Introduction to Database Systems
CSE 444

Lecture 12

Transactions: concurrency control

(part 2)

Outline

- Concurrency control by timestamps (18.8)
- Concurrency control by validation (18.9)
- Concurrency control by snapshot isolation

- But first, a word about Phantoms…

Phantom Problem

- So far we have assumed the database to be a *static* collection of elements (=tuples)

- If tuples are inserted/deleted then the *phantom problem* appears

The Phantom Problem

"Phantom" = tuple visible only during some part of the transaction

```
T1:
    select count(*) from R where price>20
    . . . .
    . . . .
    . . . .
    . . . .
    select count(*) from R where price>20
```

```
T2:
   . . . .
   . . . .
   insert into R(name,price)
           values('Gizmo', 50)
    . . . .
```

$R_1(X), R_1(Y), R_1(Z), W_2(New), R_1(X), R_1(Y), R_1(Z), R_1(New)$

The schedule is conflict-serializable, yet we get different counts !
Not serializible because of phantoms.

Dealing with Phantoms

- In a ***static*** database:
  - Conflict serializability implies serializability
- In a ***dynamic*** database, this may fail due to phantoms
- Strict 2PL guarantees conflict serializability, but not serializability
- Expensive ways of dealing with phantoms:
  - Lock the entire table, or
  - Lock the index entry for ‘price’ (if index is available)
  - Or use *predicate locks* (a lock on an arbitrary predicate)

Serializable transactions are very expensive

Concurrency Control Mechanisms

- Pessimistic:
  - Locks
- Optimistic
  - Timestamp based: basic, multiversion
  - Validation
  - Snapshot isolation: a variant of both

Timestamps

- Each transaction receives a unique timestamp TS(T)

Could be:

- The system’s clock
- A unique counter, incremented by the scheduler

Timestamps

Main invariant:

The timestamp order defines
the serialization order of the transaction

Will generate a schedule that is view-equivalent to a serial schedule, and is recoverable

Main Idea

- For any two conflicting actions, ensure that their order is the serialized order:

In each of these cases

- $w_U(X) \ldots r_T(X)$
- $r_U(X) \ldots w_T(X)$
- $w_U(X) \ldots w_T(X)$

Read too late ?

Write too late ?

When T requests $r/w_T(X)$, need to check $TS(U) \leq TS(T)$

Timestamps

With each element X, associate

- RT(X) = the highest timestamp of any transaction that read X
- WT(X) = the highest timestamp of any transaction that wrote X
- C(X) = the commit bit: true when transaction with highest timestamp that wrote X committed

If 1 element = 1 page, these are associated with each page X in the buffer pool

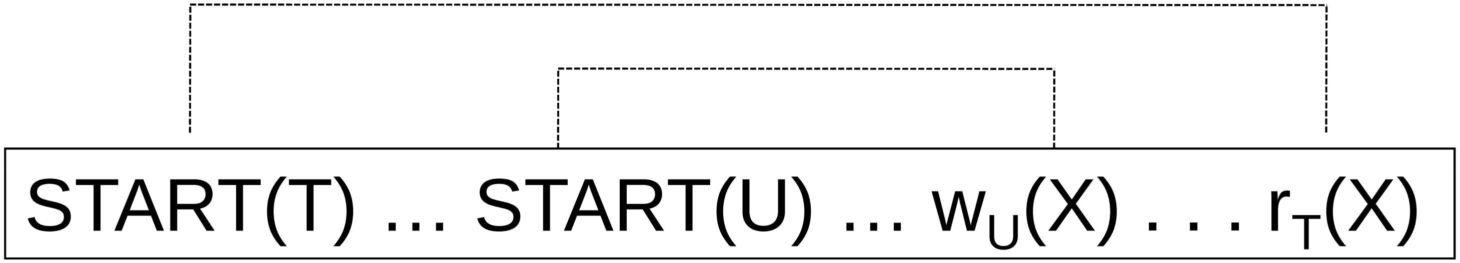
Time-based Scheduling

- Note: simple version that ignores the commit bit
  - If transactions abort, may result in non-recoverable schedule
- Transaction wants to read element X
  - If TS(T) < WT(X) then ROLLBACK
  - Else read X and update RT(X) to larger of TS(T) or RT(X)
- Transaction wants to write element X
  - If TS(T) < RT(X) then ROLLBACK
  - Else if TS(T) < WT(X) ignore write & continue (Thomas Write Rule)
  - Otherwise, write X and update WT(X) to TS(T)

