18.3 - 18.4 More Concurrency Control

Database System Concepts
Rachel Lee

Content

18.3 Multiple Granularity

a. Explicit/Implicit Lock Modes
b. Intention Lock Modes
c. Multiple Granularity Locking Protocol

18.4 Delete, Insert, and Predicate Reads

a. Delete Operation
b. Insert Operation
c. The Phantom Phenomenon + Solutions

18.3 Multiple Granularity

Granularity - “the scale or level of detail present”

A transaction $\mathbf{T}_i$ might need to access an entire relation (table), while a different transaction $\mathbf{T}_j$ might only need to access a single tuple (row)

→ Question: Should there be one big lock for entire relation or individual small locks for each tuple ?

→ Solution: **Hierarchy of granularity,** with smaller granularities (i.e. tuples) nested inside larger granularities (i.e relations)

Example

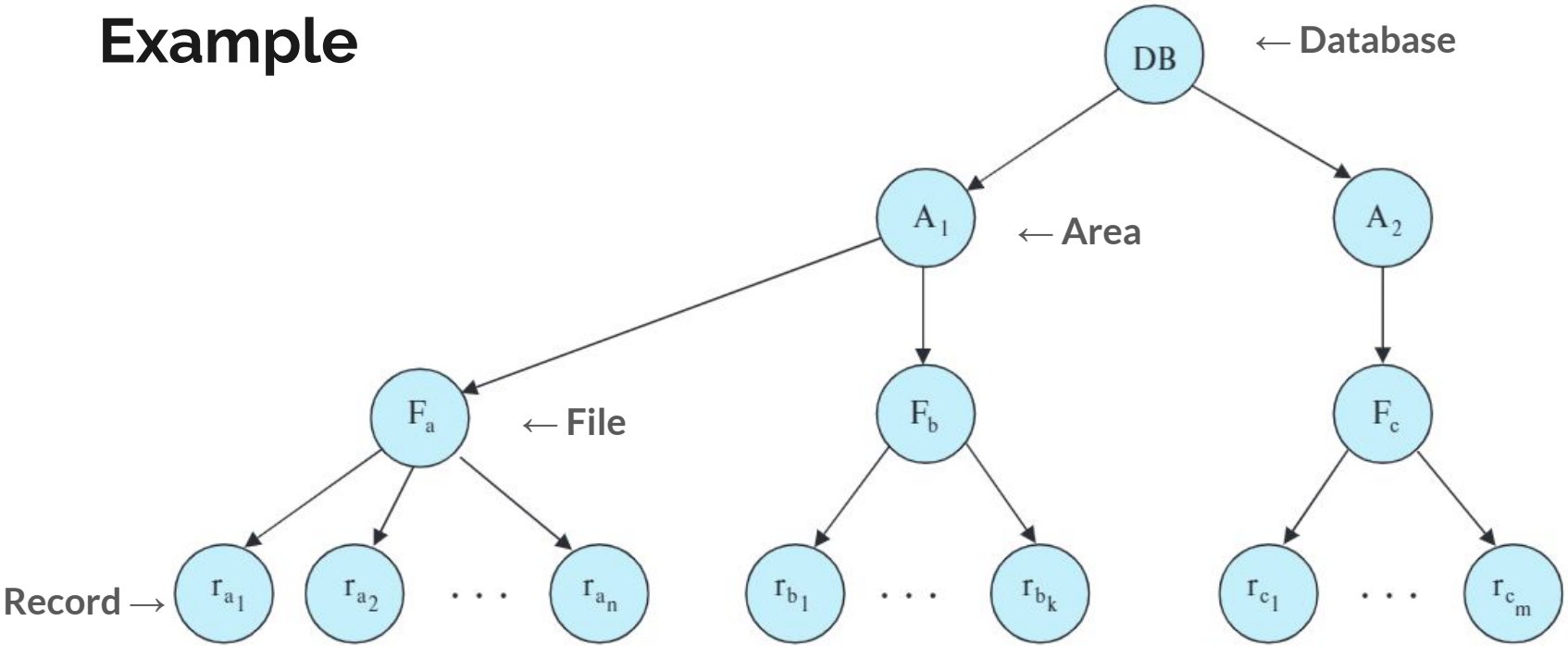
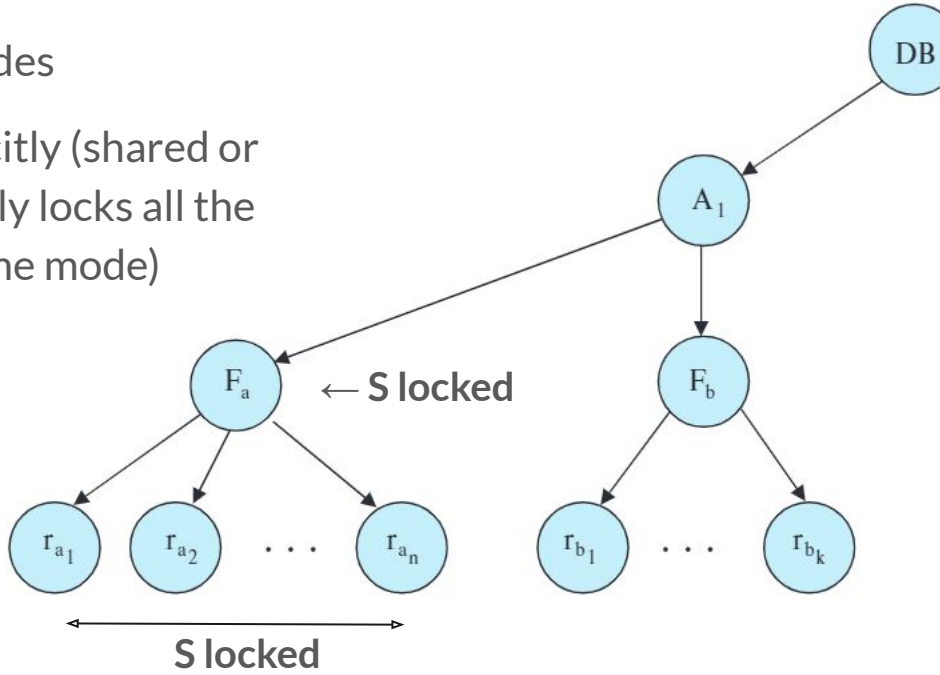


