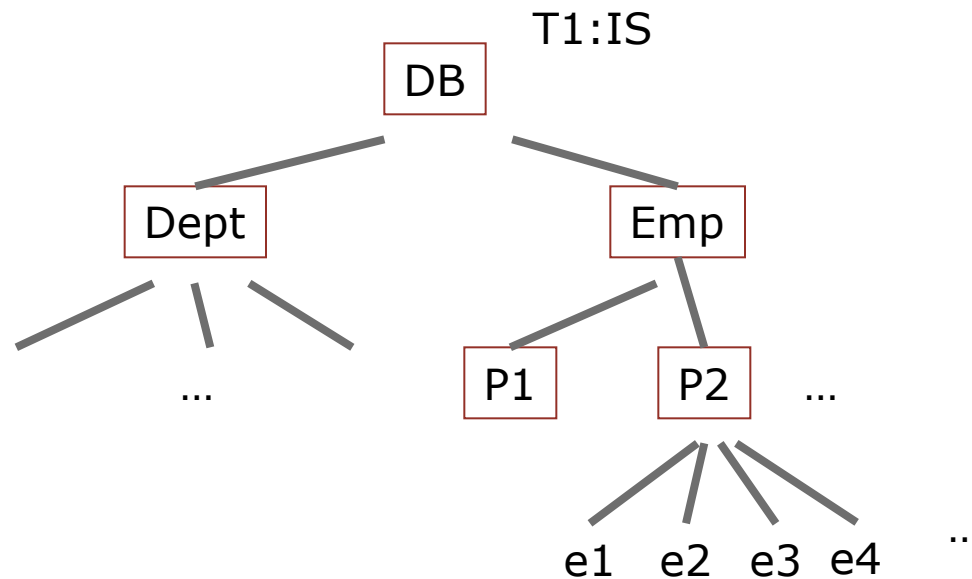
Scenario 1:

T1 scans Emp;
T2 uses
indirect index,
finds e3

T1: SELECT * FROM Emp
WHERE age > 25
T2: UPDATE Emp
SET sal = sal * 1.1
WHERE ssn = 42

	--	IS	IX	S	X
--	✓	✓	✓	✓	✓
IS	✓	✓	✓	✓	
IX	✓	✓	✓		
S	✓	✓		✓	
X	✓				

Examples: Multiple-Granularity Locks



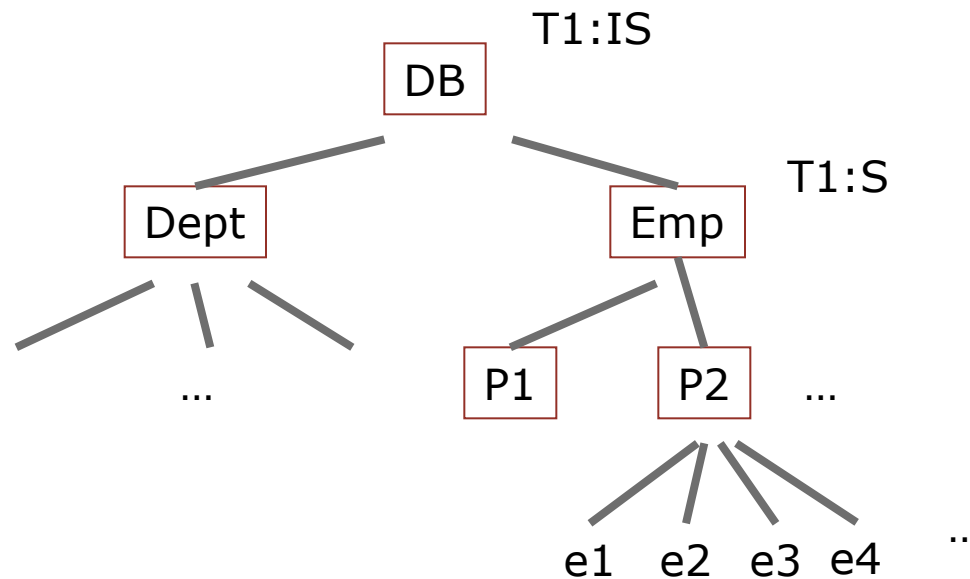
Scenario 1:
 T1 scans Emp;
 T2 uses
 indirect index,
 finds e3

T1: SELECT * FROM Emp
 WHERE age > 25
T2: UPDATE Emp
 SET sal = sal * 1.1
 WHERE ssn = 42

	--	IS	IX	S	X
--	✓	✓	✓	✓	✓
IS	✓	✓	✓	✓	
IX	✓	✓	✓		
S	✓	✓		✓	
X	✓				



Examples: Multiple-Granularity Locks

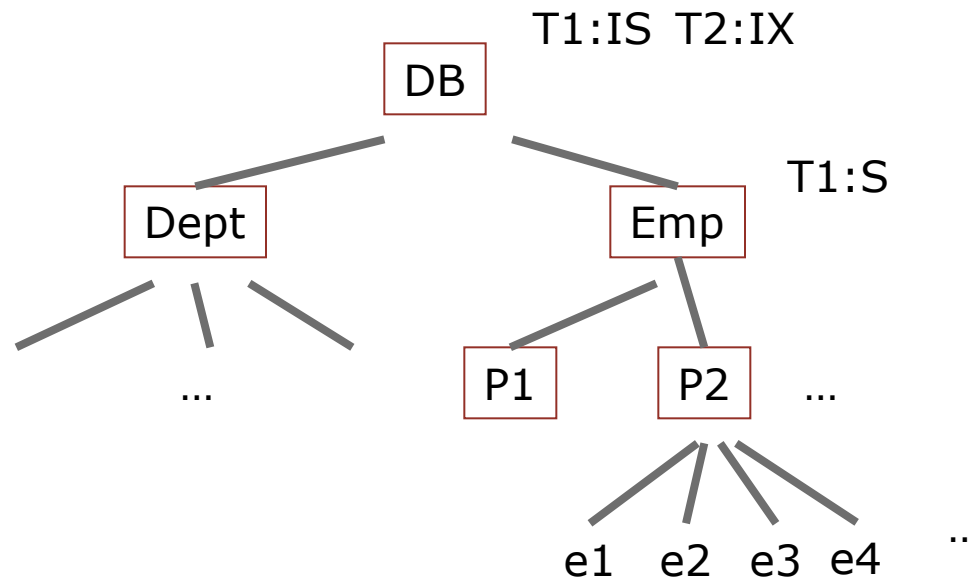


Scenario 1:
 T1 scans Emp;
 T2 uses
 indirect index,
 finds e3

T1: SELECT * FROM Emp
 WHERE age > 25
T2: UPDATE Emp
 SET sal = sal * 1.1
 WHERE ssn = 42