Details

Read too late:

- T wants to read X, and TS(T) < WT(X)



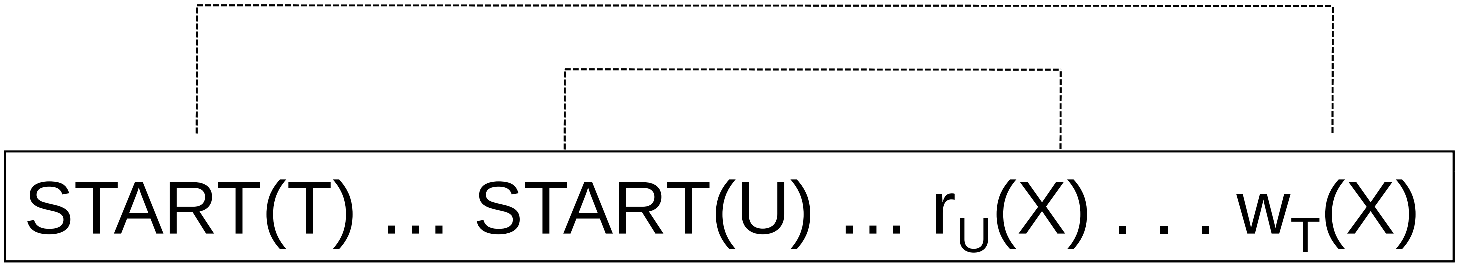
START(T) … START(U) … $w_U(X)$ . . . $r_T(X)$

Need to rollback T !

Details

Write too late:

- T wants to write X, and TS(T) < RT(X)



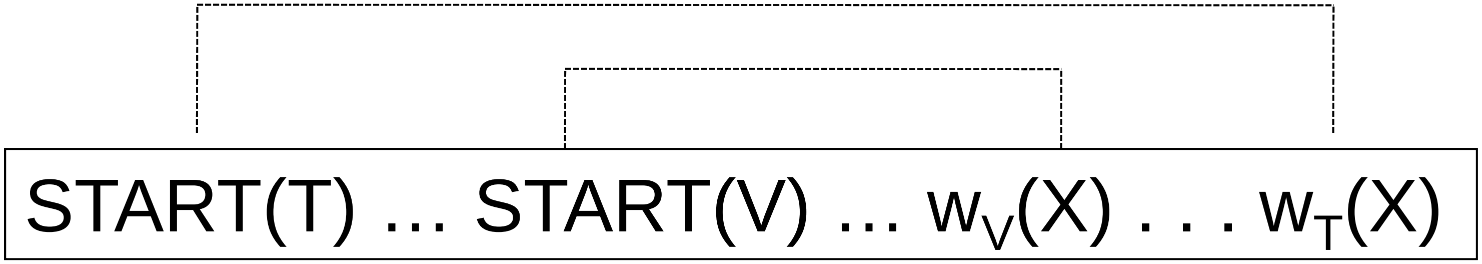
START(T) … START(U) … $r_U(X)$ . . . $w_T(X)$

Need to rollback T !

Details

Write too late, but we can still handle it:

- T wants to write X, and
  $TS(T) >= RT(X)$ but $WT(X) > TS(T)$



START(T) … START(V) … $w_V(X)$ . . . $w_T(X)$

Don't write X at all !
(Thomas' rule)