Figure 18.15 Granularity hierarchy.

The hierarchy assumes that if you need a file, you're going to need all the records in the file too.

Explicit/Implicit Lock Modes

- Uses the shared and exclusive lock modes
- When a transaction locks a node explicitly (shared or exclusive), the transaction also implicitly locks all the descendants of that node (with the same mode)
- Example: If T_i wants to read F_a , then T_i puts an *explicit* shared lock on F_a and *implicit* shared locks on r_{a1} - r_{an}

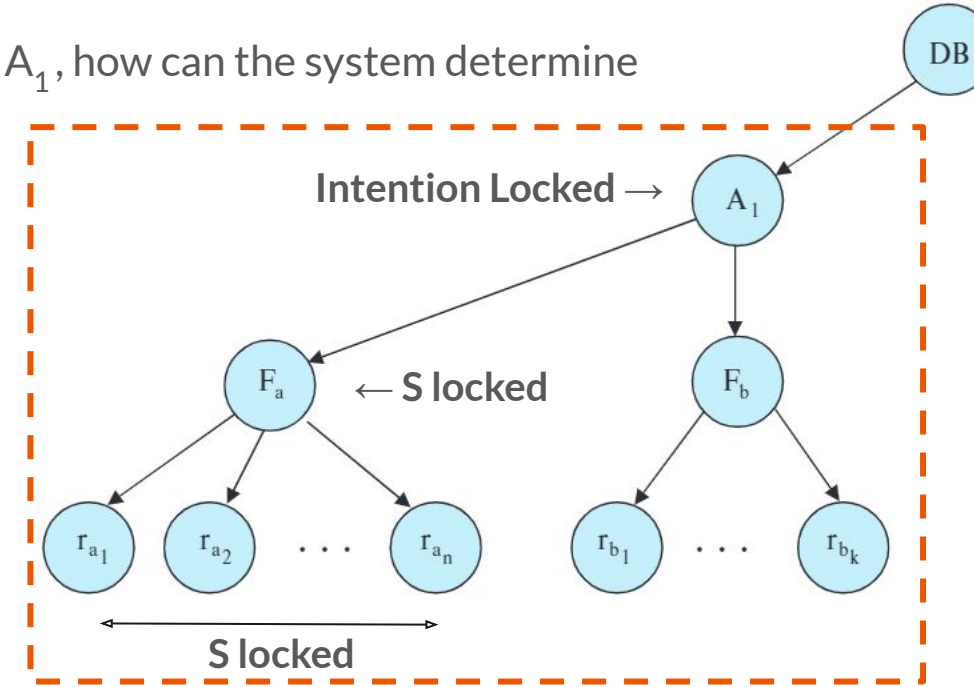


Intention Lock Modes

- Example: T_j wants to exclusively lock A_1, how can the system determine if this node can be locked?

  → Place **intention locks** on all ancestors above a locked node.