	--	IS	IX	S	X
--	✓	✓	✓	✓	✓
IS	✓	✓	✓	✓	
IX	✓	✓	✓		
S	✓	✓		✓	
X	✓				

Examples: Multiple-Granularity Locks

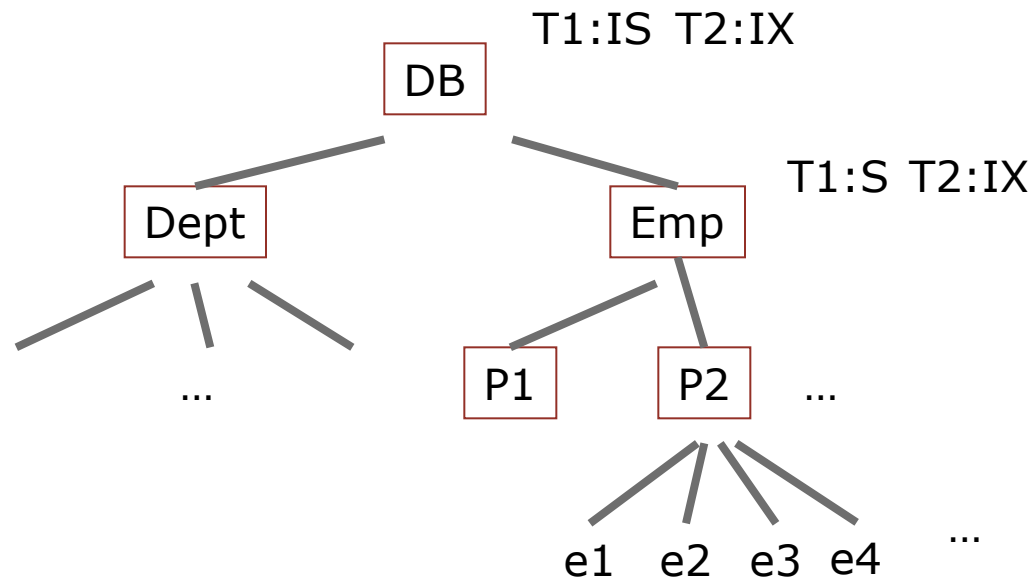


Scenario 1:
 T1 scans Emp;
 T2 uses
 indirect index,
 finds e3

T1: SELECT * FROM Emp
 WHERE age > 25
T2: UPDATE Emp
 SET sal = sal * 1.1
 WHERE ssn = 42

	--	IS	IX	S	X
--	✓	✓	✓	✓	✓
IS	✓	✓	✓	✓	
IX	✓	✓	✓		
S	✓	✓		✓	
X	✓				

Examples: Multiple-Granularity Locks

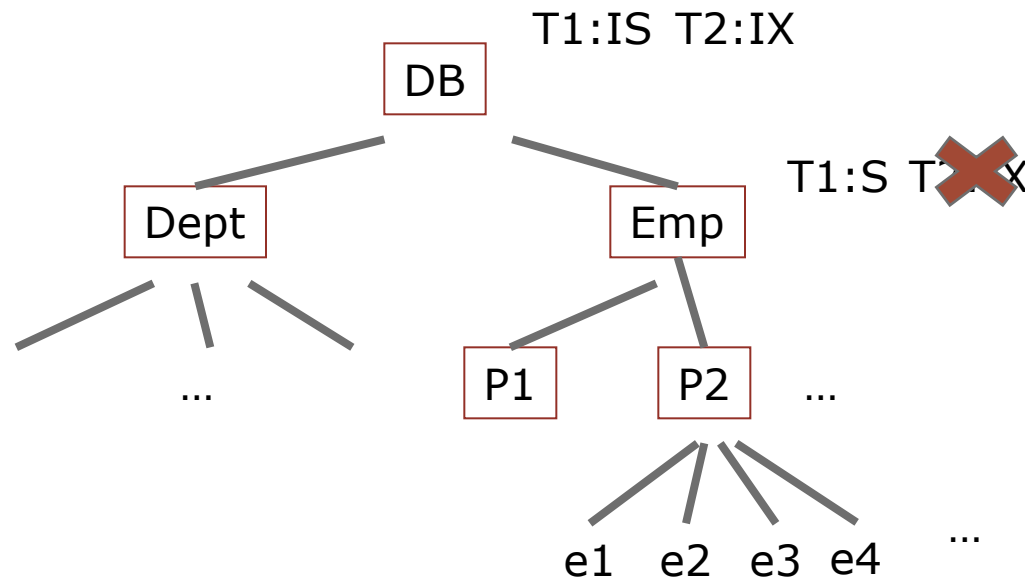


Scenario 1:
 T1 scans Emp;
 T2 uses
 indirect index,
 finds e3

T1: SELECT * FROM Emp
 WHERE age > 25
T2: UPDATE Emp
 SET sal = sal * 1.1
 WHERE ssn = 42

