Project5

Concurrency Control

Project Hierarchy

- Your project hierarchy should be like this.
  - Your_hconnect_repo
    - project5
      - include/
      - lib/
      - Makefile
      - src/
- If your Makefile doesn't make libbpt.a library file at the exact path, you'll get zero score. (your_hconnect_repo/project5/lib/libbpt.a)

Lock Manager

- Your database system is not yet supporting transaction.
- Implement **transaction** concept that can support '**I**solation' and '**C**onsistency' using your own lock manager (lock table).
- Your lock manager should provide:
  - Conflict-serializable schedule for transactions
  - Strict-2PL
  - Deadlock detection (abort the transaction if detected)
  - Record-level locking with Shared(S)/Exclusive(X) mode

Project Specification

- Your library should provide two APIs below for transaction operations.

- ***int trx_begin(void)***
  - Allocate a transaction structure and initialize it.
  - Return a unique **transaction id (>= 1)** if success, otherwise return 0.
  - Note that the transaction id should be unique for each transaction, that is, you need to allocate a transaction id holding a mutex.

- ***int trx_commit(int trx_id)***
  - Clean up the transaction with given trx_id (transaction id) and its related information that has been used in your lock manager. (Shrinking phase of strict 2PL)
  - Return the completed transaction id if success, otherwise return 0.

Project Specification

- Also, your library should provide two APIs below for database operations, that can be wrapped in a transaction.

- ***int db_find(int table_id, int64_t key, char* ret_val, int trx_id)***
  - Read a value in the table with matching key for the transaction having ***trx_id***.
  - return 0 (SUCCESS): operation is successfully done and the transaction can continue the next operation.
  - return non-zero (FAILED): operation is failed (e.g., deadlock detected) and the transaction should be aborted. Note that all tasks that need to be handled (e.g., **releasing the locks that are held by this transaction**, **rollback of previous operations**, etc…) should be completed in db_find().

- ***int db_update(int table_id, int64_t key, char* values, int trx_id)***
  - Find the matching key and modify the values.
  - return 0 (SUCCESS): operation is successfully done and the transaction can continue the next operation.
  - return non-zero (FAILED): operation is failed (e.g., deadlock detected) and the transaction should be aborted. Note that all tasks that need to be handled (e.g., **releasing the locks that are held on this transaction**, **rollback of previous operations**, etc…) should be completed in db_update().

Project Specification

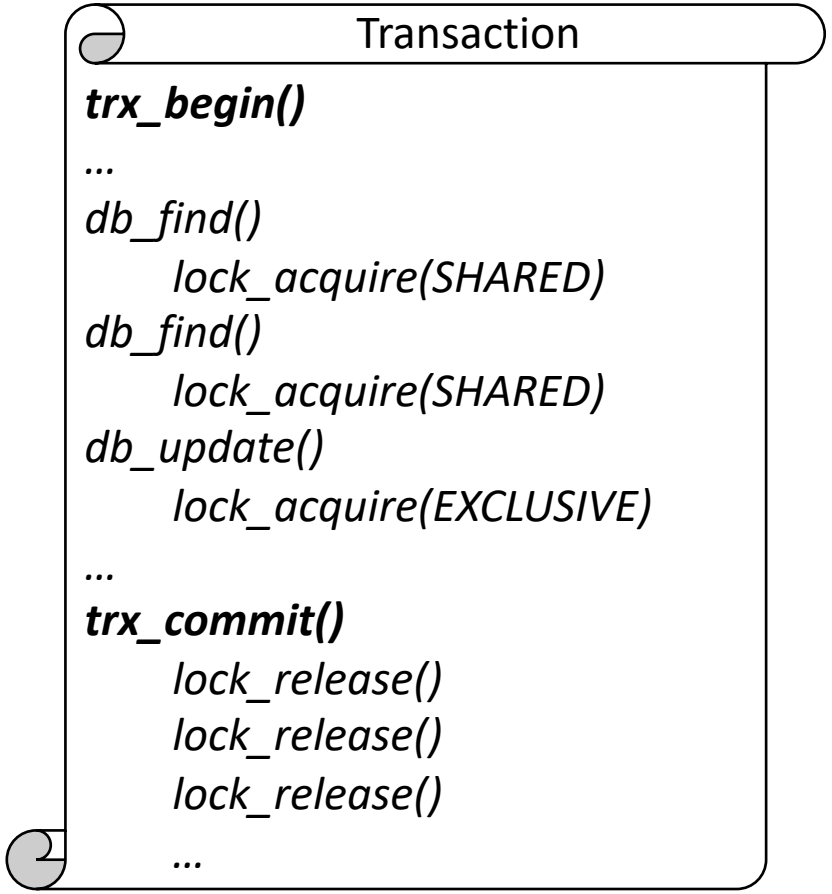
- Note that, in this project, you don't have to support *db_insert()* or *db_delete()* working **in a transaction** that may require structure modifications on b+ tree.

- We will first populate the database with *db_insert()*, or open a sample database file, and then run transactions in our test.

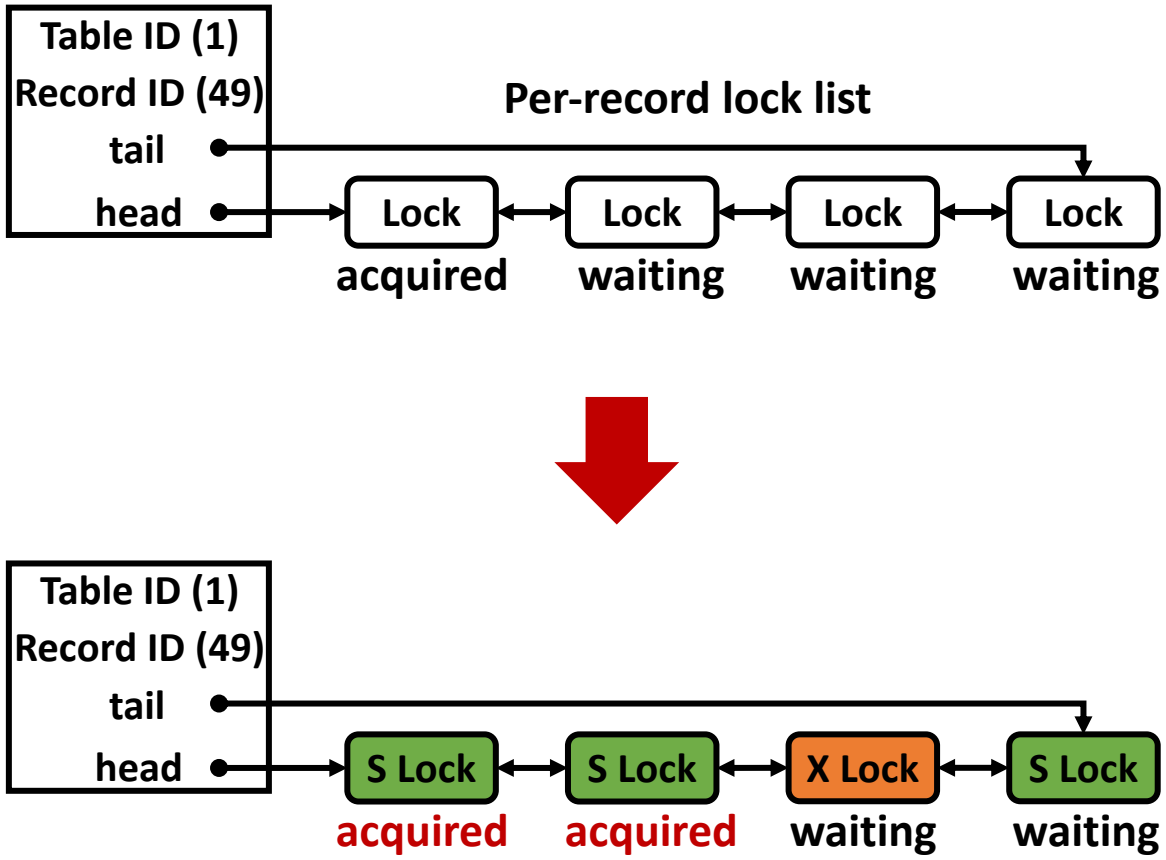
APIs for Lock Table Module

- Your lock module's APIs would like below. Use them appropriately in your database operation functions.
  - ➢ It is accepted to change the APIs (return type, parameters, etc.) of the lock table module if you want.
- ***int init_lock_table(void)***
  - Initialize any data structures required for implementing lock table, such as hash table, lock table latch, etc.
  - If success, return 0. Otherwise, return non-zero value.
- ***lock_t* lock_acquire(int table_id, int64_t key, int trx_id, int lock_mode)***
  - Allocate and append a new lock object to the lock list of the record having the *key.*
    - If there is a predecessor's conflicting lock object in the lock list, **sleep** until the predecessor to release its lock.
    - If there is no predecessor's conflicting lock object, return the address of the new lock object.
  - If an error occurs, return NULL.
  - ***lock_mode: 0 (SHARED) or 1 (EXCLUSIVE)***
- ***int lock_release(lock_t* lock_obj)***
  - Remove the *lock_obj* from the lock list.
    - If there is a successor's lock waiting for the transaction releasing the lock, **wake up** the successor.
  - If success, return 0. Otherwise, return non-zero value.

