Today:

- Timestamp-based concurrency control.
- Validation techniques for concurrency control.
- Scheduling and cascading rollback.
- Deadlocks.

Soon:

- Distributed locking, commitment.

Serializability Via Validation

Another timestamp-based approach, but:

- Don't keep timestamps with DB elements.
- System maintains information about transactions and their
  - *Read set* $= RS(T) =$ set of elements read by transaction T.
  - *Write set* $= WS(T) =$ set of elements written by T.
- Timestamp of a transaction is the time of its *validation* (after executing, but before writing to DB).
  - Transactions must behave as if they were executed serially in the order of their timestamps.

Validation Algorithm

Implement transactions to have three phases:

1. *Read*: values read from the DB to memory.
2. *Validate*: after all computation done in memory buffers, check that read- and write-sets do not imply a violation of the assumed serial order.
   - Rollback and restart if not.
3. *Write*: if validation is successful, write to DB.

System Data

The system maintains:

1. $START$ = the set of transactions that have started but not been validated.

2. VAL = set of transactions that have gone through phase 2 successfully, but not finished phase 3.

3. FIN = the set of transactions that have completed phase 3.

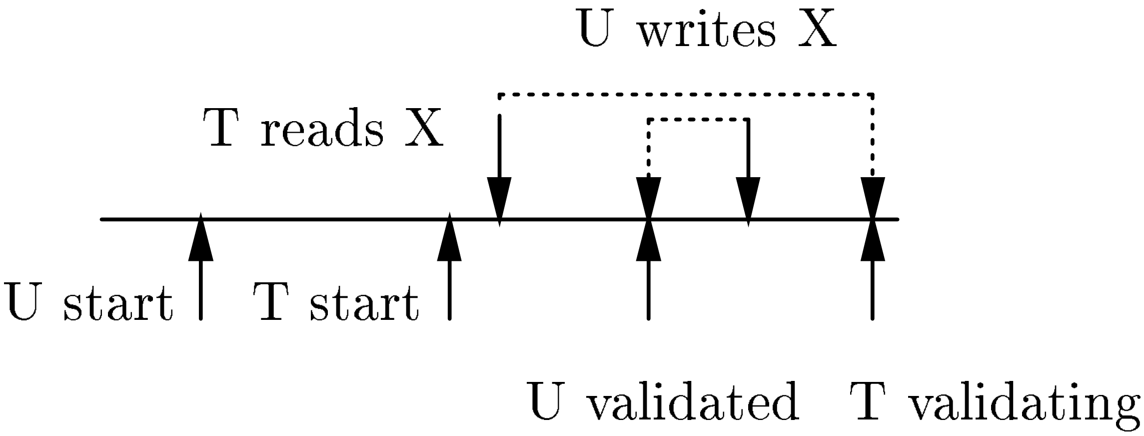
   - These can be flushed periodically in a checkpoint-like process.

- we keep the times $START(T)$, $VAL(T)$, and $FIN(T)$ for each transaction T in one of these sets, if known.

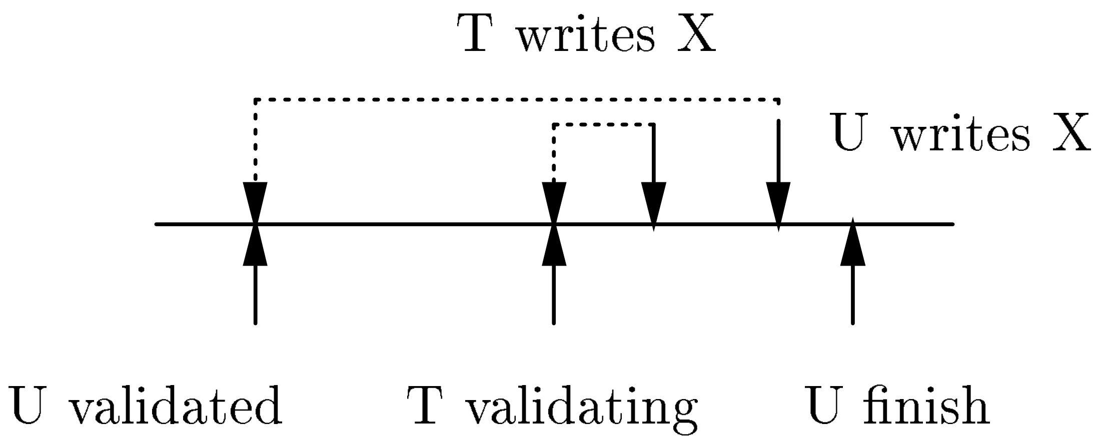
Validation Process

Suppose we need to validate T:

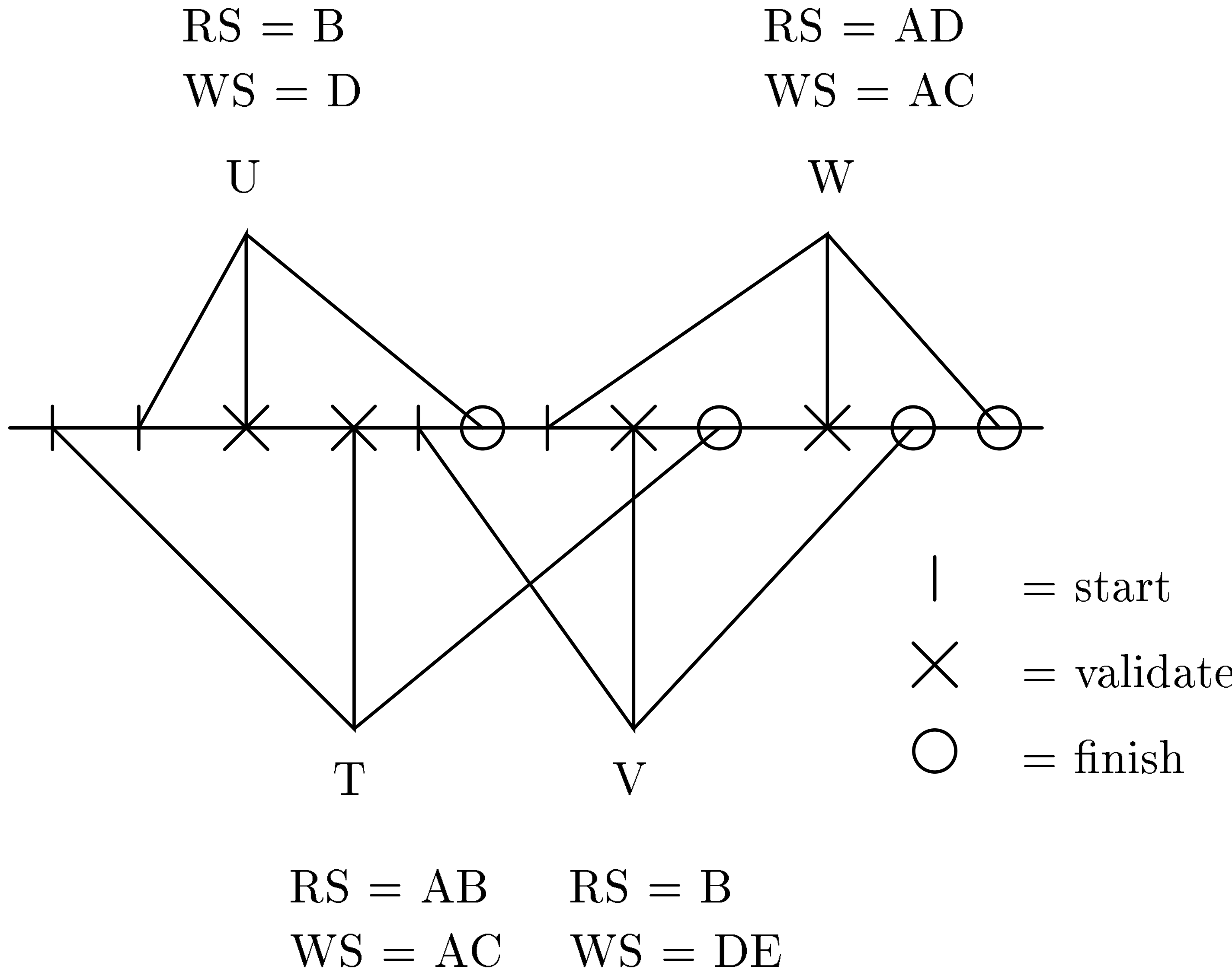
- Problem if $RS(T) \cap WS(U) \neq \emptyset$ for some transaction U that has validated, but did not finish before T started.