	--	IS	IX	S	X
--	✓	✓	✓	✓	✓
IS	✓	✓	✓	✓	
IX	✓	✓	✓		
S	✓	✓		✓	
X	✓				

Examples: Multiple-Granularity Locks

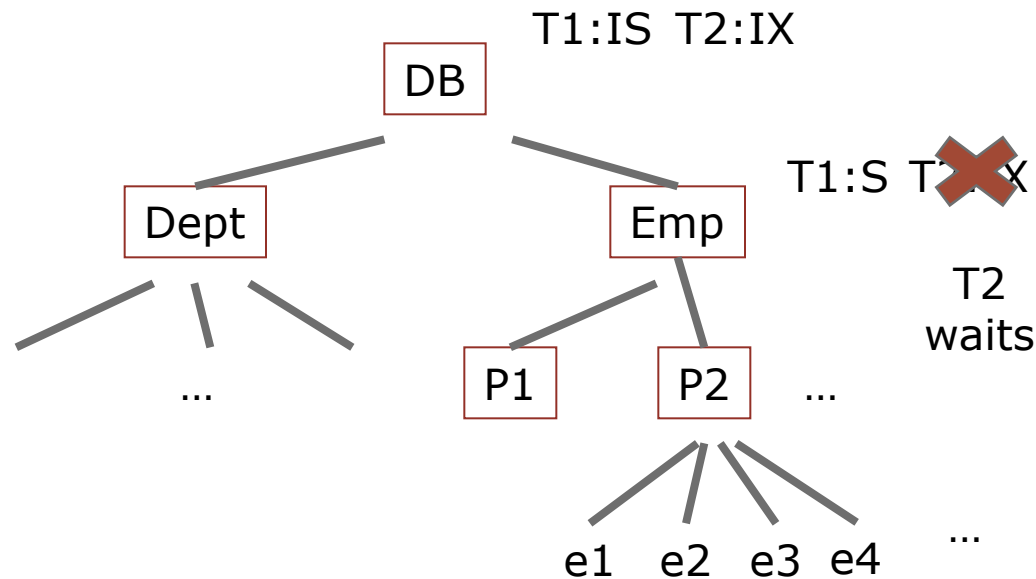


Scenario 1:
 T1 scans Emp;
 T2 uses
 indirect index,
 finds e3

T1: SELECT * FROM Emp
 WHERE age > 25
T2: UPDATE Emp
 SET sal = sal * 1.1
 WHERE ssn = 42

	--	IS	IX	S	X
--	✓	✓	✓	✓	✓
IS	✓	✓	✓	✓	
IX	✓	✓	✓		
S	✓	✓		✓	
X	✓				

Examples: Multiple-Granularity Locks

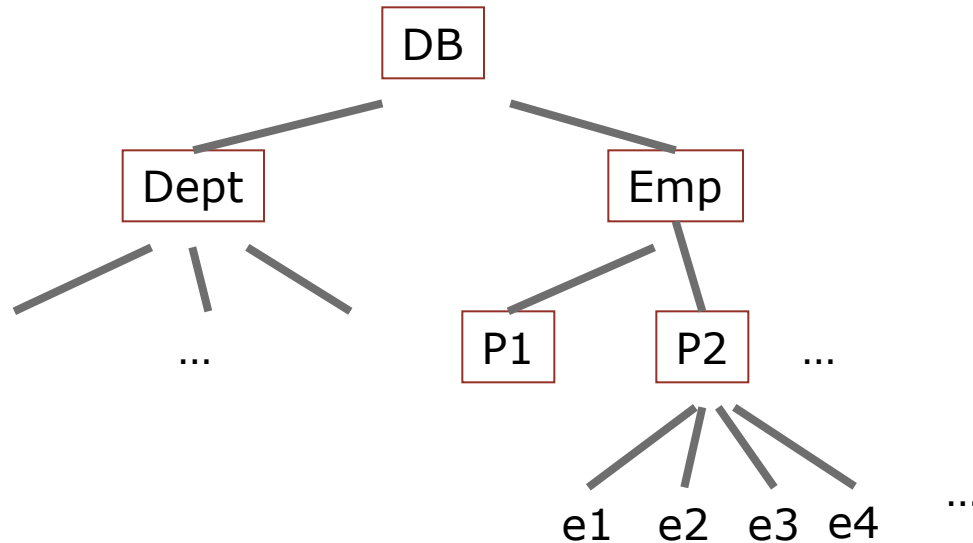


Scenario 1:
 T1 scans Emp;
 T2 uses
 indirect index,
 finds e3

T1: SELECT * FROM Emp
 WHERE age > 25
T2: UPDATE Emp
 SET sal = sal * 1.1
 WHERE ssn = 42

	--	IS	IX	S	X
--	✓	✓	✓	✓	✓
IS	✓	✓	✓	✓	
IX	✓	✓	✓		
S	✓	✓		✓	
X	✓				

Examples: Multiple-Granularity Locks



Scenario 2:

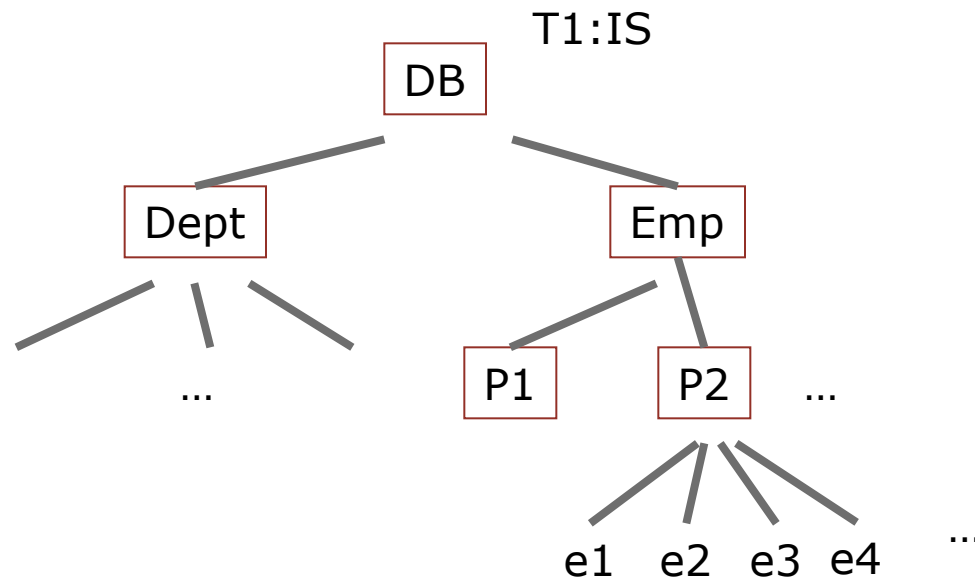
T1 and T2 use indexes; T1 finds e1, e2, e3, etc; T2 finds e3

