Concurrency Control: Serializability, Schedules, Advanced Topics

PCSD, Marcos Vaz Salles

Do-it-yourself-recap: Locking Solutions for Isolation in ACID Transactions

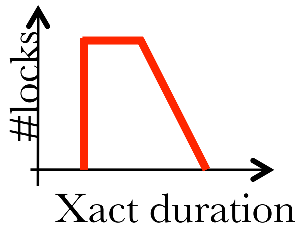
	S	X
S	Yes	No
X	No	No

Conflict or View Serializable? Deadlocks? Cascading aborts?

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*

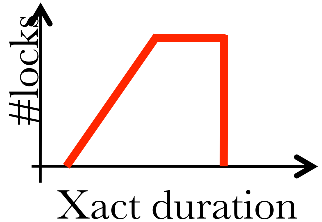
- Greater concurrency
- Conservative Two Phase Locking (2PL)
- Problems?

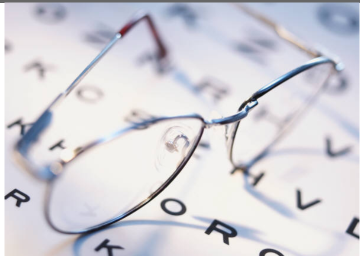


Solution 5

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 (2PL)
- Problems?





What should we learn today?

- Explain deadlock prevention and detection techniques
- Apply deadlock detection using a waits-for graph to transaction schedules
- Explain situations where predicate locking is required
- Explain the optimistic concurrency control and multi-version concurrency control models
- Predict validation decisions under optimistic concurrency control

Deadlocks

- Deadlock: Cycle of transactions waiting for locks to be released by each other.
- Two ways of dealing with deadlocks:
  - Deadlock prevention
  - Deadlock detection

Deadlock Prevention

- Assign priorities based on timestamps. Assume Ti wants a lock that Tj holds. Two policies are possible:
  - Wait-Die: It Ti has higher priority, Ti waits for Tj; otherwise Ti aborts
  - Wound-wait: If Ti has higher priority, Tj aborts; otherwise Ti waits

- If a transaction re-starts, make sure it has its original timestamp

Source: Ramakrishnan & Gehrke (partial)

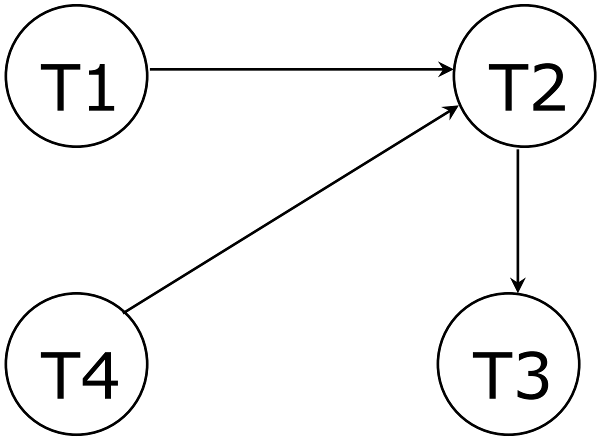
Deadlock Detection

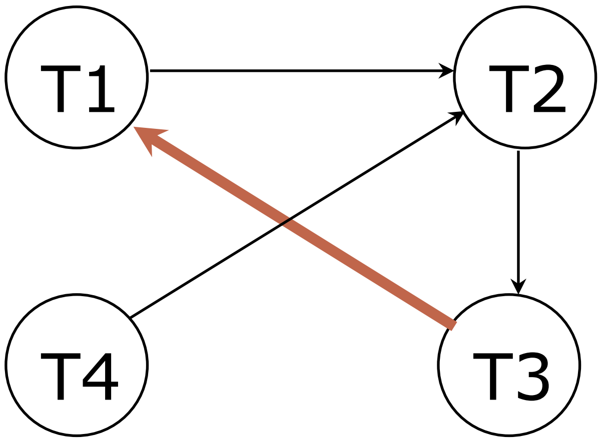
- Create a waits-for graph:
  - Nodes are transactions
  - There is an edge from Ti to Tj if Ti is waiting for Tj to release a lock
- Periodically check for cycles in the waits-for graph

Deadlock Detection

- **Example**

```
T1:  S(A), R(A),            S(B)
T2:             X(B),W(B)                  X(C)
T3:                            S(C), R(C)            X(A)
T4:                                             X(B)
```





Source: Ramakrishnan & Gehrke

Do the following schedules lead to deadlock?