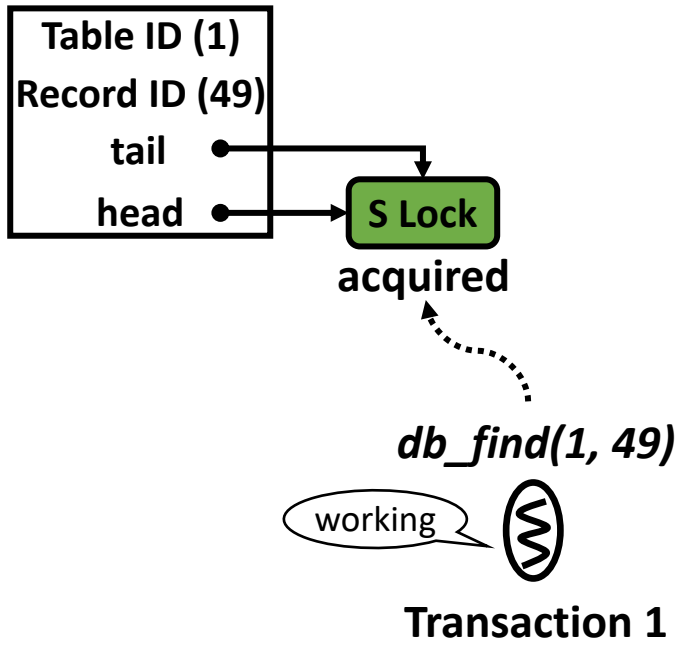
Transaction Example



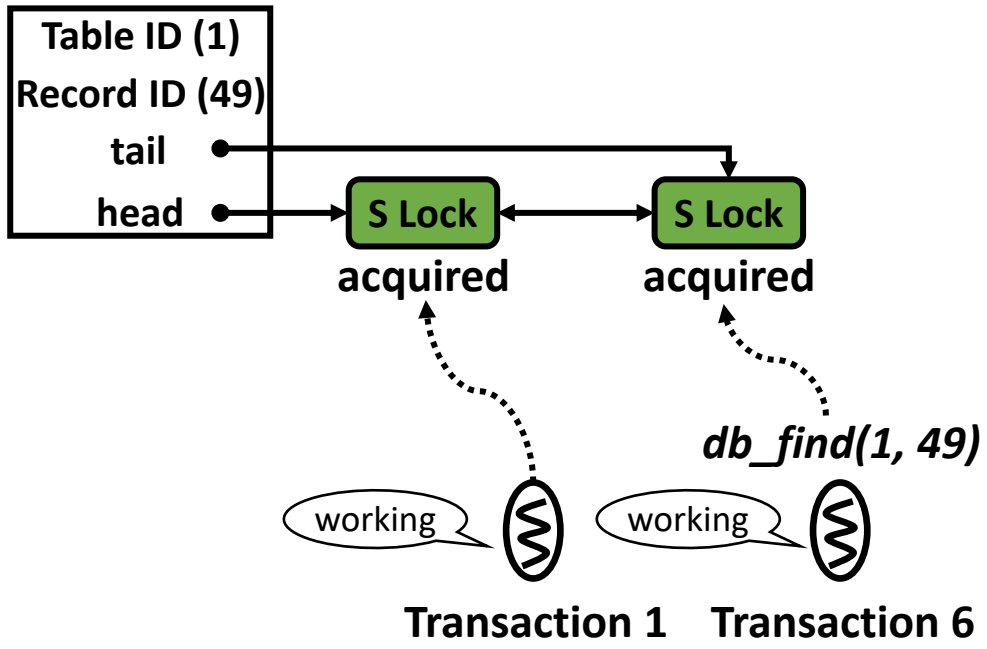
Shared / Exclusive Lock



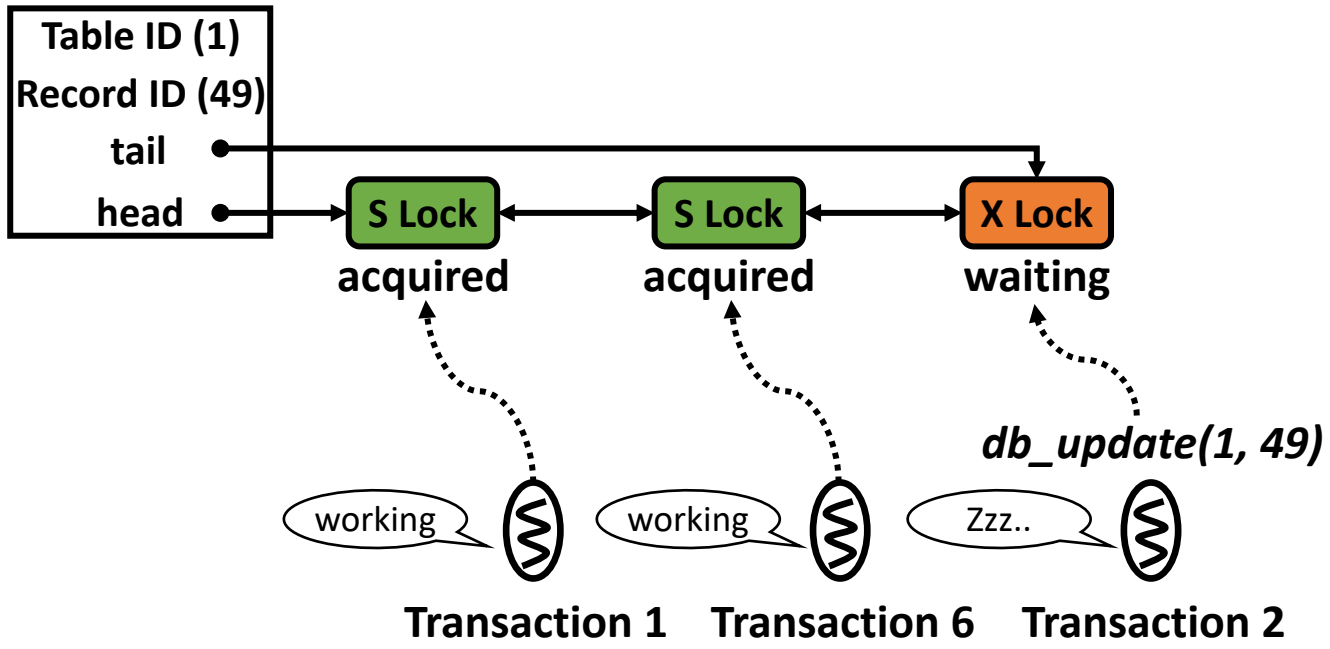
Shared / Exclusive Lock



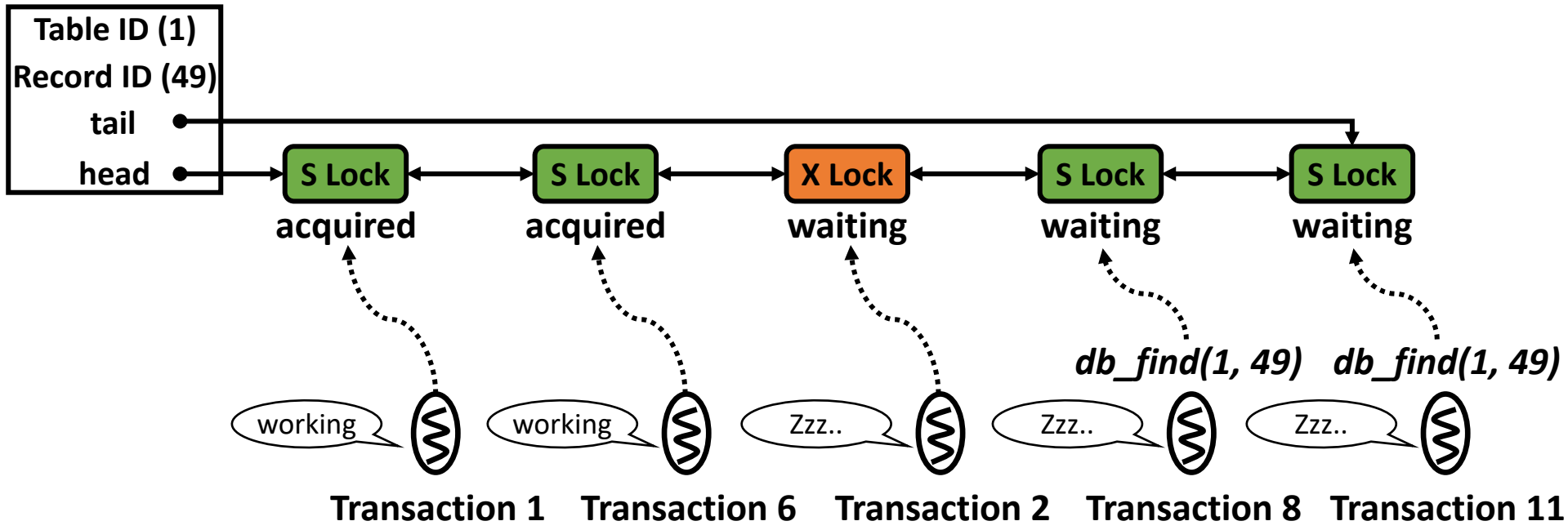
Shared / Exclusive Lock



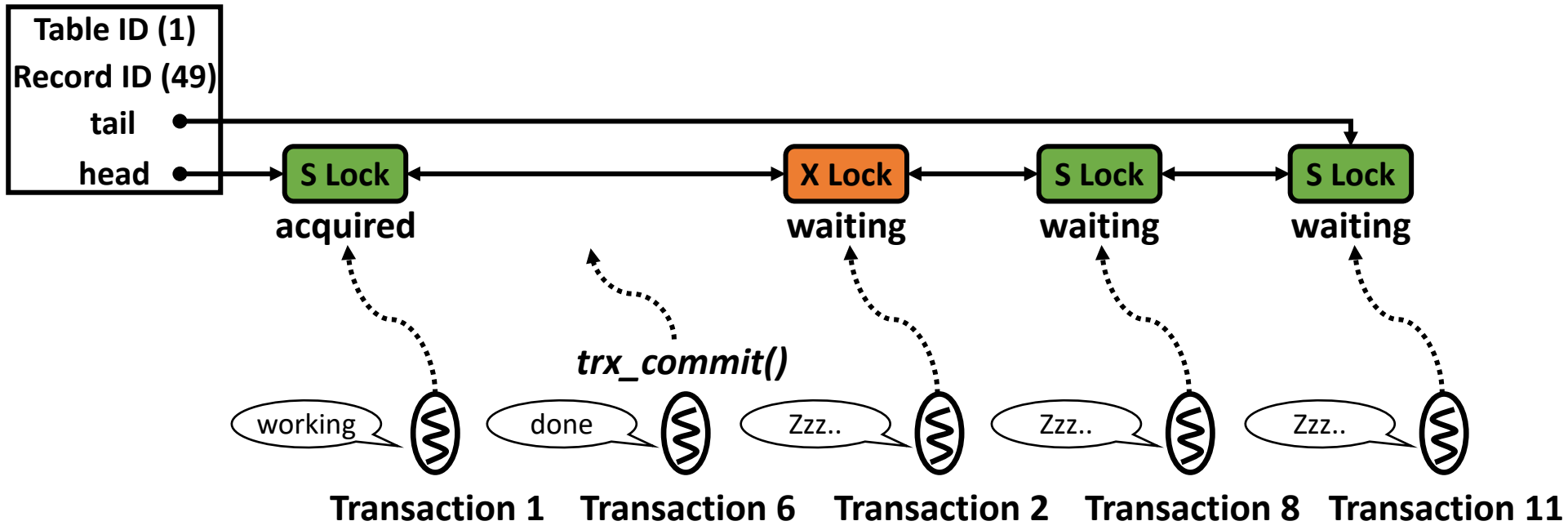
Shared / Exclusive Lock



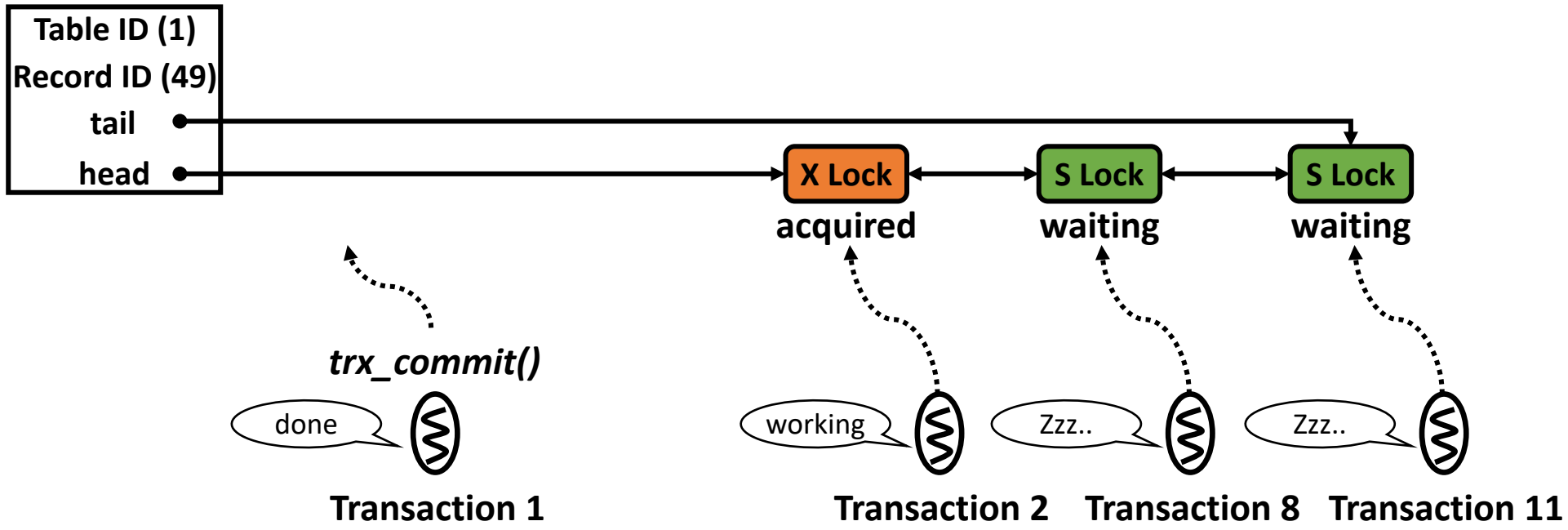
Shared / Exclusive Lock



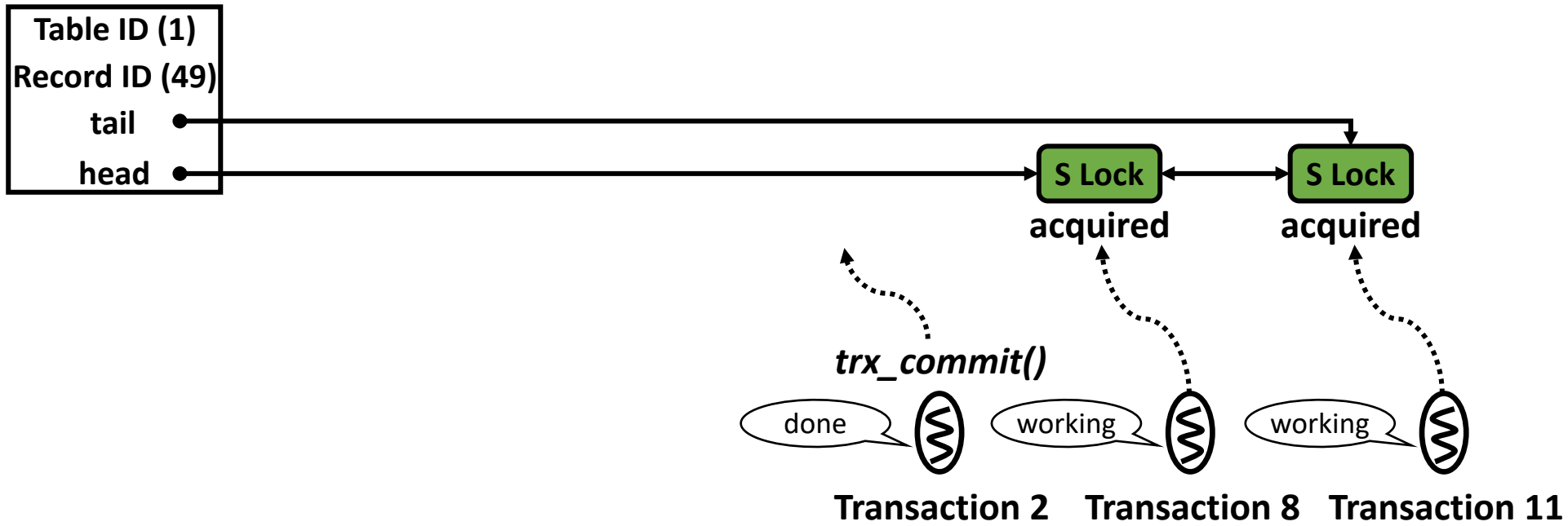