T1: read several records by key

T2: write to record e3

	--	IS	IX	S	X
--	✓	✓	✓	✓	✓
IS	✓	✓	✓	✓	
IX	✓	✓	✓		
S	✓	✓		✓	
X	✓				

Examples: Multiple-Granularity Locks

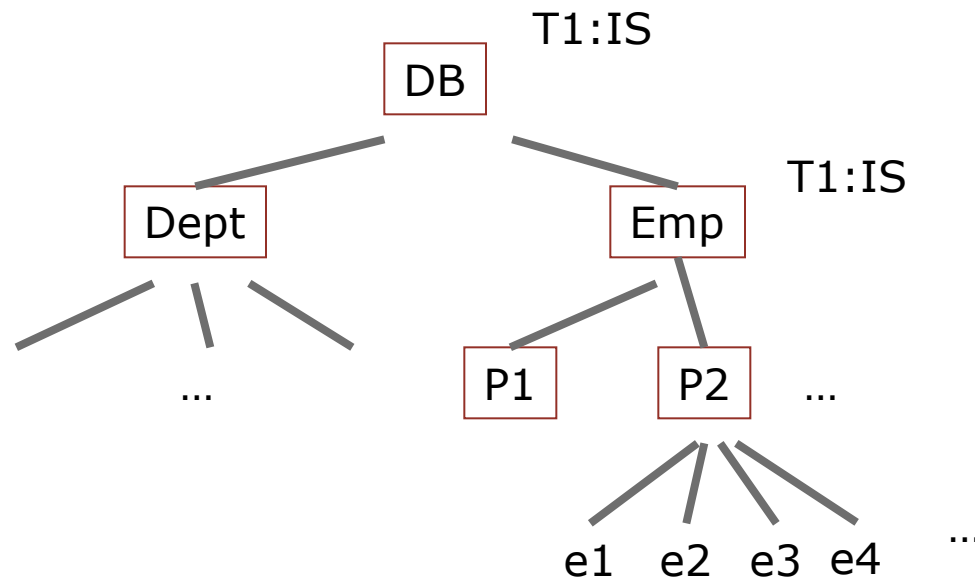


Scenario 2:
 T1 and T2 use
 indexes; T1
 finds e1, e2,
 e3, etc; T2
 finds e3

T1: read several records by key
T2: write to record e3

	--	IS	IX	S	X
--	✓	✓	✓	✓	✓
IS	✓	✓	✓	✓	
IX	✓	✓	✓		
S	✓	✓		✓	
X	✓				

Examples: Multiple-Granularity Locks

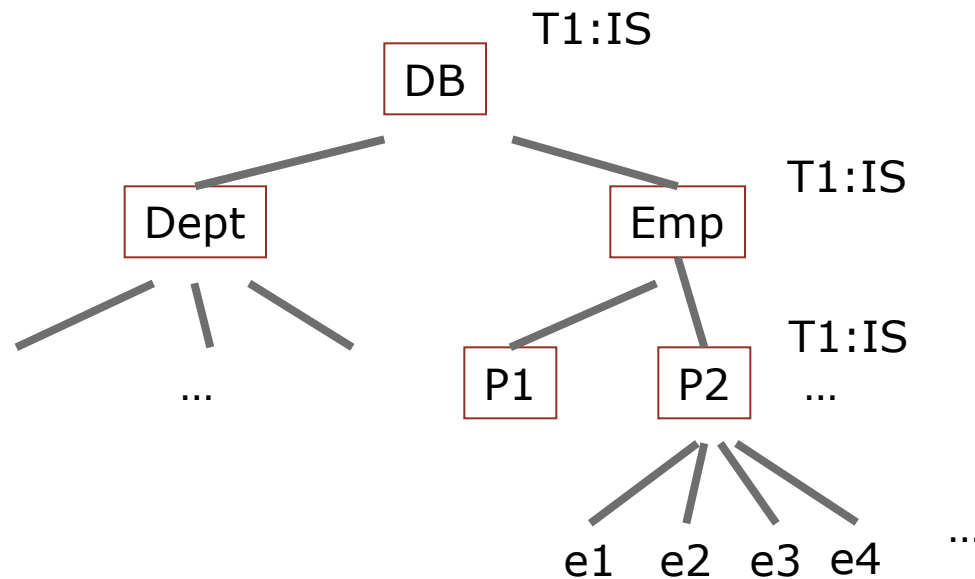


Scenario 2:
T1 and T2 use
indexes; T1
finds e1, e2,
e3, etc; T2
finds e3

T1: read several records by key
T2: write to record e3

	--	IS	IX	S	X
--	✓	✓	✓	✓	✓
IS	✓	✓	✓	✓	
IX	✓	✓	✓		
S	✓	✓		✓	
X	✓				

Examples: Multiple-Granularity Locks

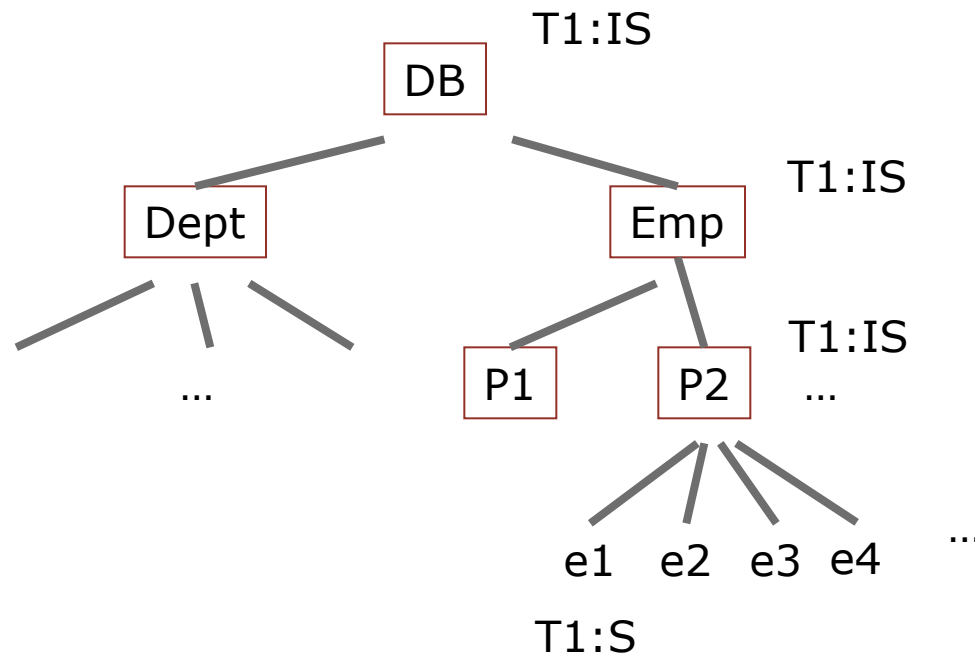


Scenario 2:
 T1 and T2 use
 indexes; T1
 finds e1, e2,
 e3, etc; T2
 finds e3

T1: read several records by key
T2: write to record e3

	--	IS	IX	S	X
--	✓	✓	✓	✓	✓
IS	✓	✓	✓	✓	
IX	✓	✓	✓		
S	✓	✓		✓	
X	✓				

Examples: Multiple-Granularity Locks

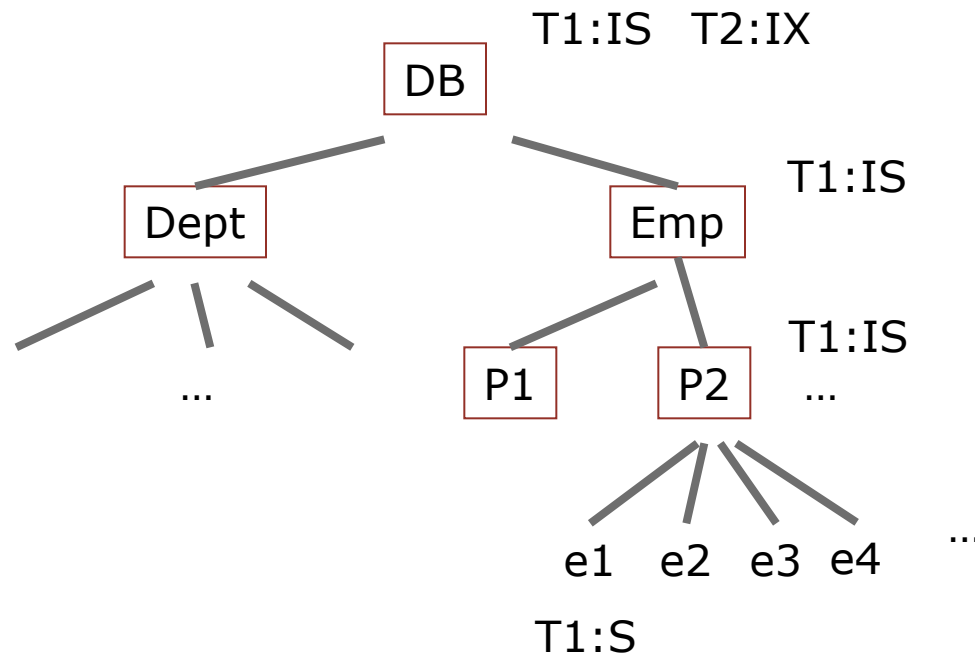


T1: read several records by key