- Build the waits-for graph for each of the following transaction schedules

```
T1: S(A)             X(D)    X(C)   C
T2:            X(A)                    X(B)
T3:      S(B)                                S(C)
```

Note: we only show locking operations for brevity! C denotes commit, all locks released

```
T1:                   S(C) S(A)           X(D)
T2: S(B)                           X(C)
T3:         S(D) X(B)
```

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 assure serializability:
  - 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”!

Source: Ramakrishnan & Gehrke (partial)

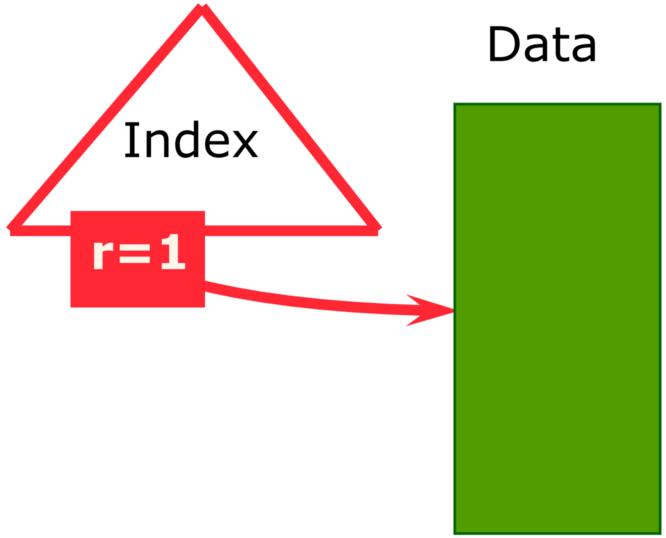
The Problem

- 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 conflict serializability guarantees serializability only if the set of objects is fixed!

Source: Ramakrishnan & Gehrke

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.



Source: Ramakrishnan & Gehrke (partial)