Shared / Exclusive Lock



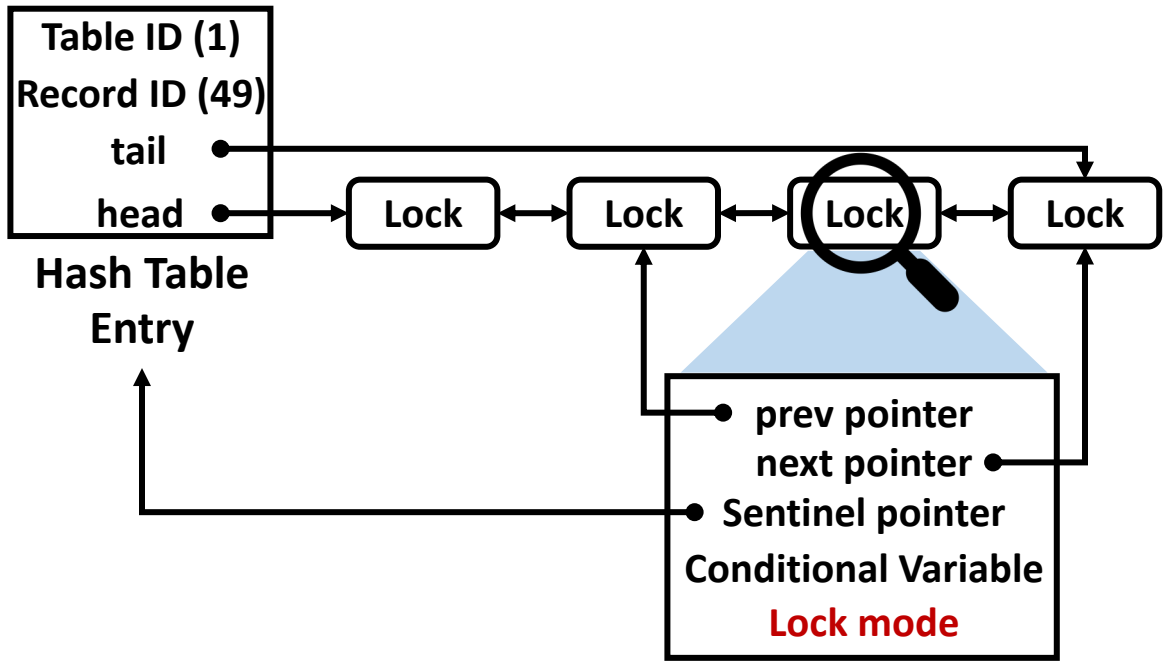
Shared / Exclusive Lock



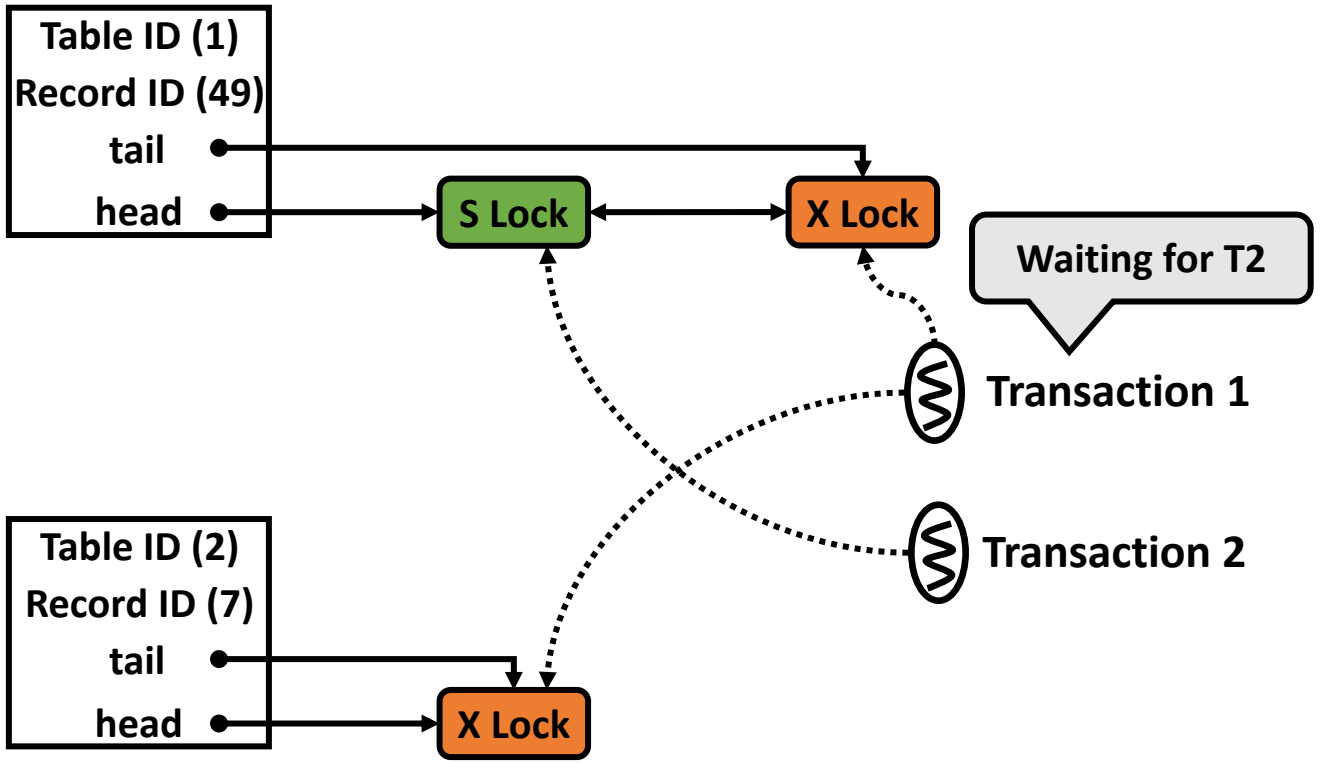
Shared / Exclusive Lock



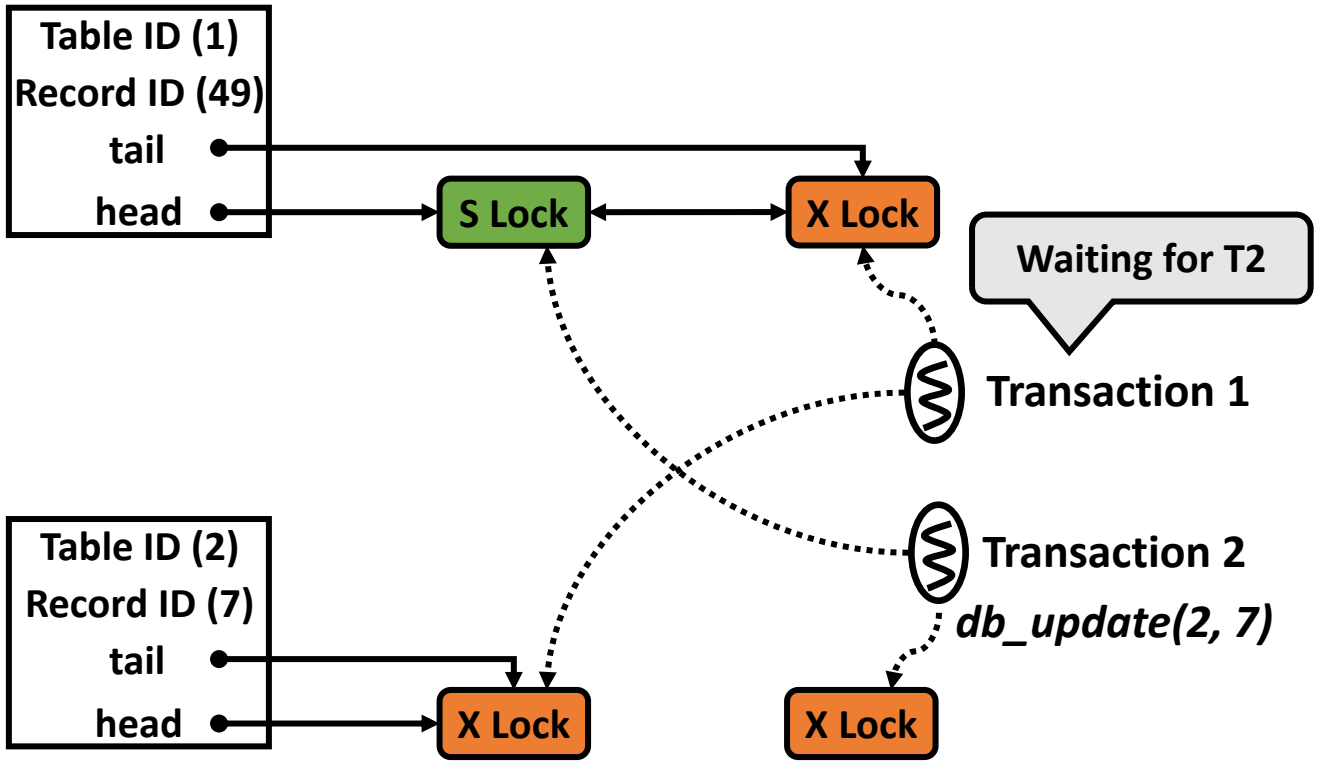
Shared / Exclusive Lock



Deadlock



Deadlock



Deadlock

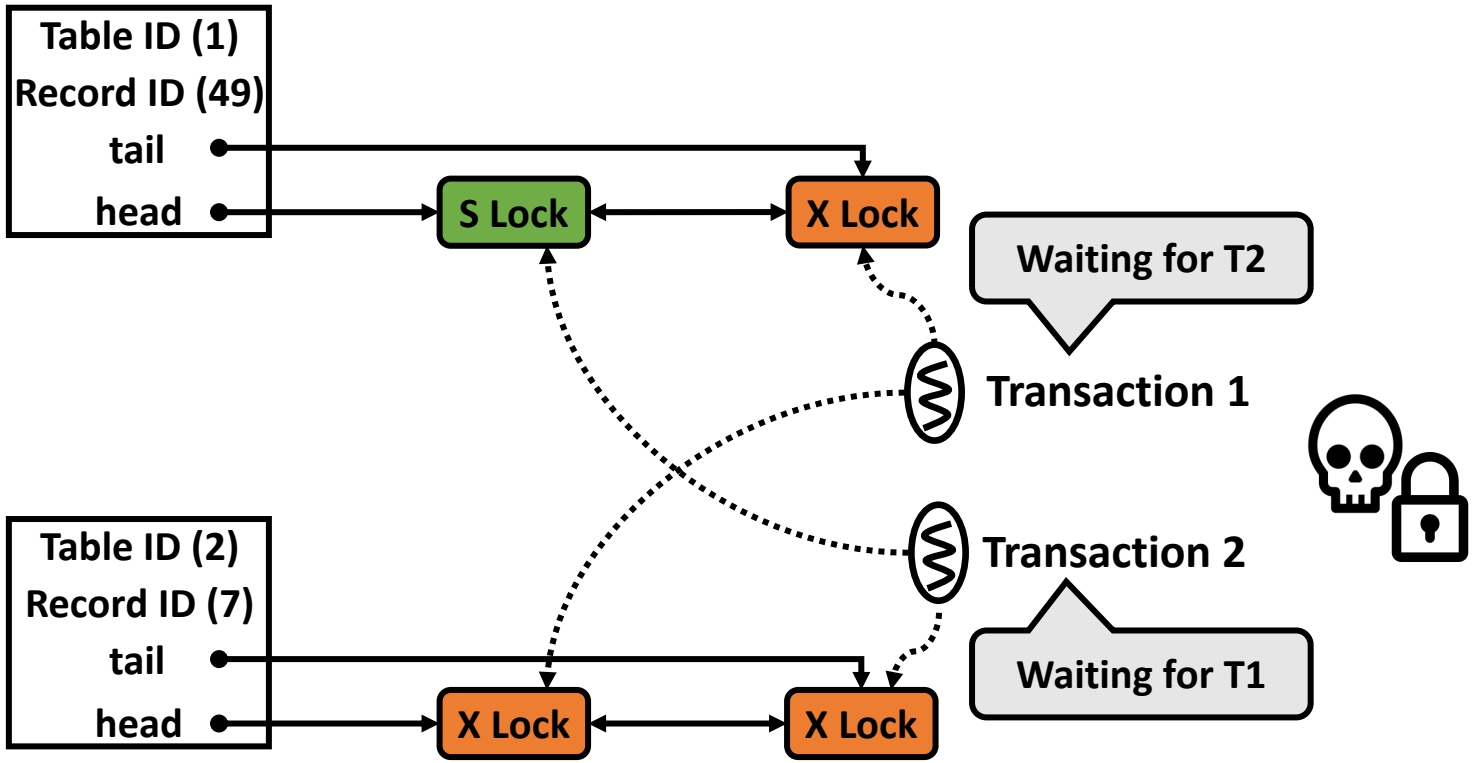


Table ID (1)
Record ID (49)
tail
head
S Lock
X Lock
Waiting for T2
Transaction 1
Transaction 2
Waiting for T1
Table ID (2)
Record ID (7)
tail
head
X Lock
X Lock