T2: write to record e3

Scenario 2:
 T1 and T2 use
 indexes; T1
 finds e1, e2,
 e3, etc; T2
 finds e3

	--	IS	IX	S	X
--	✓	✓	✓	✓	✓
IS	✓	✓	✓	✓	
IX	✓	✓	✓		
S	✓	✓		✓	
X	✓				

Examples: Multiple-Granularity Locks

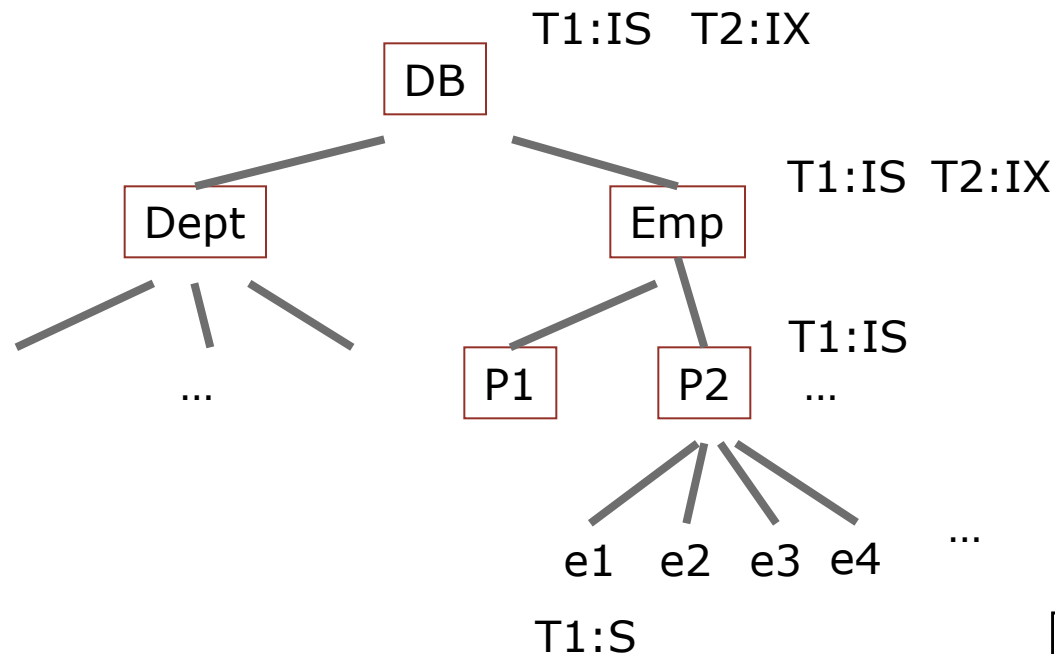


T1: read several records by key
T2: write to record e3

Scenario 2:
 T1 and T2 use
 indexes; T1
 finds e1, e2,
 e3, etc; T2
 finds e3

	--	IS	IX	S	X
--	✓	✓	✓	✓	✓
IS	✓	✓	✓	✓	
IX	✓	✓	✓		
S	✓	✓		✓	
X	✓				

Examples: Multiple-Granularity Locks

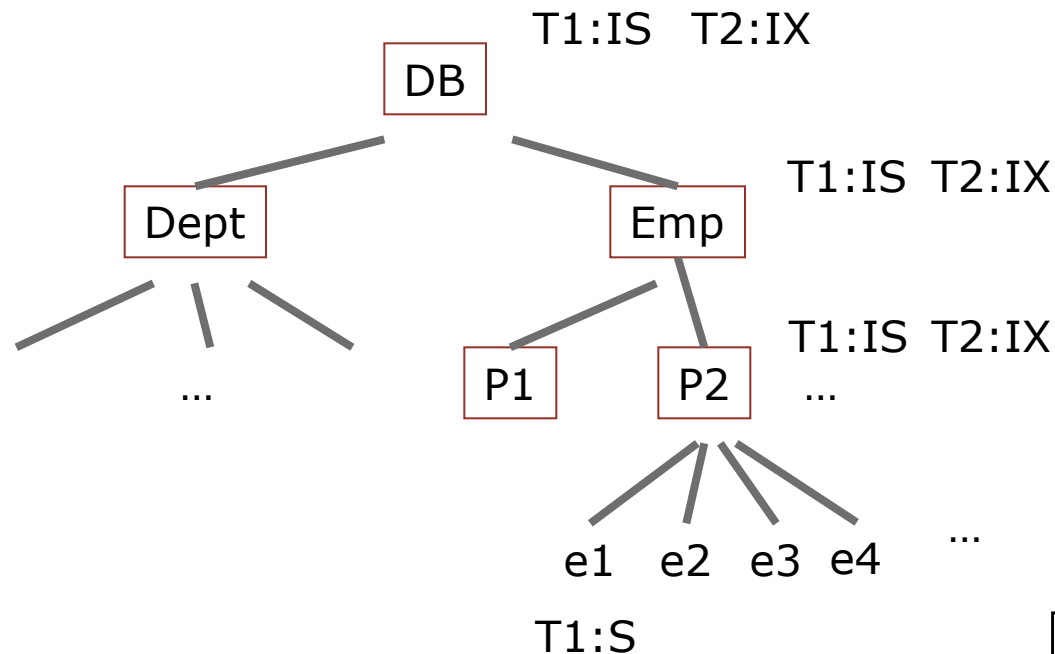


Scenario 2:
 T1 and T2 use indexes; T1 finds e1, e2, e3, etc; T2 finds e3

T1: read several records by key
T2: write to record e3

	--	IS	IX	S	X
--	✓	✓	✓	✓	✓
IS	✓	✓	✓	✓	
IX	✓	✓	✓		
S	✓	✓		✓	
X	✓				

Examples: Multiple-Granularity Locks

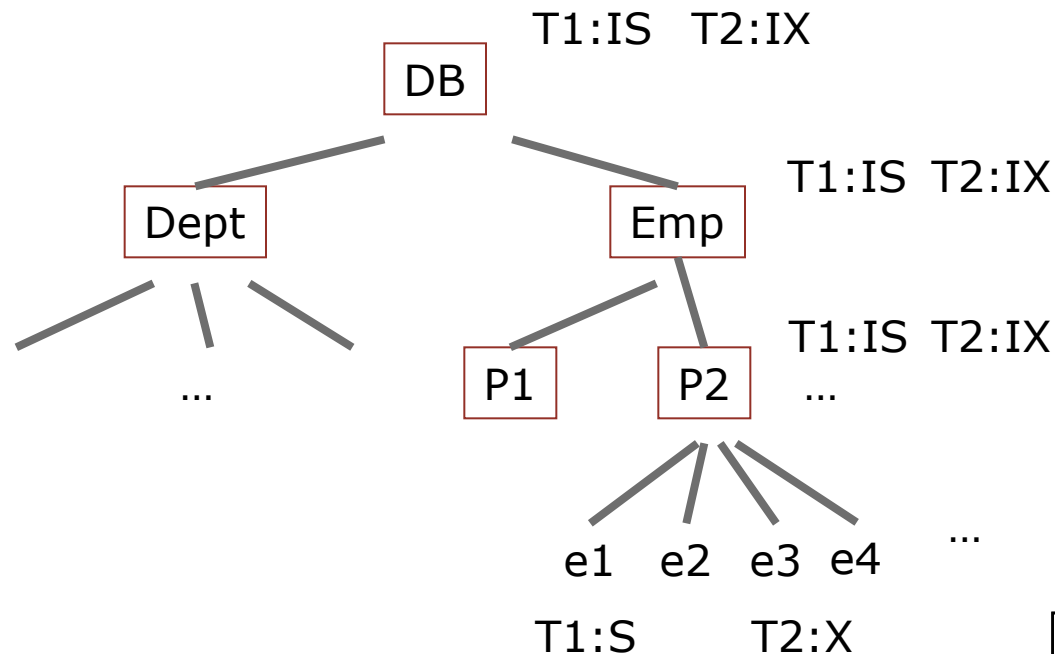


Scenario 2:
 T1 and T2 use indexes; T1 finds e1, e2, e3, etc; T2 finds e3

T1: read several records by key
T2: write to record e3

	--	IS	IX	S	X
--	✓	✓	✓	✓	✓
IS	✓	✓	✓	✓	
IX	✓	✓	✓		
S	✓	✓		✓	
X	✓				

Examples: Multiple-Granularity Locks