Multiple-Granularity Locks

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

Database
|
Tables
|
Pages
|
Tuples

contains ↓

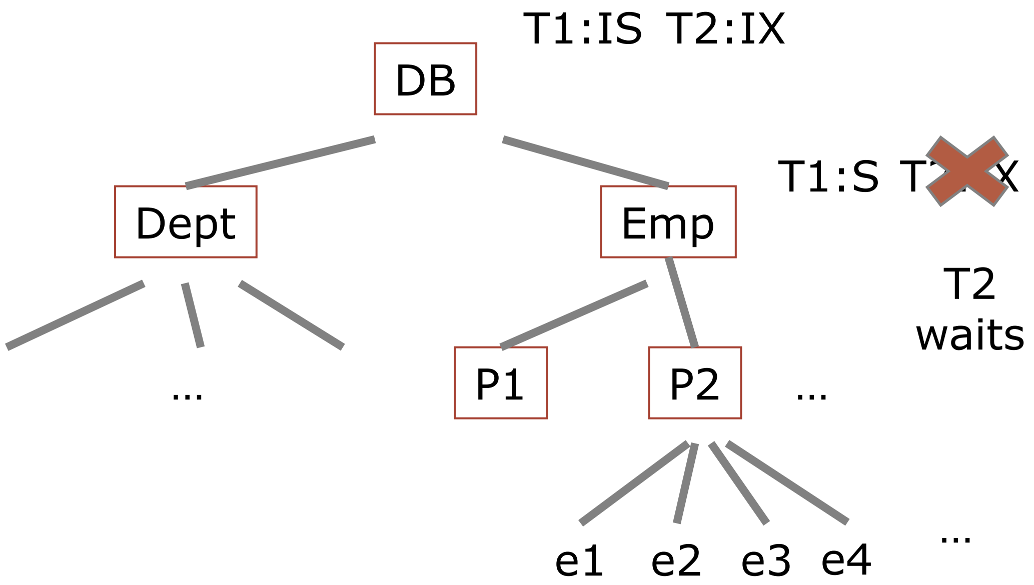
Solution: New Lock Modes, Protocol

- 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	√				

Source: Ramakrishnan & Gehrke

Examples: Multiple-Granularity Locks

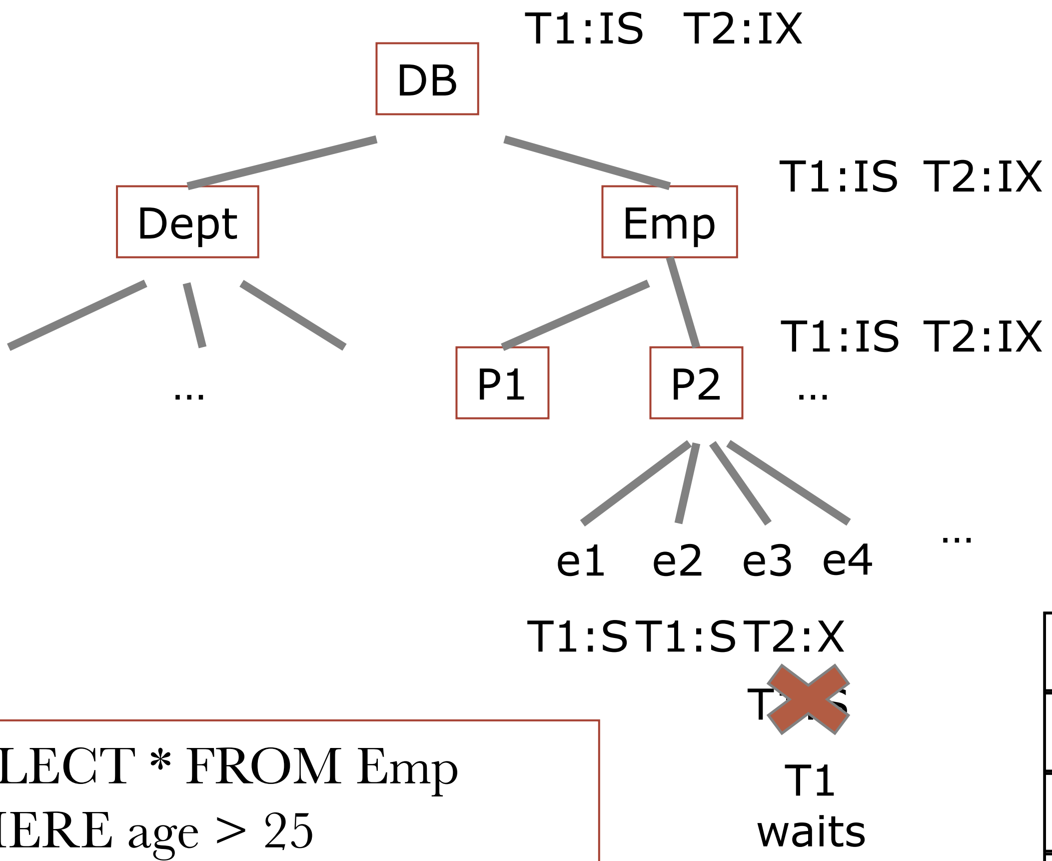


Scenario 1:
T1 scans Emp;
T2 uses indirect index, finds e3

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 starts range scan on e1, T2 finds e3

	--	IS	IX	S	X
--	√	√	√	√	√
IS	√	√	√	√	
IX	√	√	√		
S	√	√		√	
X	√				

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

Questions so far?