Deadlock

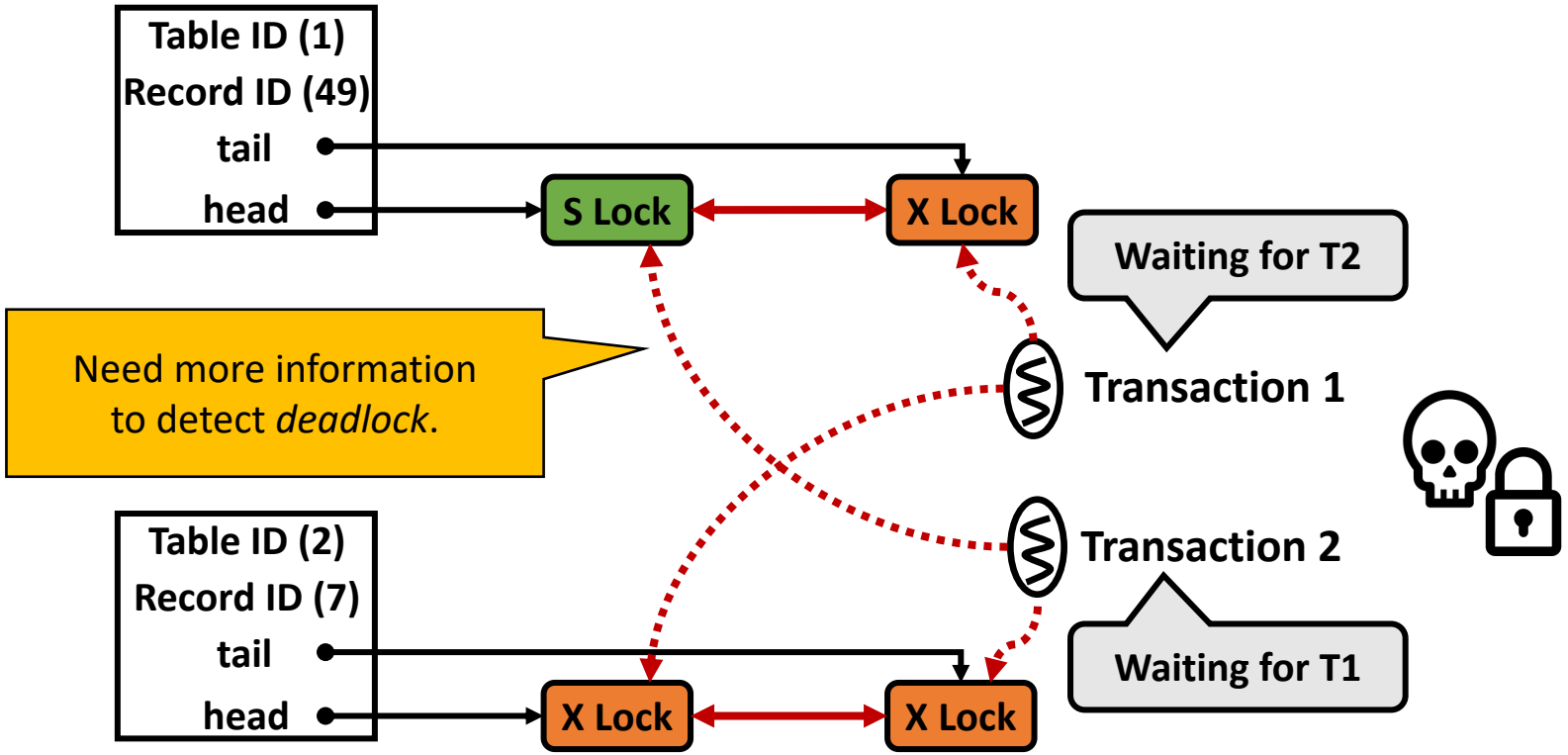
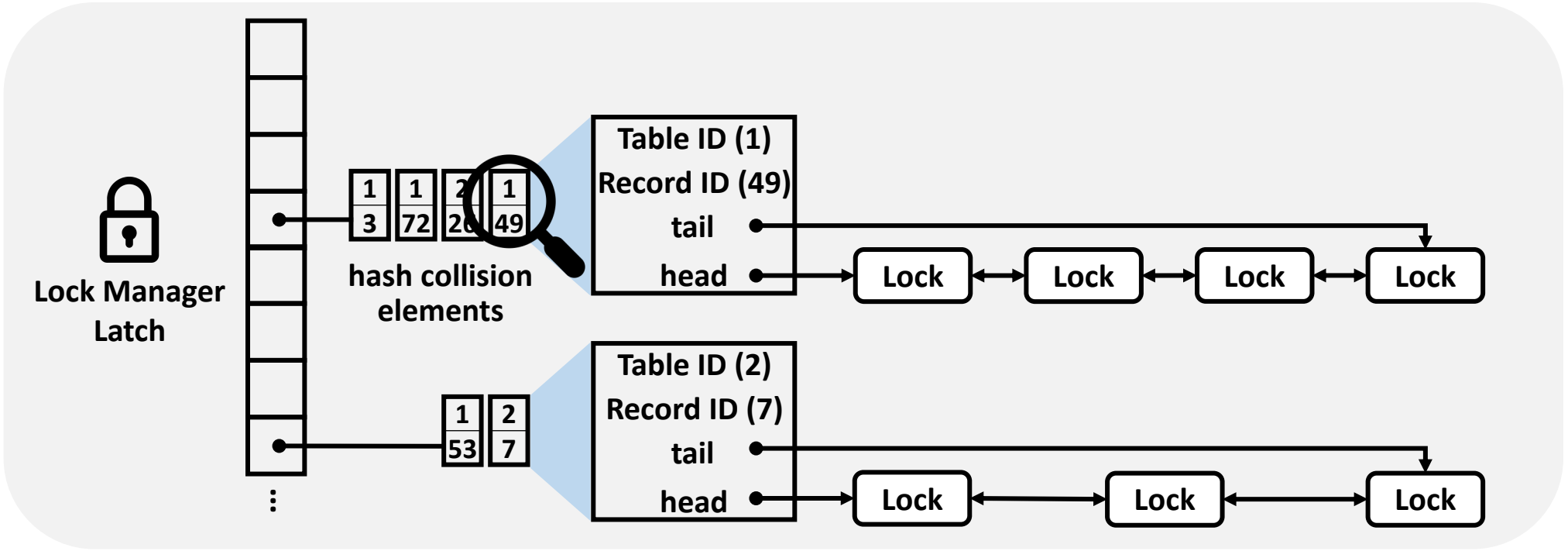
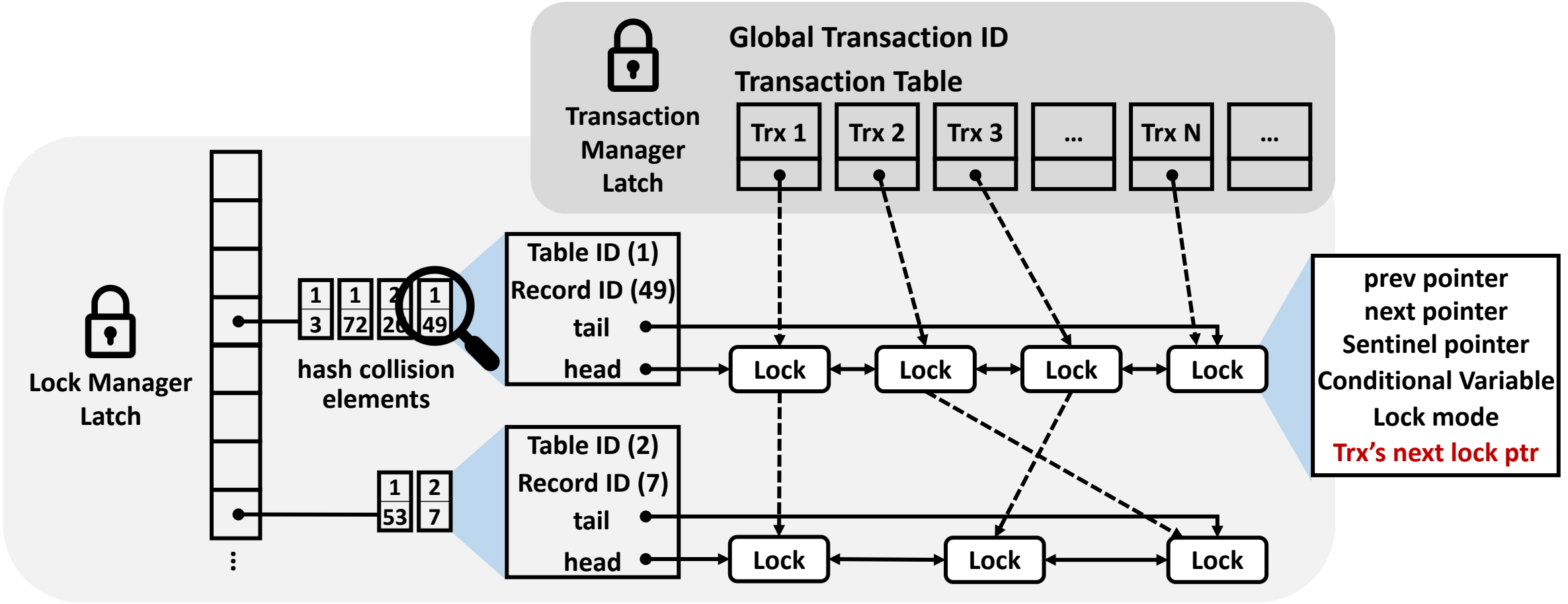


Table ID (1)
Record ID (49)
tail
head
S Lock
X Lock
Waiting for T2
Need more information
to detect *deadlock*.
Transaction 1
Transaction 2
Table ID (2)
Record ID (7)
tail
head
X Lock
X Lock
Waiting for T1