- Problem if $WS(T) \cap WS(U) \neq \emptyset$ for some transaction U that has validated, but did not finish before T began to validate.



Example



- General rule: compare your RS and WS against the WS of all validated transactions, except:
  - ❖ Skip your RS if it finished before you started; skip your WS if it has finished now.

- U validates with no prior transactions, so no problem.
- T validates: U has already validated, but is not finished.
  - Must check $RS(T) \cap WS(U) = \emptyset$; $WS(T) \cap WS(U) = \emptyset$.
- V validates: U is validated and finished; T is validated, not finished.
  - Must check: $RS(V) \cap WS(T) = \emptyset$; $WS(V) \cap WS(T) = \emptyset$; $RS(V) \cap WS(U) = \emptyset$.
- W validates: U finished before W started, so no checks; T finished before W started validating; V validated but not finished.
  - Must check: $RS(W) \cap WS(T) = \emptyset$; $RS(W) \cap WS(V) = \emptyset$; $WS(W) \cap WS(V) = \emptyset$.
  - Only the last is true, so W must abort for two reasons.

Comparison: Locks, Timestamps, Validation

- Storage:
  - Locks: lock(s) per locked DB element.
  - Timestamps: timestamp(s) per (any) DB element, plus transactions.
  - Validation: timestamps and read/write sets per transaction.
- Performance:
  - Locks: avoids rollbacks, even when interference is heavy.
  - Timestamps, validation: low overhead if interference uncommon.
  - Timestamps: earlier discovery of problems.

Cascaded Rollback

Risk: if transaction T_1 aborts after T_2 has read something T_1 wrote into the DB, then T_2 has "read dirty data" and must abort also (*cascading rollback*).

- A schedule is called *recoverable* if whenever T_2 has read something T_1 wrote, and T_2 commits, then T_1 has committed previously.

Assuring Recoverability

- With locking, scheduler must not release a write-lock until *after* commit or abort is executed.

- Timestamps: already discussed; depends on whether we put flags on DB elements to tell whether their writer has committed.

- Validation: never a problem; you can't commit until your "suppliers" have committed.

Logical Undo/Redo

Consider B-tree nodes as DB elements with individual locks. Problems:

- We change nodes by insertion/deletion.
  - Does whole new/old value need to be logged? Much more expensive that logging new/old integers or tuples.
- What would it mean to "undo" an insert or delete?
  - Cannot restore old node, because other transactions may have in the interrum inserted/deleted from the B-tree and committed.
- Simple solution: rule for avoiding cascading rollbacks would prevent such a transaction from committing, so the problem cannot exist.
- Two useful ideas on next slide.

Log Changes

If all we want to do is be able to redo or undo DB modifications, keep a description of the change to a DB element in the log, rather than entire new/old values.

- Example: "Key K and pointer P were inserted into B-tree node N."
- Can save a *lot* of log space.

Logical Undo/Redo

Think of a DB element, e.g., a B-tree, as an abstract value, rather than a collection of bits.

- We may not be able to restore the bits, but we can make changes that have the desired effect.
- Example: We may not be able to undo a B-tree insertion exactly, because of intervening, committed changes. But we can perform a deletion that leaves an *equivalent* tree as far as its use as an index is concerned.

Deadlocks

We've seen several causes of deadlocks:

- Two transactions each acquire a lock on a DB element the other needs.
- Two transactions try to upgrade locks on elements the other is reading.

Detecting Deadlocks

1. Lock table includes graph of who is waiting for whom. Look for cycles.
2. *Timeout.* If a transaction hasn't completed in x minutes, abort it.

Deadlock Prevention

Just as serializability schemes are all based on forcing a consistent order to transactions (e.g., point of first unlock for 2PL), deadlock-prevention schemes need to establish a priority order.

1. Require transactions to take all locks at the same time.
   - Priority order = wall-clock time.
2. Place an order on DB elements, and require transactions to take locks in that order.
   - Priority order = who first locks a commonly needed element.

Example

T_1 needs locks on A, B, and C. T_2 needs locks on B, C, and D.

- T_1 locks A; T_2 locks B. Now, T_1 is stalled and cannot lock B or C. Thus, T_2 surely gets to lock C first.