The Problems with Locking

- Locking is a pessimistic approach in which conflicts are prevented. Disadvantages:
  - Lock management overhead.
  - Deadlock detection/resolution.
  - Lock contention for heavily used objects.
- Remember: We must devise a way to enforce serializability, without destroying concurrency
- Two approaches:
  - Prevent violations → locking
  - Fix violations → aborts

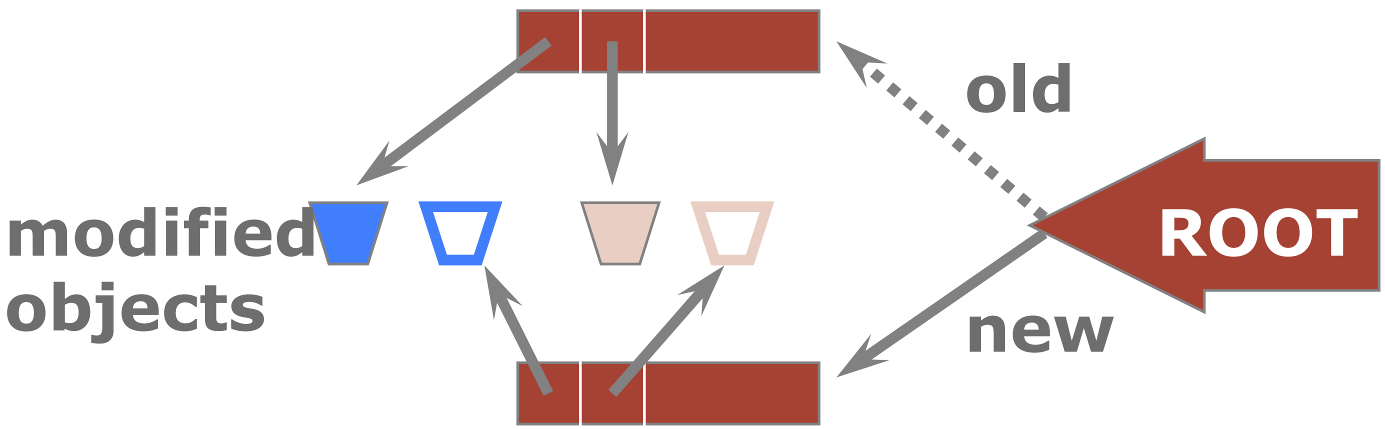
How can we design a protocol based on aborts instead of locks?

Source: Ramakrishnan & Gehrke (partial)

Optimistic CC: Kung-Robinson Model

- Xacts have three phases

- READ: Xacts read from the database, but make changes to private copies of objects.
- VALIDATE: Check for conflicts.
- WRITE: Make local copies of changes public.



Source: Ramakrishnan & Gehrke (partial)

Validation

- Test conditions that are sufficient to ensure that no conflict occurred.
- Each Xact is assigned a numeric id.
  - Just use a **timestamp**.
- Xact ids assigned at end of READ phase, just before validation begins. (Why then?)
- ReadSet(Ti): Set of objects read by Xact Ti.
- WriteSet(Ti): Set of objects modified by Ti.

Test 1

- For all i and j such that Ti < Tj, check that Ti completes before Tj begins.

Source: Ramakrishnan & Gehrke (partial)

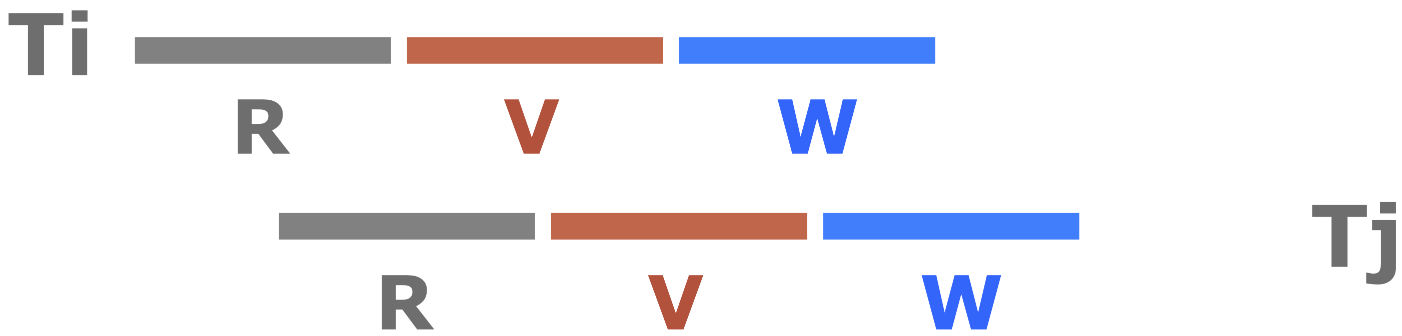
Test 2

- For all i and j such that Ti < Tj, check that:
  - Ti completes before Tj begins its Write phase **+**
  - WriteSet(Ti) $\cap$ ReadSet(Tj) is empty.