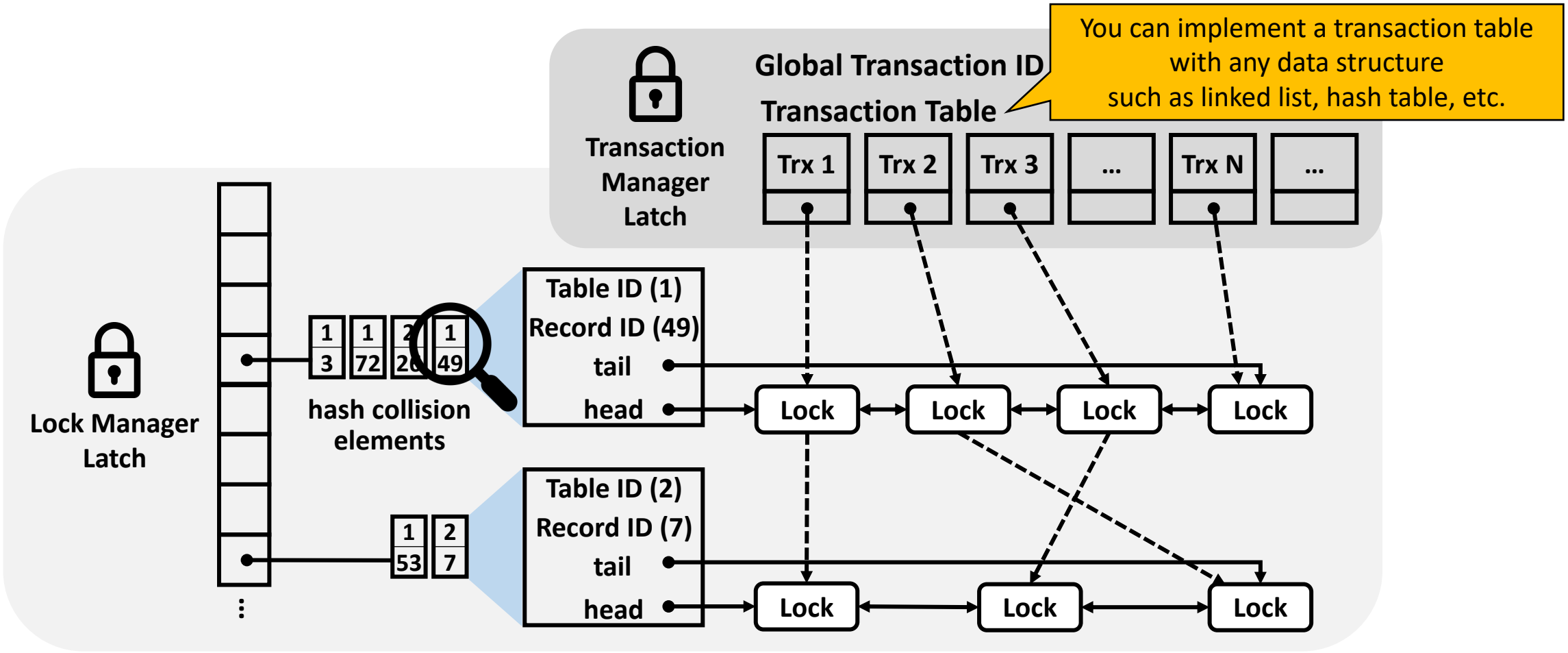
Lock Manager



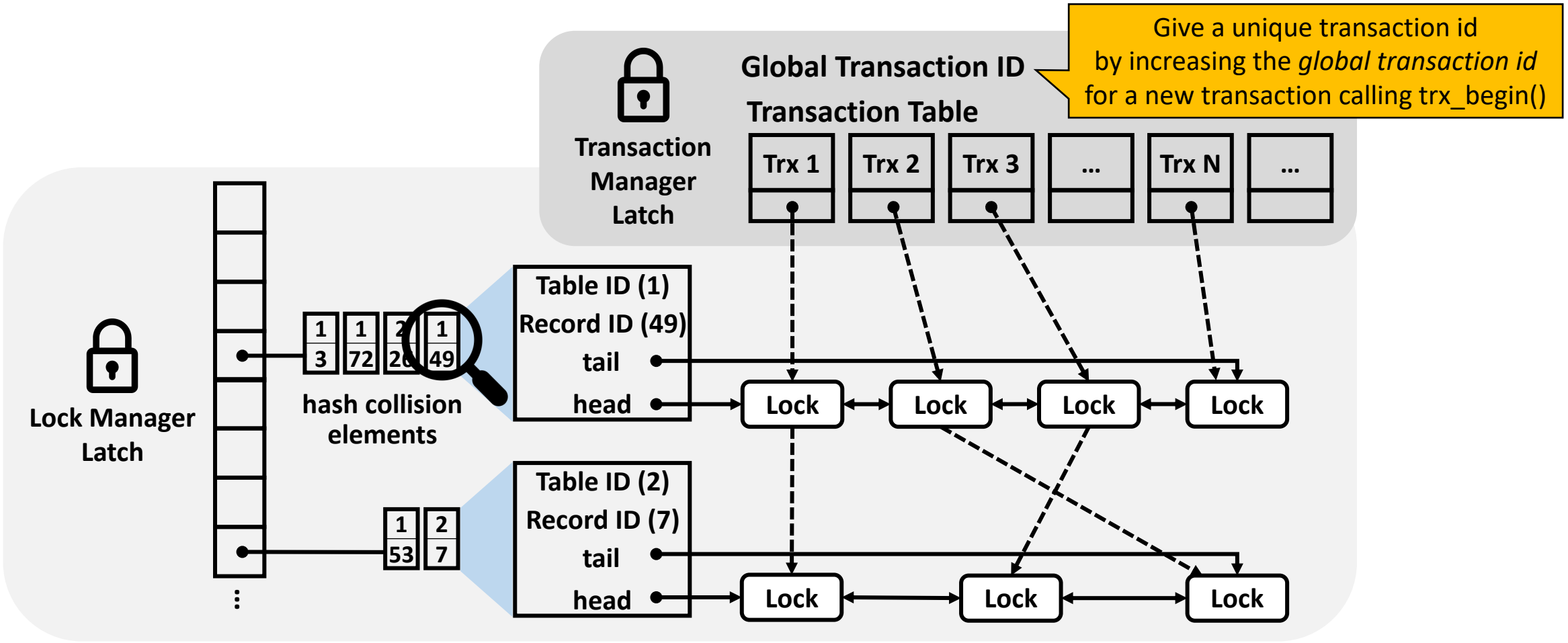
Transaction Manager



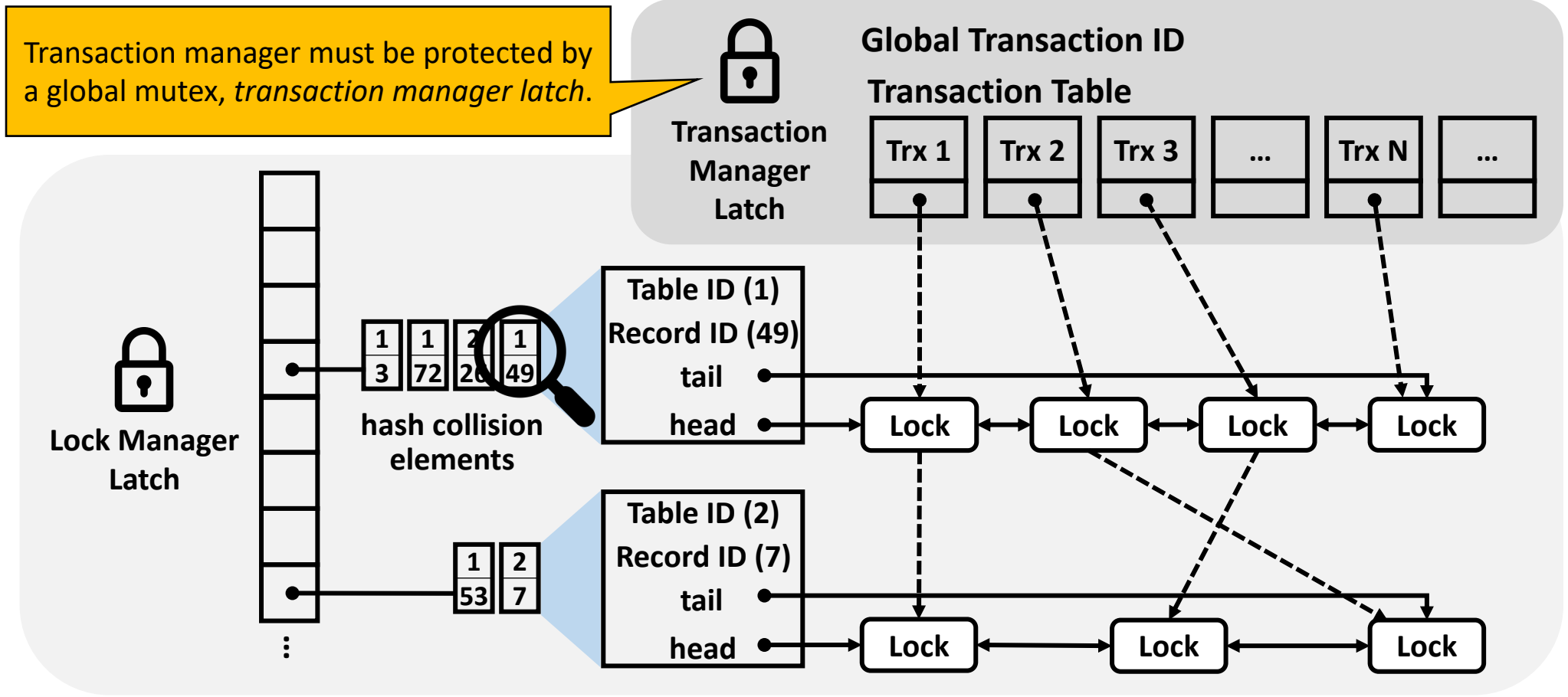
Transaction Manager



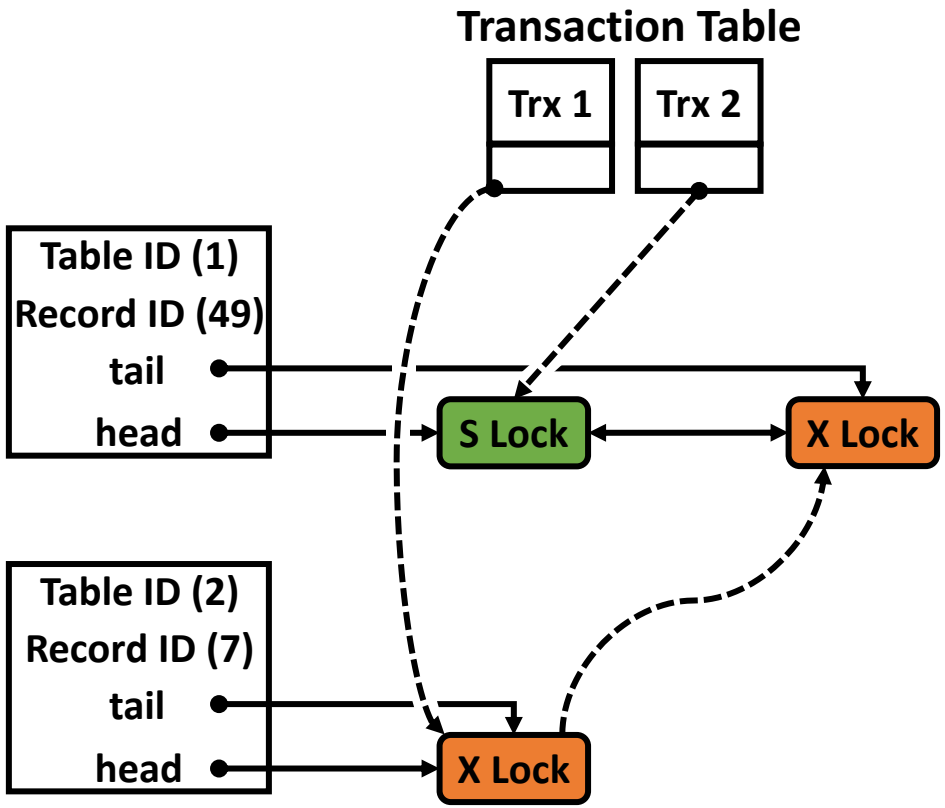
Transaction Manager



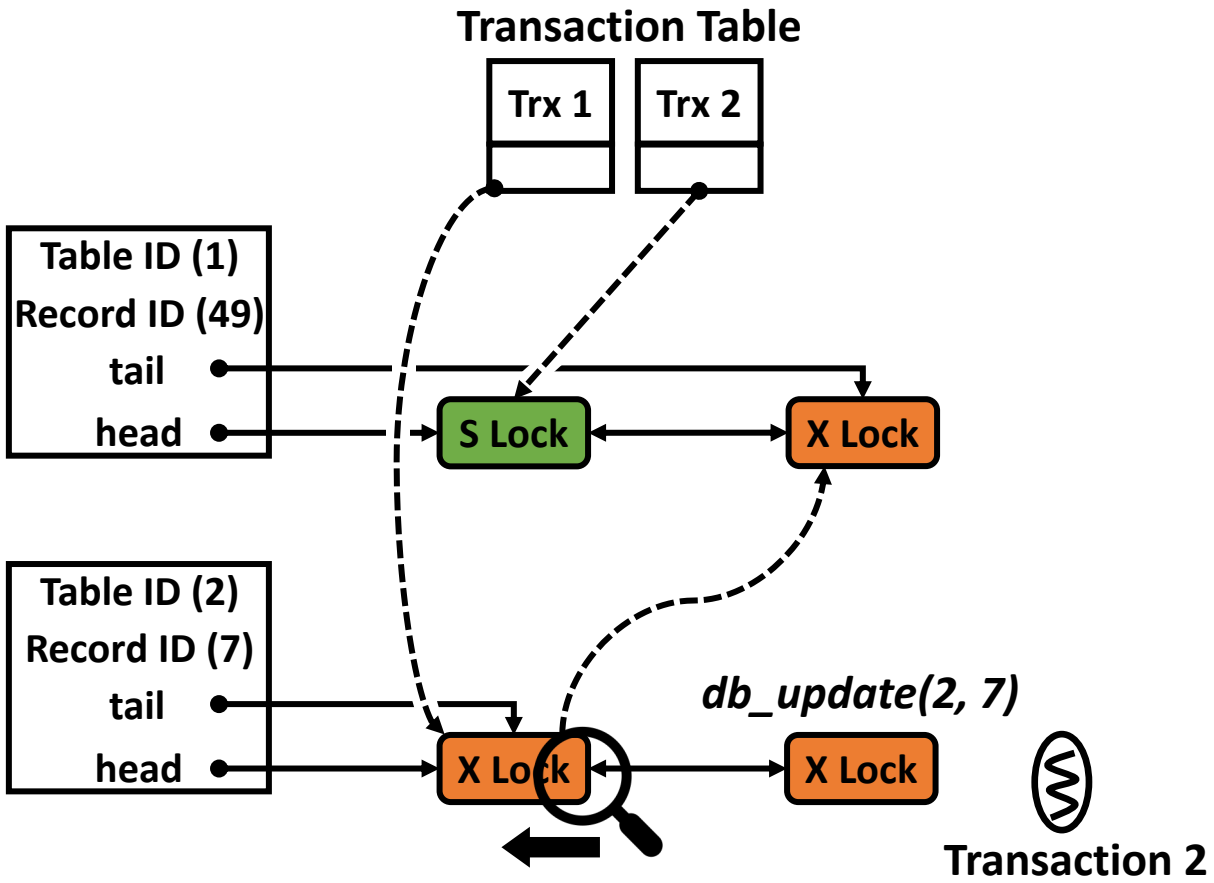
Transaction Manager



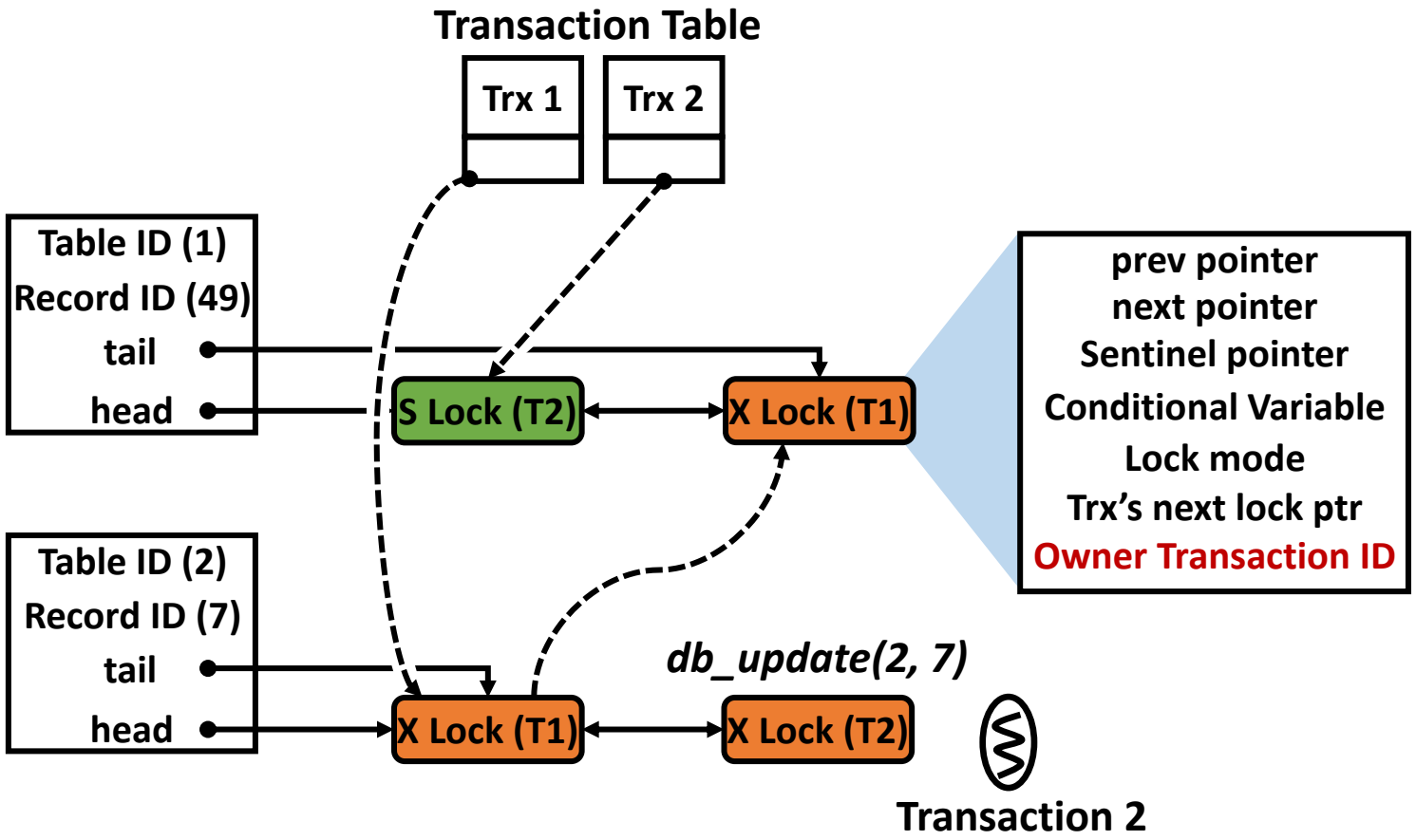
Deadlock Detection



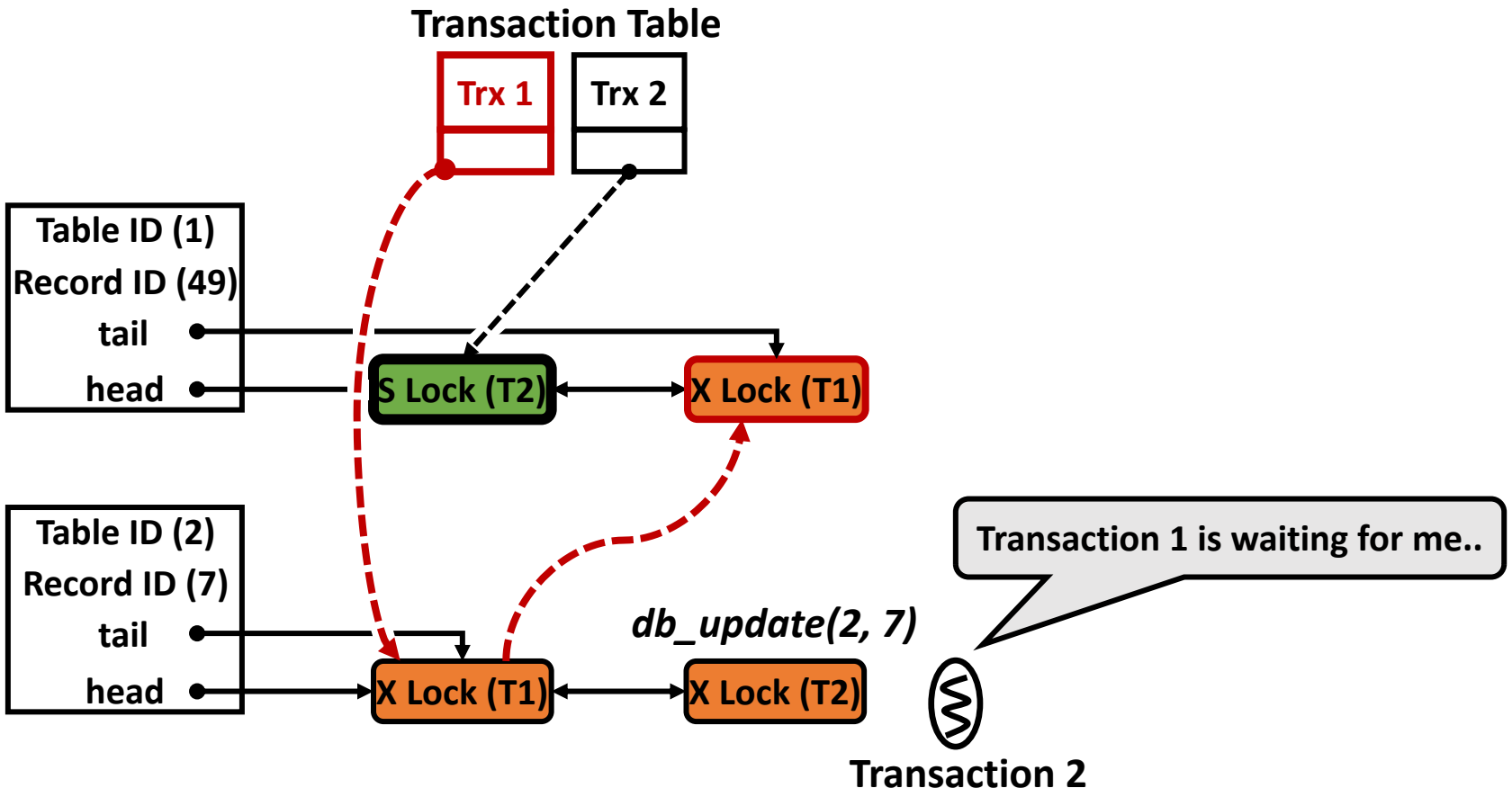