- **Intention Lock Modes** indicate which nodes *cannot* be locked successfully because they would conflict with existing locks on descendent nodes



Intention Lock Modes

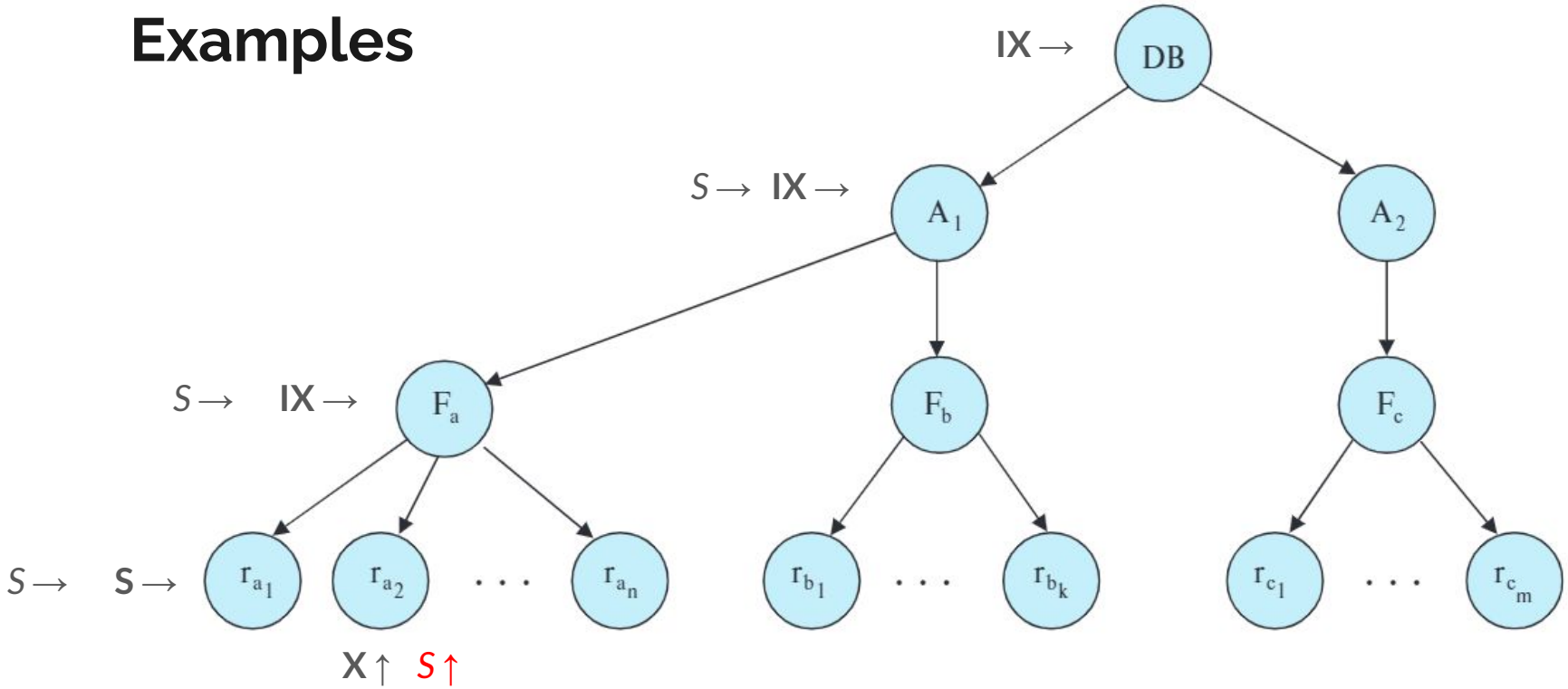
- Intention Lock Types
  - Intention-Shared Mode (IS): there is explicit locking at a lower level of the tree, but only shared-mode locks
  - Intention-Exclusive Mode (IX): there is explicit locking at a lower level of the tree, with exclusive-mode OR shared-mode locks
  - Shared and Intention-Exclusive (SIX) mode: this node is locked explicitly in shared-mode, and there is explicit locking at a lower level with exclusive-mode locks
- Note: Locks are acquired in top-down order, and released in bottom up order

	IS	IX	S	SIX	X
IS	true	true	true	true	false
IX	true	true	false	false	false
S	true	false	true	false	false
SIX	true	false	false	false	false
X	false	false	false	false	false

Figure 18.16 Compatibility matrix.

Example: If a node currently in Intention-Shared (IS) lock mode, then a different transaction could read from that node, Shared lock (S), but it could not write to the node, Exclusive lock (X).