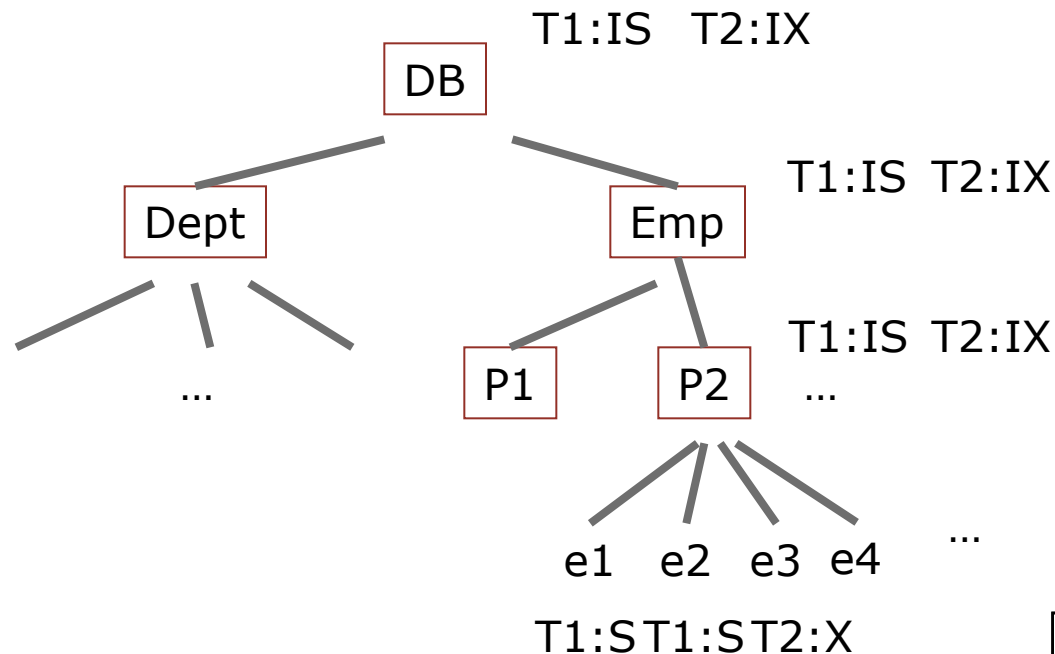


Scenario 2:
 T1 and T2 use indexes; T1 finds e1, e2, e3, etc; T2 finds e3

T1: read several records by key
T2: write to record e3

	--	IS	IX	S	X
--	✓	✓	✓	✓	✓
IS	✓	✓	✓	✓	
IX	✓	✓	✓		
S	✓	✓		✓	
X	✓				

Examples: Multiple-Granularity Locks

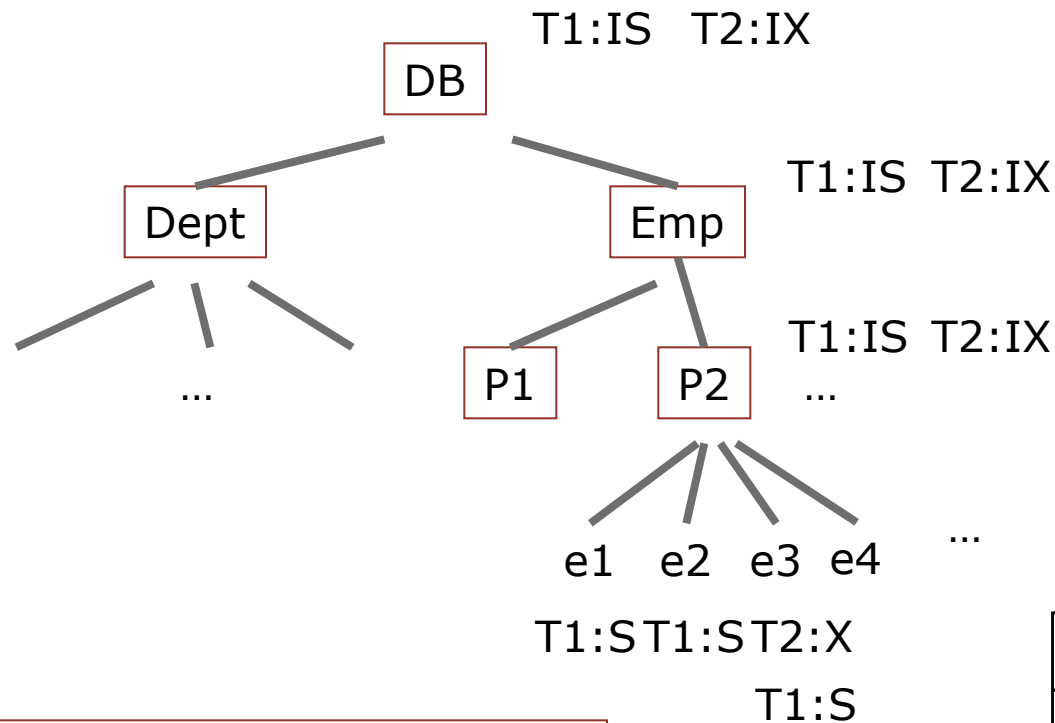


Scenario 2:
T1 and T2 use indexes; T1 finds e1, e2, e3, etc; T2 finds e3

T1: read several records by key
T2: write to record e3

	--	IS	IX	S	X
--	✓	✓	✓	✓	✓
IS	✓	✓	✓	✓	
IX	✓	✓	✓		
S	✓	✓		✓	
X	✓				

Examples: Multiple-Granularity Locks

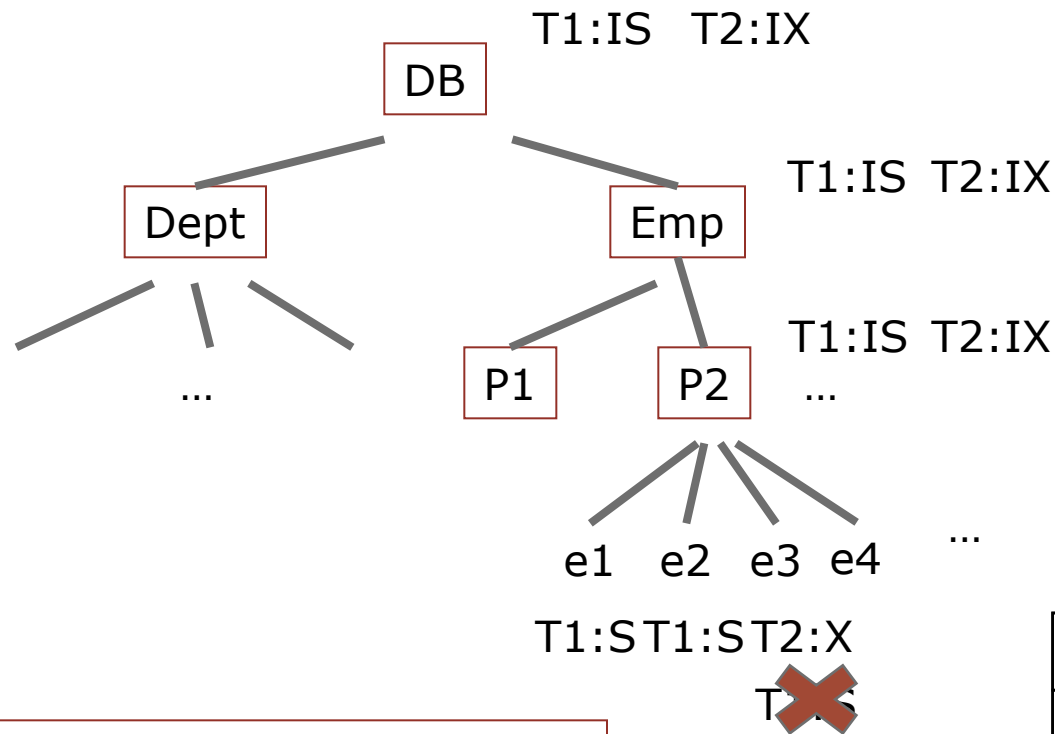


Scenario 2:
T1 and T2 use indexes; T1 finds e1, e2, e3, etc; T2 finds e3

T1: read several records by key
T2: write to record e3

	--	IS	IX	S	X
--	✓	✓	✓	✓	✓
IS	✓	✓	✓	✓	
IX	✓	✓	✓		
S	✓	✓		✓	
X	✓				

Examples: Multiple-Granularity Locks

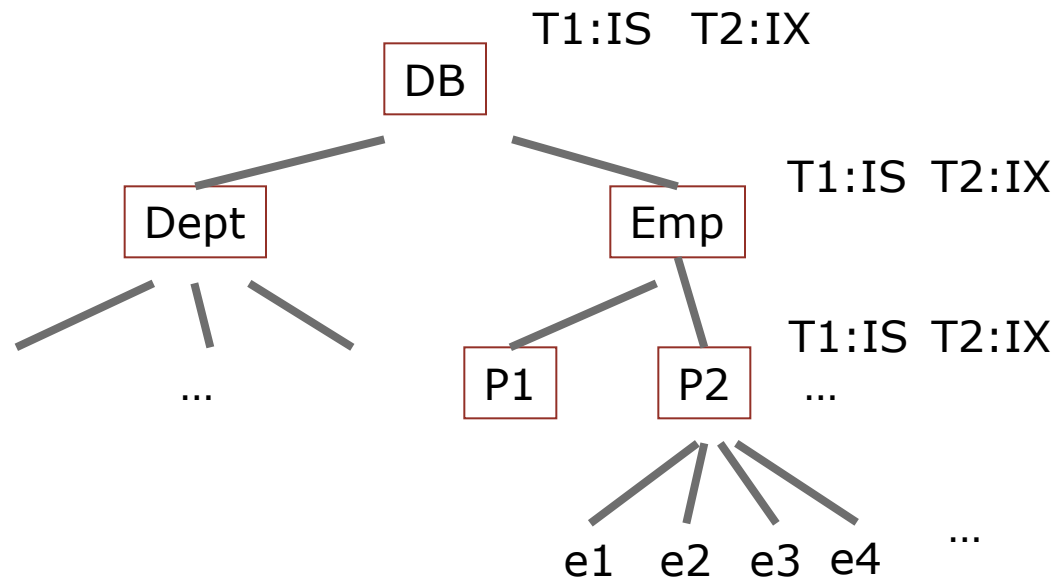


Scenario 2:
 T1 and T2 use indexes; T1 finds e1, e2, e3, etc; T2 finds e3

T1: read several records by key
T2: write to record e3

	--	IS	IX	S	X
--	✓	✓	✓	✓	✓
IS	✓	✓	✓	✓	
IX	✓	✓	✓		
S	✓	✓		✓	
X	✓				