Deadlock Detection



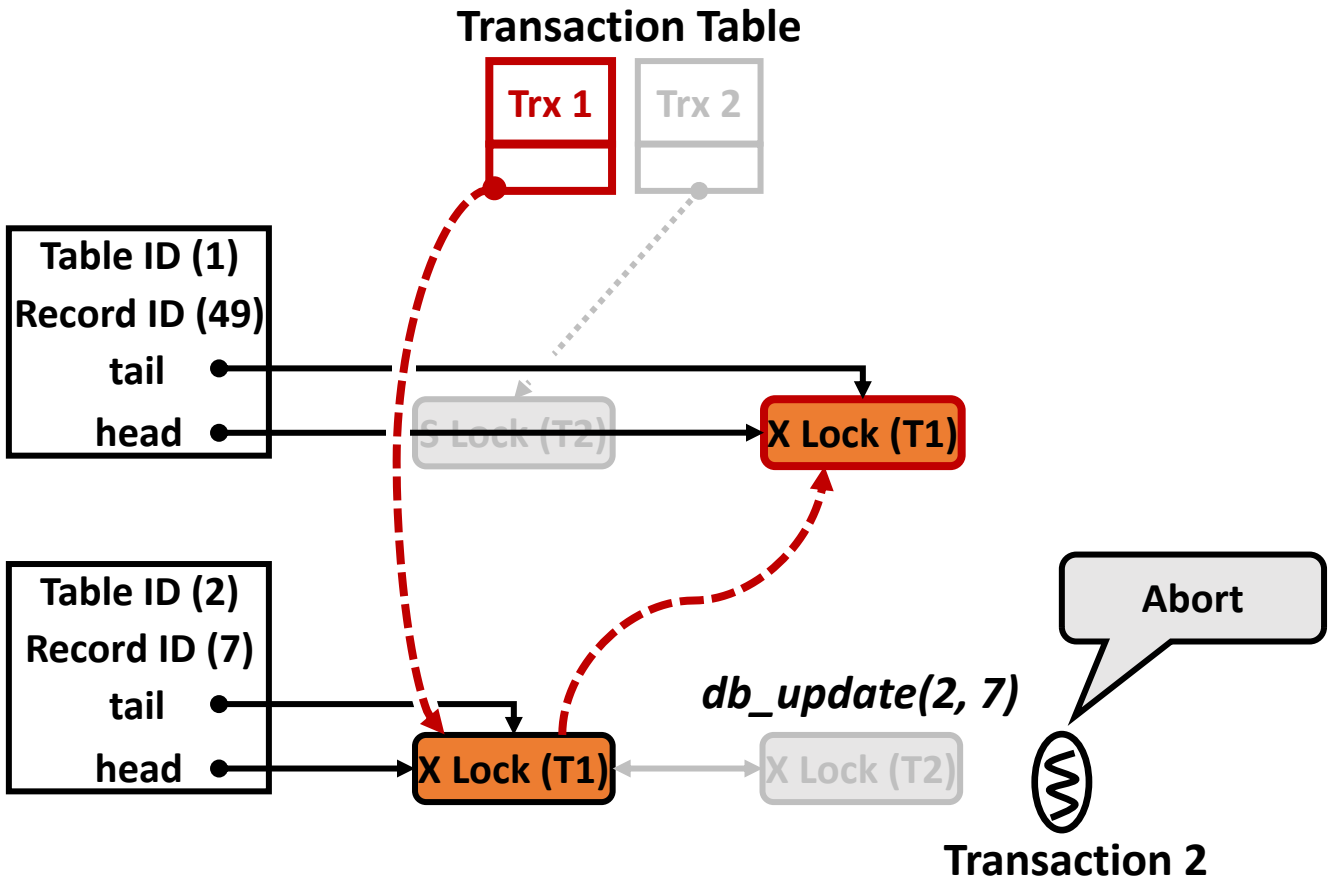
Deadlock Detection



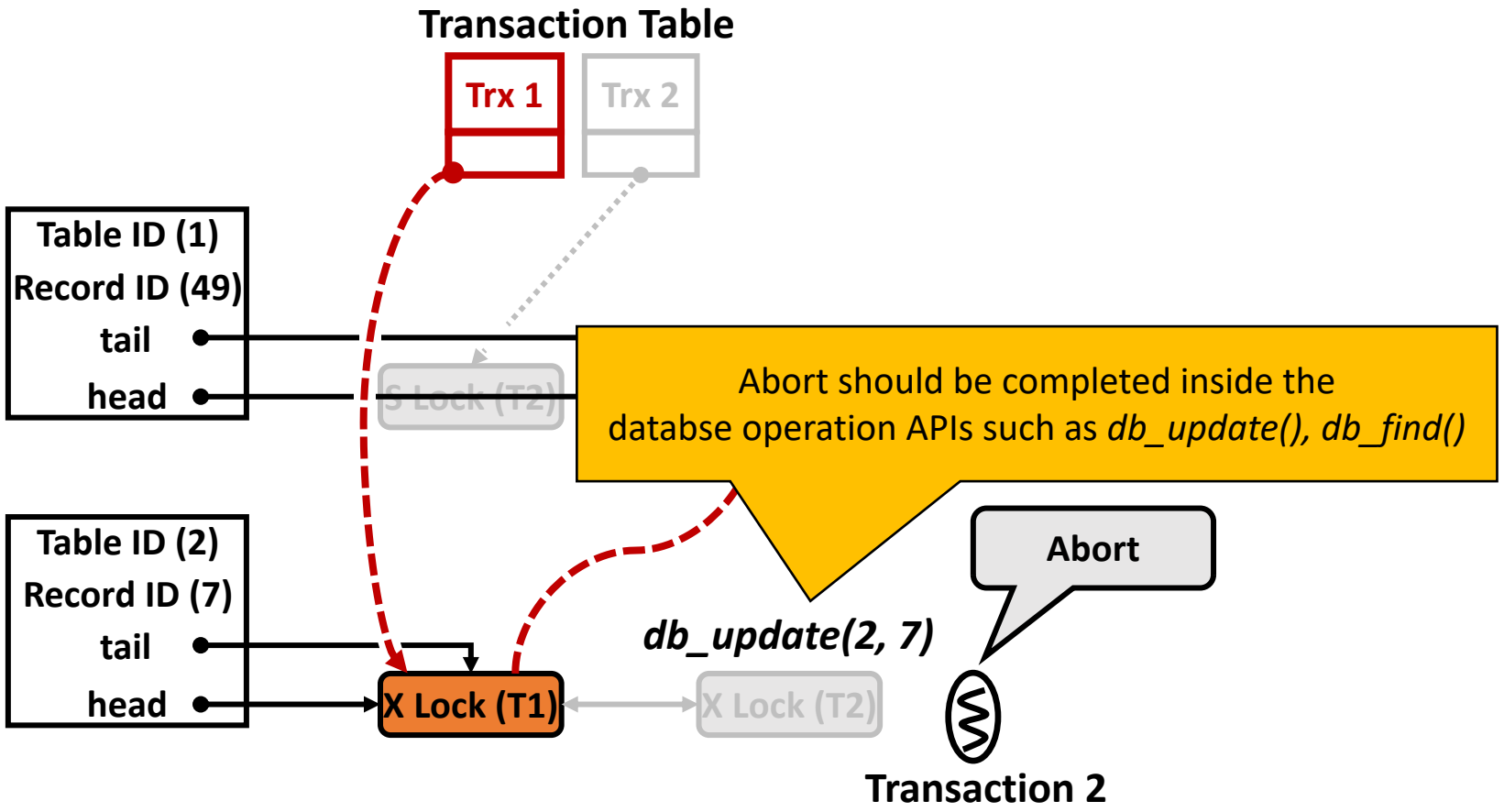
Deadlock Detection



Deadlock Detection

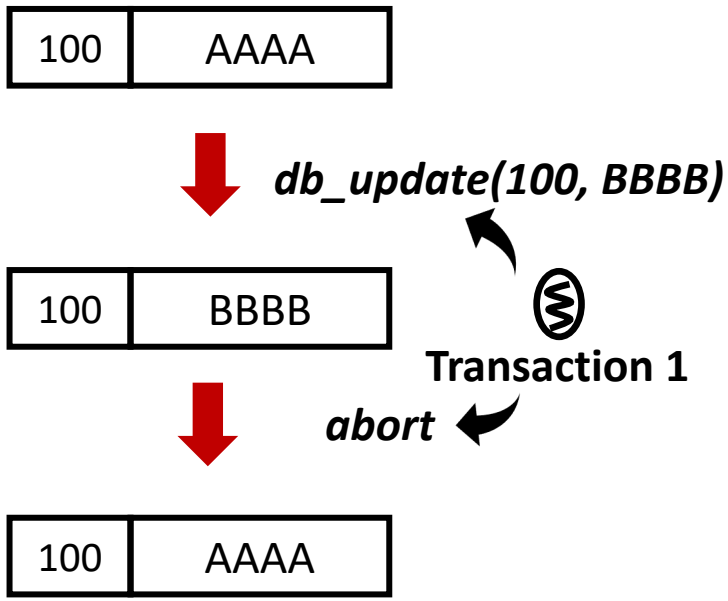


Deadlock Detection

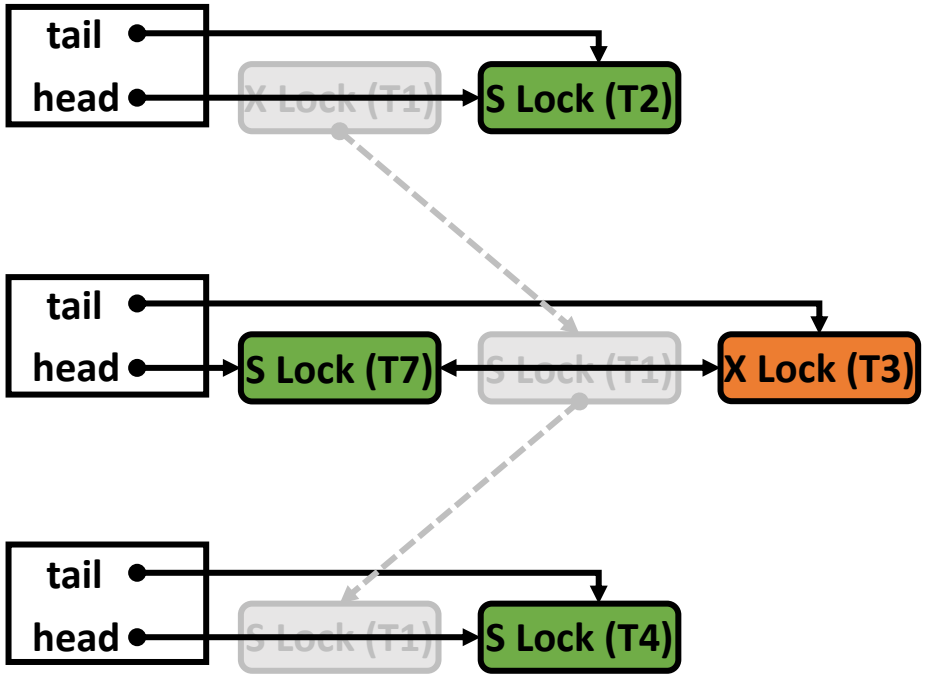


Transaction Abort

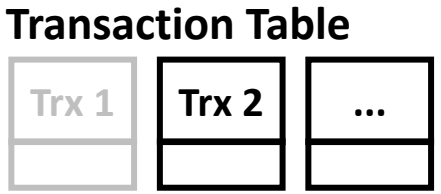
1.
Undo all modified records by the transaction