Does Tj read dirty data? Does Ti overwrite Tj's writes?

Test 3

- For all i and j such that Ti < Tj, check that:
  - Ti completes Read phase before Tj does **+**
  - WriteSet(Ti) $\cap$ ReadSet(Tj) is empty **+**
  - WriteSet(Ti) $\cap$ WriteSet(Tj) is empty.



Does Tj read dirty data? Does Ti overwrite Tj's writes?

Source: Ramakrishnan & Gehrke (partial)

Validation Example

- Predict whether T3 will be allowed to commit, given the transactions below

```
T1: RS(T1) = {1, 2, 3},    WS(T1) = {4},
      T1 completes before T3 begins with its write phase.
T2: RS(T2) = {6, 7, 8},    WS(T2) = {8},
      T2 completes read phase before T3 does.
T3: RS(T3) = {3, 5, 6, 7}, WS(T3) = {4},
      allow commit or roll back?
```

Overheads in Optimistic CC

- Must record read/write activity in ReadSet and WriteSet per Xact.
  - Must create and destroy these sets as needed.
- Must check for conflicts during validation, and must make validated writes “global”.
  - Critical section can reduce concurrency.
  - Scheme for making writes global can reduce clustering of objects.
- Optimistic CC restarts Xacts that fail validation.
  - Work done so far is wasted; requires clean-up.

- Still, optimistic techniques widely used in software transactional memory (STM)

Source: Ramakrishnan & Gehrke (partial)

Multiversion Concurrency Control (MVCC)

- This approach maintains a number of **versions** of a data item and allocates the **right version to a read operation** of a transaction. Thus unlike other mechanisms a **read operation in this mechanism is never rejected**.
- Side effect:
  - Significantly more storage (RAM and disk) is required to maintain multiple versions. To check unlimited growth of versions, a **garbage collection** is run when some criteria is satisfied
- Many commercial database systems implement a combination of MVCC and S2PL
- See compendium for more details

Source: Elmasri & Navathe, 2007 (partial)

Snapshot Isolation

- Often databases implement properties that are **weaker** than serializability
- **Snapshot isolation**
  - **Snapshots:** Transactions see snapshot as of beginning of their execution
  - **First Committer Wins:** Conflicting writes to same item lead to aborts
- May lead to **write skew**
  - Database must have at least one doctor on call
  - Two doctors on call concurrently examine snapshot and see exactly each other on call
  - Doctors update their own records to being on leave
    - No write-write conflicts: different records!
  - After commits, database has no doctors on call

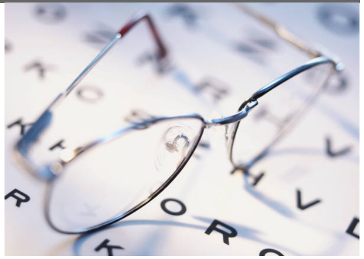
Transaction Support in SQL-92

- Each transaction has an access mode, a diagnostics size, and an isolation level.

Does not correspond to serializability!

Isolation Level	**Dirty Read**	**Unrepeatable Read**	**Phantom Problem**
Read Uncommitted	Maybe	Maybe	Maybe
Read Committed	No	Maybe	Maybe
Repeatable Reads	No	No	Maybe
Serializable	No	No	No

Source: Ramakrishnan & Gehrke



What should we learn today?

- Explain deadlock prevention and detection techniques
- Apply deadlock detection using a waits-for graph to transaction schedules
- Explain situations where predicate locking is required
- Explain the optimistic concurrency control and multi-version concurrency control models
- Predict validation decisions under optimistic concurrency control