Examples: Multiple-Granularity Locks



T1: read several records by key
T2: write to record e3

T1:ST1:ST2:X
~~T2:IX~~
 T1
 waits

Scenario 2:
 T1 and T2 use
 indexes; T1
 finds e1, e2,
 e3, etc; T2
 finds e3

	--	IS	IX	S	X
--	✓	✓	✓	✓	✓
IS	✓	✓	✓	✓	
IX	✓	✓	✓		
S	✓	✓		✓	
X	✓				

Questions so far?

Is Strict 2PL correct? (assuming database is **not** dynamic)

- We will formalize now and next class **serializability** and argue that Strict 2PL is correct
 - Full proof is left as homework 😊
- Strict 2PL can however deadlock
 - We will see how to handle deadlocks automatically



Schedules

- Consider a possible interleaving (schedule):

T1:	$A = A + 100,$	$B = B - 100$
T2:	$A = 1.06 * A, \quad B = 1.06 * B$	

- The system's view of the schedule:

T1:	$R(A), W(A),$	$R(B), W(B)$
T2:	$R(A), W(A), R(B), W(B)$	

Scheduling Transactions

- *Serial schedule*: Schedule that does not interleave the actions of different transactions.
- *Equivalent schedules*:
 - Involve the same actions of the same transactions
 - For any database state the effect (on the set of objects in the database) of executing the two schedules is the same