2.
Release all acquired lock objects

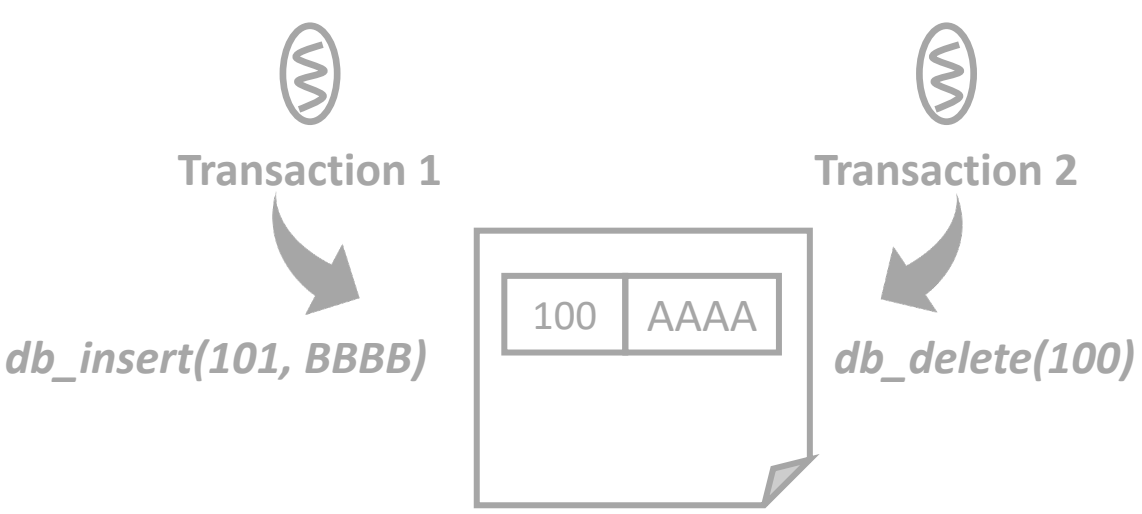


3.
Remove the transaction table entry

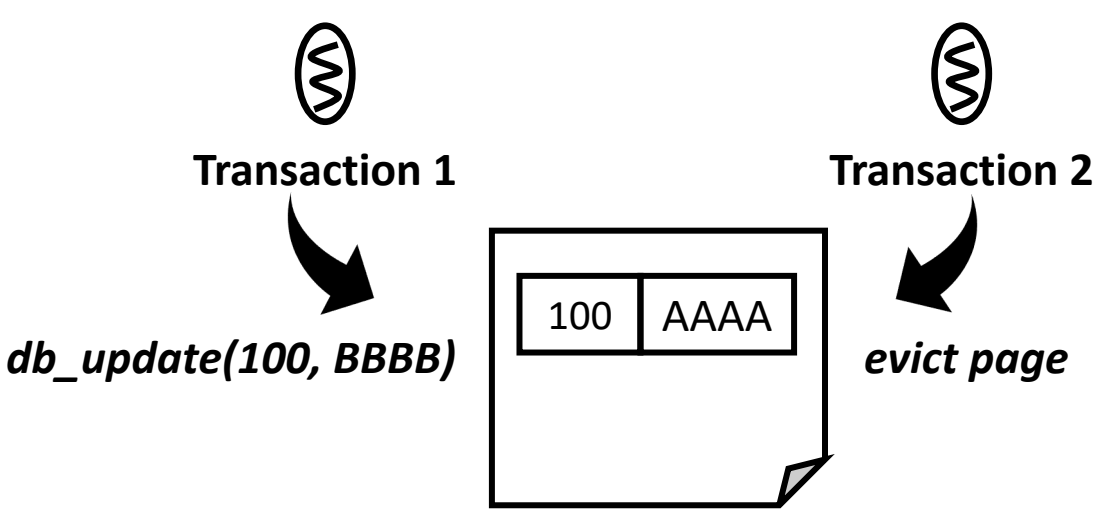


Buffer Manager Issues

- Your buffer manager must correctly work under concurrent accesses by multiple transactions.



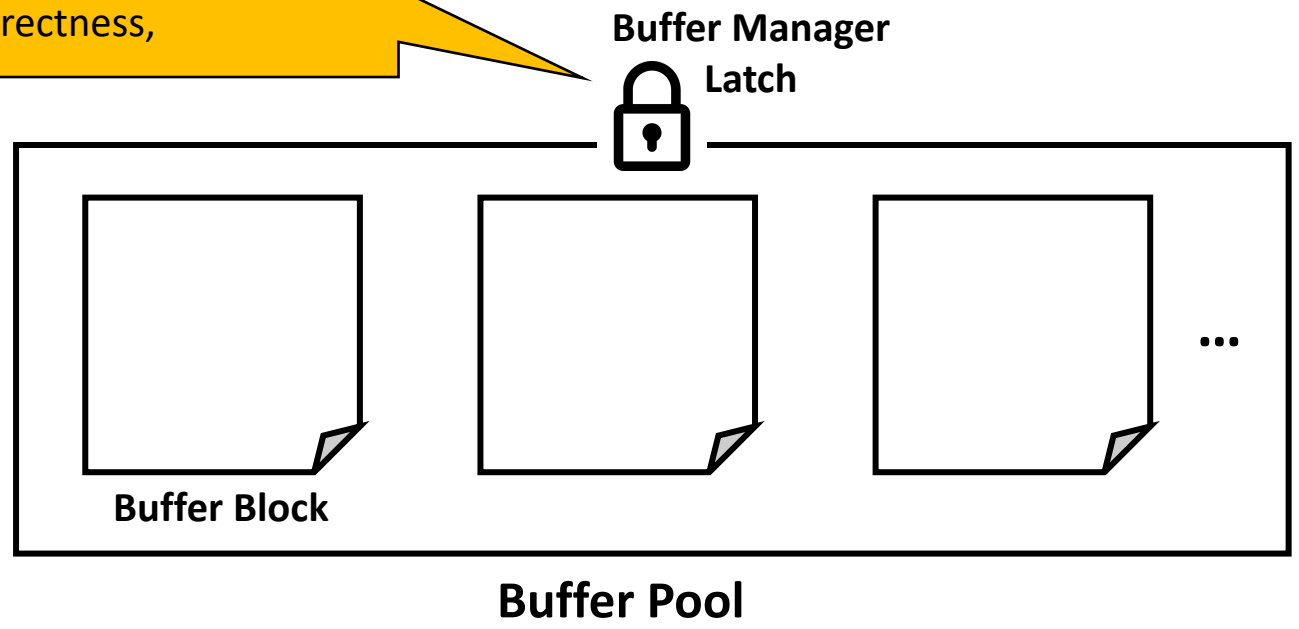
We do not have to consider concurrent db_delete() or db_insert() operations in this project.



However, we still have to deal with concurrent accesses by multiple transactions.

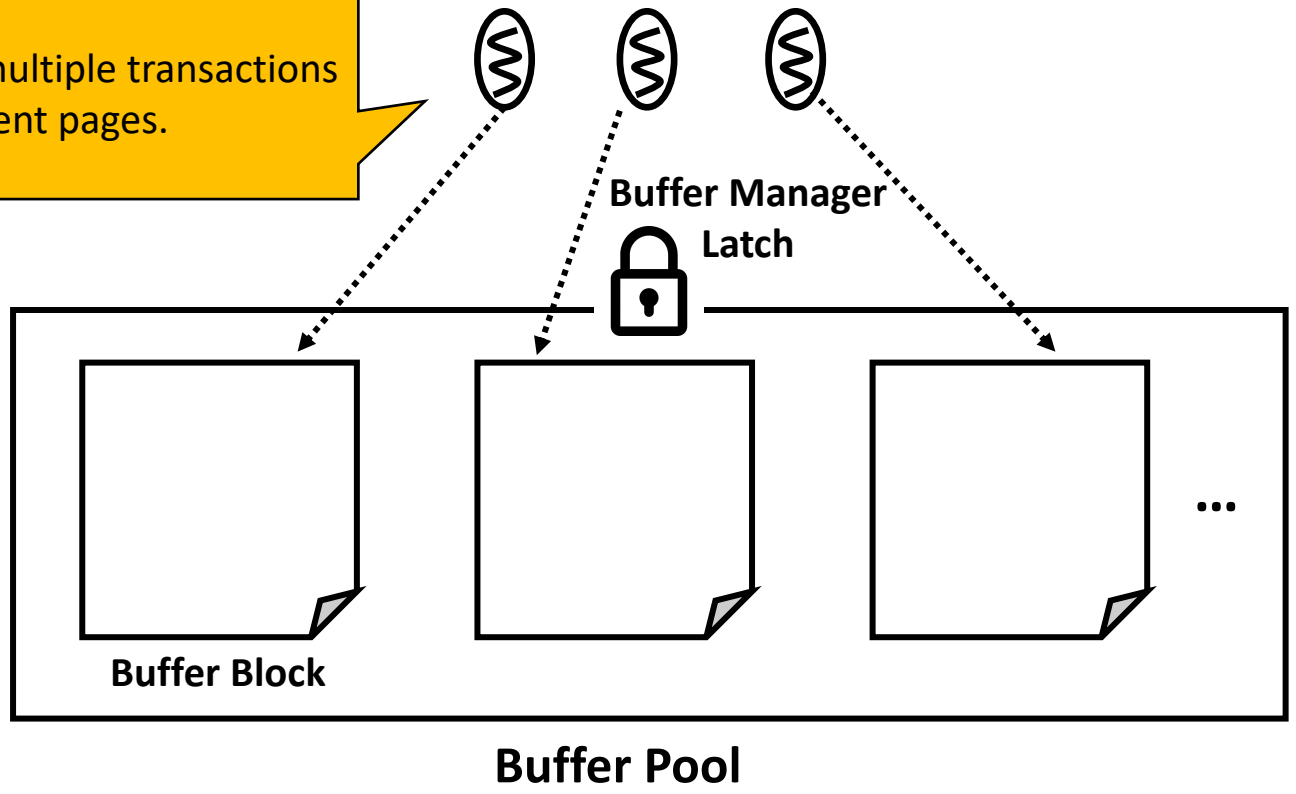
Buffer Manager Issues

It is enough to safely protect the entire buffer pool by using a global **buffer manager latch** for satisfying correctness,

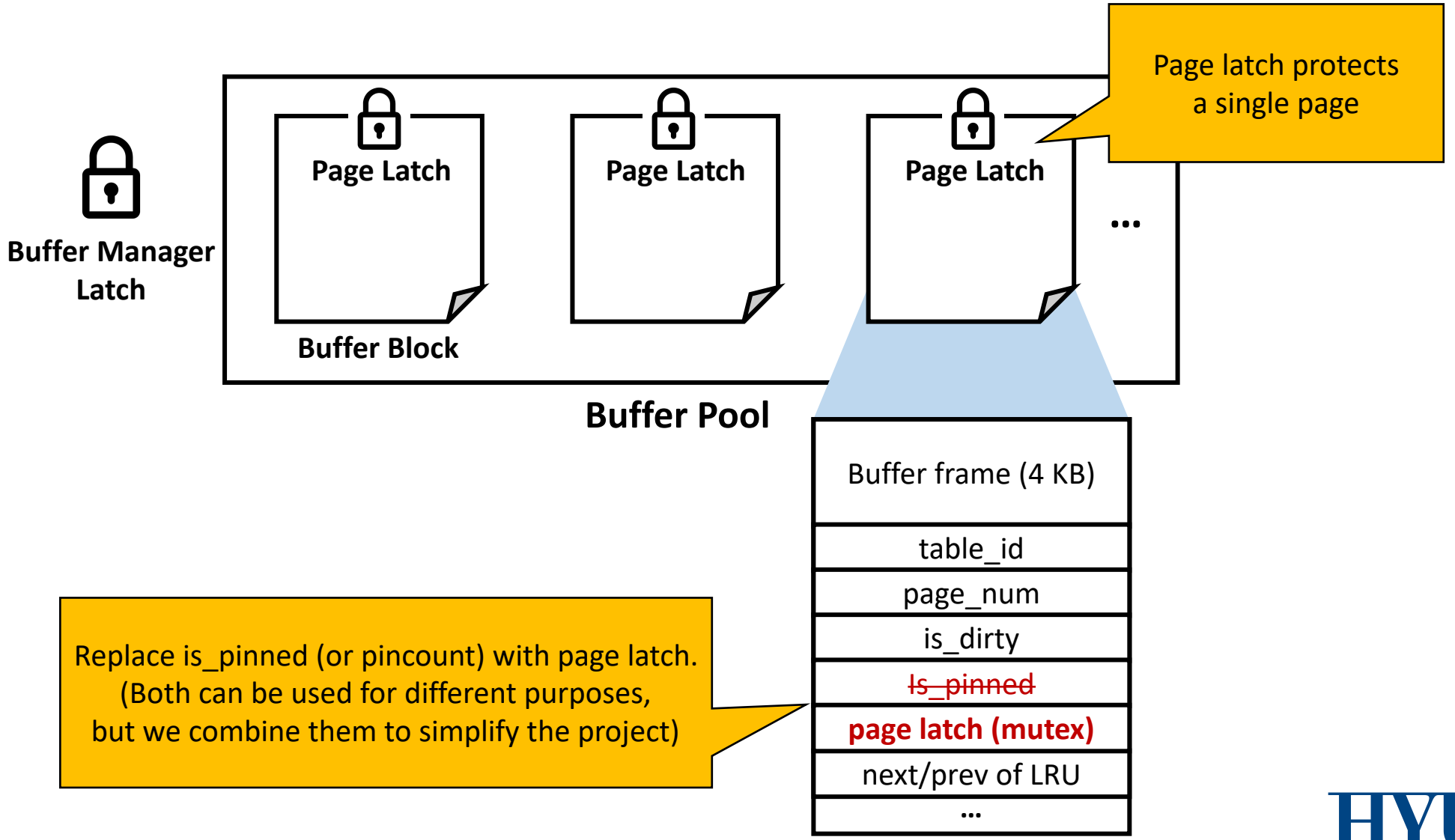


Buffer Manager Issues

but too inefficient if there are multiple transactions trying to access different pages.