Examples



1. T_i reads r_{a1} ✅ 2. T_j writes to r_{a2} ✅ 3. T_k reads A_1 ❌

Multiple-Granularity Locking Protocol

1. Transaction T_i must observe the lock-compatibility function of compatibility matrix
2. Transaction T_i must lock the root of the tree first, in any mode (top-down traversal)
3. Transaction T_i can lock a node Q in S or IS mode only if T_i currently has the parent of Q locked in either IX or IS mode (i.e. remember to intention lock *first*)
4. Transaction T_i can lock a node Q in X, SIX, or IX mode only if T_i currently has the parent of Q locked in either IX or SIX mode (i.e. same as previous bullet but exclusive)
5. Transaction T_i can lock a node only if T_i has not previously unlocked any node (i.e. T_i is two phase)
6. Transaction T_i can unlock a node Q only if T_i currently has none of the children of Q locked (i.e. maintain intention locking: if a child node is locked, the parent node should indicate that using intention locking)

18.4 Delete, Insert, and Predicate Reads

Delete Operation

Conflicts with **delete(Q)** where **Q** is any data item

- **read(Q)** before **delete(Q)** is successful, **delete(Q)** before **read(Q)** is a logical error
- **write(Q)** before **delete(Q)** is successful, **delete(Q)** before **write(Q)** is a logical error
- **delete(Q)** before or after **delete(Q)** is a logical error
- if **Q** did not exist prior: **insert(Q)** before **delete(Q)** succeeds, **delete(Q)** before **insert(Q)** fails
- If **Q** did exist prior: **delete(Q)** before **insert(Q)** succeeds, **insert(Q)** before **delete(Q)** fails

delete(Q) has conflicts with all of the operations

Handling Delete Operation

Under the two-phase locking protocol: An **exclusive lock** is required for any data item before the item can be deleted

Under the time-stamp ordering protocol: Suppose T_i issues **delete(Q)**

- If the $TS(T_i) < R\text{-timestamp}(Q)$, then the value of Q that T_i was to delete has already been read by transaction T_j with $TS(T_j) > TS(T_i)$. So **delete(Q)** is rejected.
- If the $TS(T_i) < W\text{-timestamp}(Q)$, then a transaction T_j with $TS(T_j) > TS(T_i)$ has written Q already. So **delete(Q)** is rejected.
- Otherwise, the **delete(Q)** is executed

Insert Operation

insert(Q) is treated similar to a **write(Q)** operation, it conflicts with everything

Under the two-phase locking protocol: If T_i performs an **insert(Q)** operation, T_i is given an **exclusive** lock on the newly created data item Q

Under the time-stamp ordering protocol: If T_i performs an **insert(Q)** operation, the values R-timestamp(Q) and W-timestamp(Q) are set to $TS(T_i)$

The Phantom Phenomenon

Example:

T_{30}:
```
select count(*)
from instructor
where dept_name = 'Physics' ;
```

T_{31}:
```
insert into instructor
        values (11111, 'Feynman', 'Physics', 94000);
```

Problem: Does T_{30} reads the value written by T_{31} or not in the count(*)? Different serial schedules lead to conflict with each other, however T_{30} and T_{31} never access the same tuple (i.e. a “phantom” conflict).

The Phantom Phenomenon

- Since the two transactions never conflict on a specific tuple, concurrency control on a tuple level would never detect the conflict.
- It is not sufficient to only lock the data items being accessed, but transactions must also lock the information used to find the items being accessed

- Example: If T_{30} locked the entire relation *instructor*, then when T_{31} tries to insert into *instructor*, you get a conflict on a real data item rather than a phantom

```
select count(*)
from instructor
where dept_name = 'Physics' ;
```

Index-Locking

- Since locking an entire relation causes a low degree of concurrency, a better solution is to use indices that access relations (i.e. B+ trees)
- Example: Assuming there is an index page on *instructor* with the attribute *dept_name*, if modifications to any instructor require the transaction to modify the index containing “Physics”, then both T_{30} and T_{31} would have to access it, leading to a conflict.

Index-Locking Protocol

1. Every relation must have at least one index
2. A transaction T_i can access tuples of a relation only after first finding them through one or more of the indices on the relation (i.e. first check if the index is unlocked)
3. A transaction T_i that performs a lookup must acquire a shared lock on all the index leaf nodes that it accesses (i.e. lock all of the information used to find the data)
4. A transaction T_i may not insert, delete, or update a tuple in a relation without updating all indices in the relation, and obtaining exclusive locks on all index leaf nodes that are affected by the insertion, deletion, or update.
5. Locks are still obtained on tuples as usual.
6. The rules of two-phase locking protocol are still observed as usual.

Predicate Locking

Alternative Approach: Transactions acquire shared locks on predicates in a query. A lock conflict is only caused if the insert/delete satisfies the predicate.

Example: The predicate "*salary > 90000*" on the instructor relation would cause a conflict if it there was a lock on a tuple with the salary > 90000.

Downside: Very expensive and implementation is not significantly better than Index-Locking.

References

https://www.youtube.com/watch?v=qK3qohb2XU4 - multiple granularity example

https://youtu.be/mFWoNHOb5dQ?t=1823 - phantom predicates, index locking, predicate locking