- *Serializable schedule*: A schedule that is equivalent to some serial execution of the transactions.

(Note: If each transaction preserves consistency, every serializable schedule preserves consistency.)



Anomalies with Interleaved Execution

- Reading Uncommitted Data (WR Conflicts, “dirty reads”):

T1:	R(A), W(A),	R(B), W(B), Abort
T2:	R(A), W(A), C	

- Unrepeatable Reads (RW Conflicts):

T1:	R(A),	R(A), W(A), C
T2:	R(A), W(A), C	

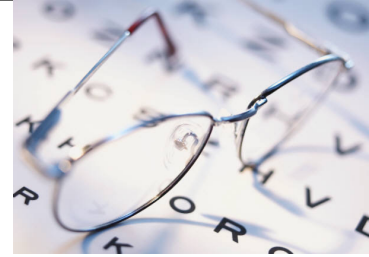


Anomalies (contd.)

- Overwriting Uncommitted Data (WW Conflicts):

T1:	W(A) ,	W(B) , C
T2:	W(A) , W(B) , C	

What should we learn today?



- Identify the multiple interpretations of the property of atomicity
- Implement methods to ensure before-or-after atomicity, and argue for their correctness
- Explain the variants of the two-phase locking (2PL) protocol, in particular the widely-used Strict 2PL
- Explain situations where predicate locking is required
- Discuss the definition of serializability and the notion of anomalies