Page Latch



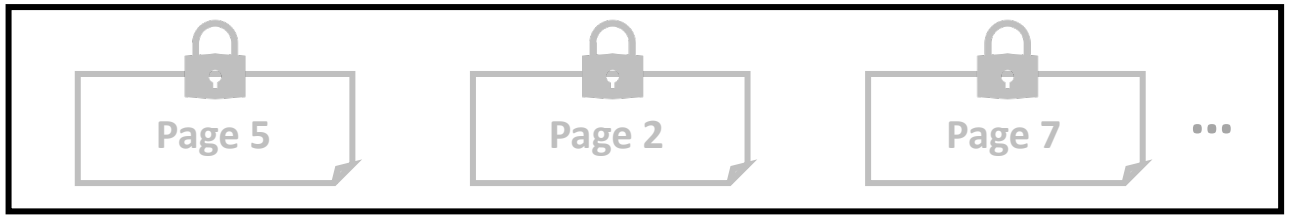
Page Latch

Transaction

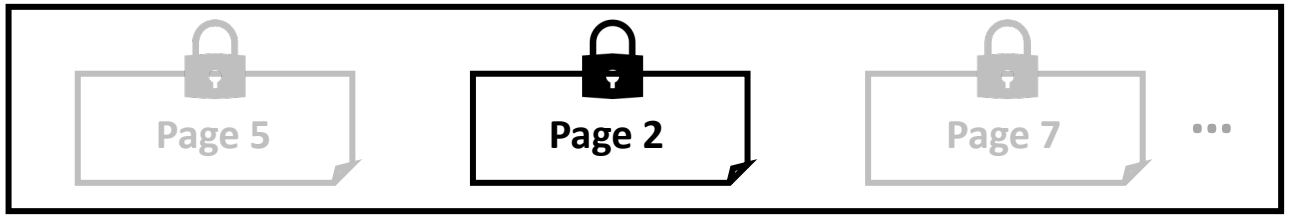
Try to access page 2

1. Acquire the buffer manager latch

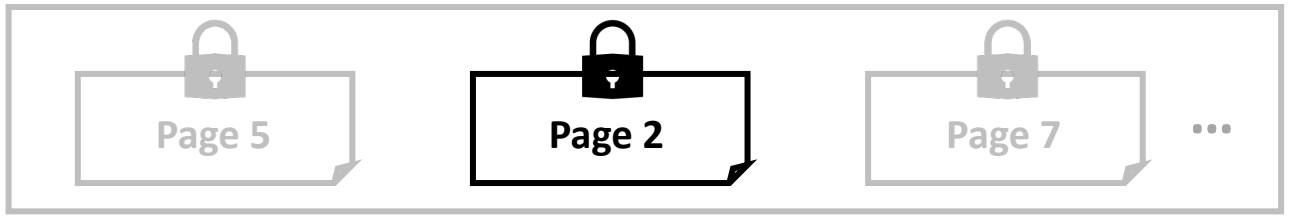
Buffer Manager Latch



2. Acquire the page latch



3. Release the buffer manager latch



Page Latch

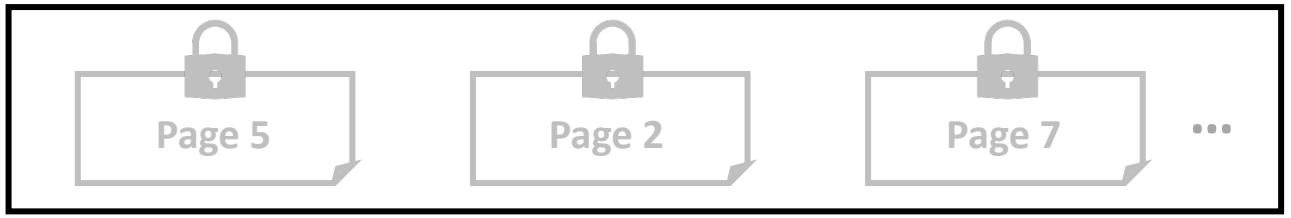
Transaction

Try to access page 2

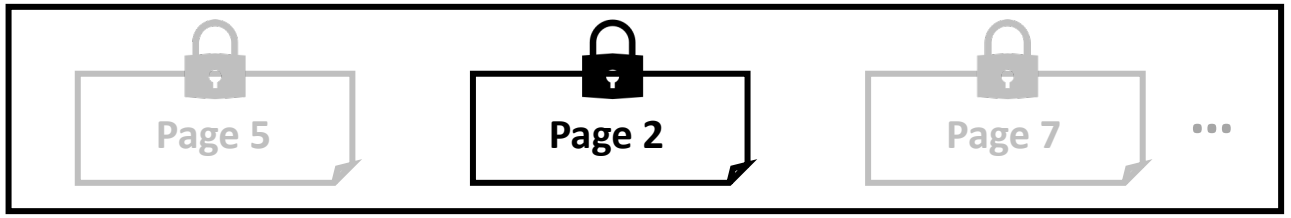
LRU list need to be protected by the buffer manager latch.

1. Acquire the buffer manager latch

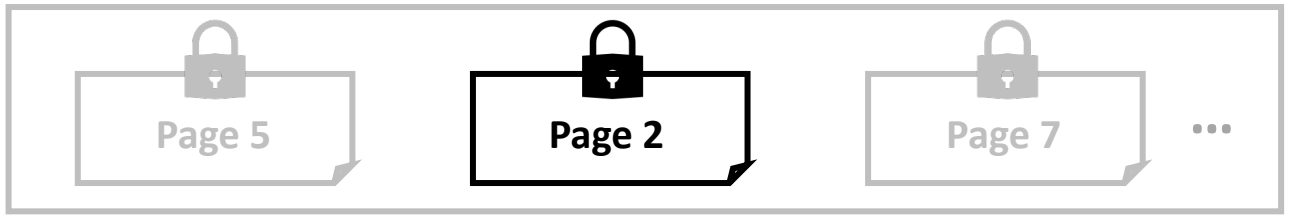
Buffer Manager Latch



2. Acquire the page latch



3. Release the buffer manager latch



Page Latch

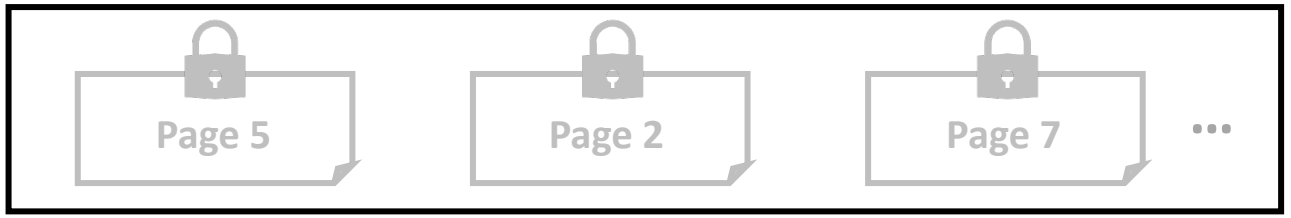
Transaction

Evict page 7

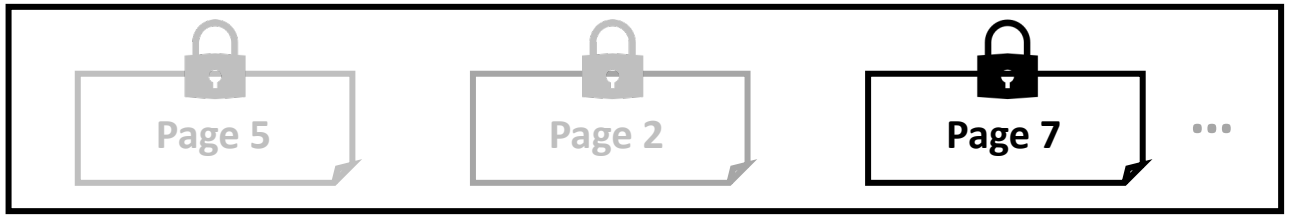
Page eviction also need to be protected by the buffer manager latch

1. Acquire the buffer manager latch

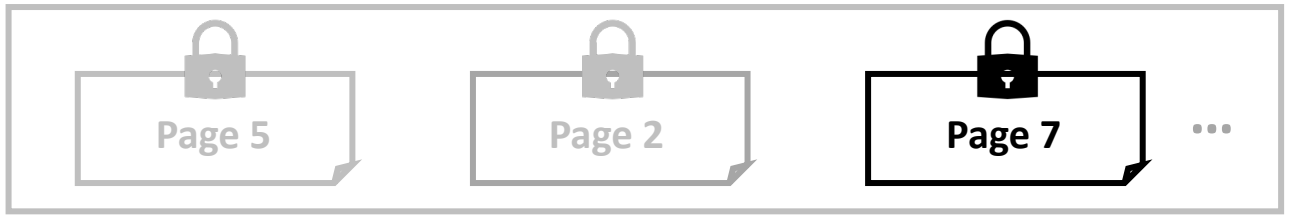
Buffer Manager Latch



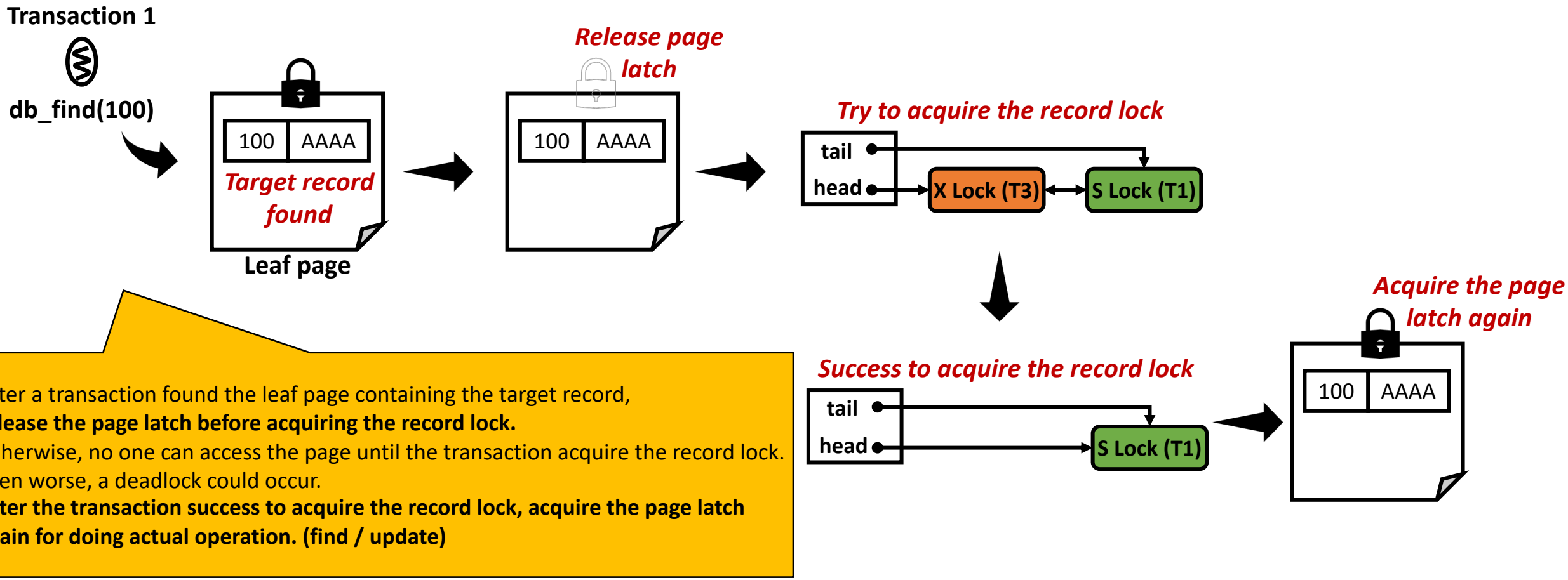
2. Acquire the page latch



3. Release the buffer manager latch



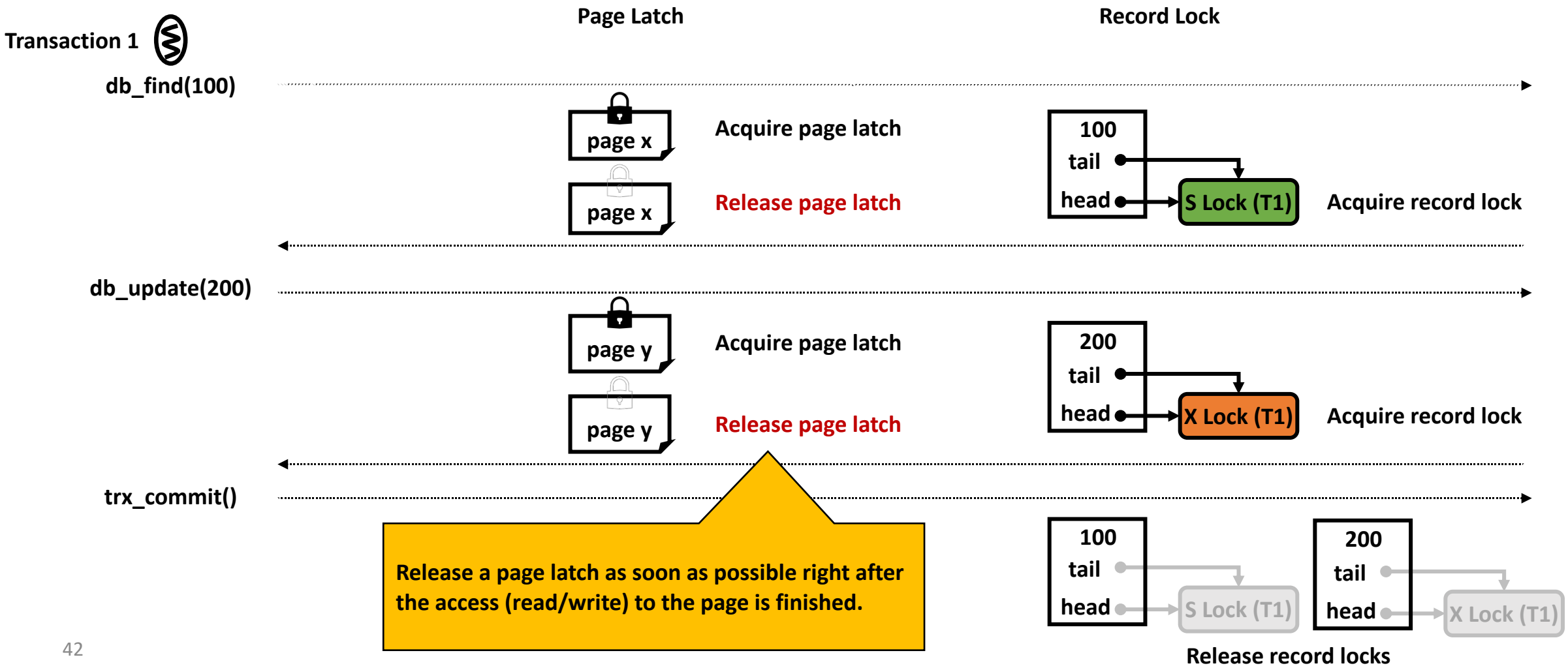
Page Latch & Record Lock



After a transaction found the leaf page containing the target record,
release the page latch before acquiring the record lock.
Otherwise, no one can access the page until the transaction acquire the record lock.
Even worse, a deadlock could occur.
After the transaction success to acquire the record lock, acquire the page latch again for doing actual operation. (find / update)

Page Latch & Record Lock

- Page latch duration vs Record lock duration



Wiki

- Your wiki should contain descriptions about
  - lock mode (shared & exclusive)
  - deadlock detection
  - abort and rollback
  - and whatever you want to describe.

Submission

- Deadline: Nov 30 11:59pm

- We will only score your commit before the deadline, and your submission after the deadline will not be accepted.

Thank you