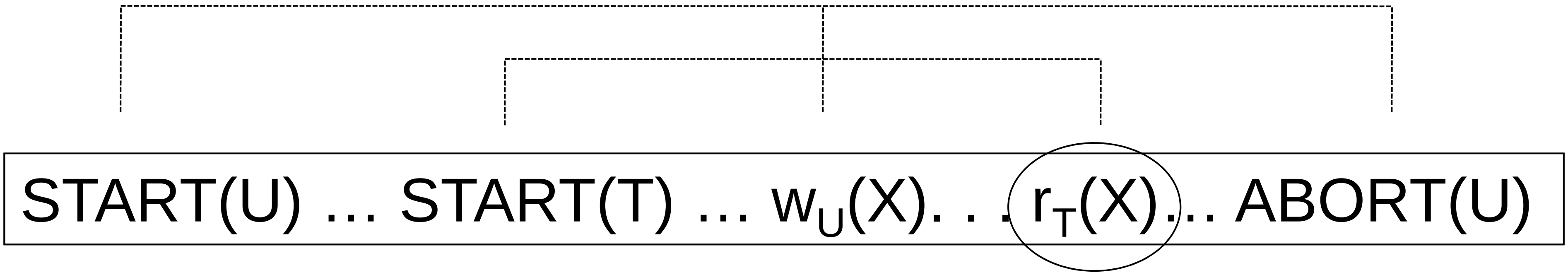
Ensuring Recoverable Schedules

- Recall the definition: if a transaction reads an element, then the transaction that wrote it must have already committed
- Use the commit bit C(X) to keep track if the transaction that last wrote X has committed

Ensuring Recoverable Schedules

Read dirty data:

- T wants to read X, and WT(X) < TS(T)
- Seems OK, but…



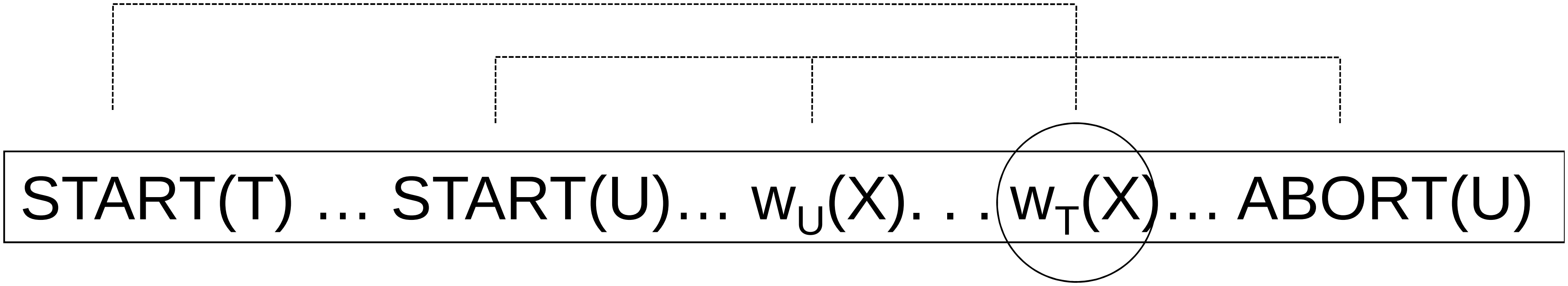
START(U) … START(T) … $w_U(X)$. . . $r_T(X)$… ABORT(U)

If C(X)=false, T needs to wait for it to become true

Ensuring Recoverable Schedules

Need to revise Thomas' rule:

- T wants to write X, and WT(X) > TS(T)
- Seems OK not to write at all, but …



START(T) … START(U)… w_U(X). . . w_T(X)… ABORT(U)

If C(X)=false, T needs to wait for it to become true

Timestamp-based Scheduling

- When a transaction T requests r(X) or w(X), the scheduler examines RT(X), WT(X), C(X), and decides one of:

- To grant the request, or
- To rollback T (and restart with later timestamp)
- To delay T until C(X) = true

Timestamp-based Scheduling

Transaction wants to READ element X

```
    If TS(T) < WT(X)  then ROLLBACK
    Else If C(X) = false, then WAIT
    Else READ and update RT(X) to larger of TS(T) or RT(X)
```

Transaction wants to WRITE element X

```
    If TS(T) < RT(X) then ROLLBACK
    Else if TS(T) < WT(X)
        Then If C(X) = false then WAIT
                else IGNORE write (Thomas Write Rule)
    Otherwise, WRITE, and update WT(X)=TS(T), C(X)=false
```

See book sec. 18.8.4 for detailed rules

Summary of Timestamp-based Scheduling

- Conflict-serializable
- Recoverable
  - Even avoids cascading aborts
- Does NOT handle phantoms

Multiversion Timestamp

- When transaction T requests r(X) but WT(X) > TS(T), then T must rollback
- Idea: keep multiple versions of X: X_t, X_{t-1}, X_{t-2}, . . .

$$\boxed{TS(X_t) > TS(X_{t-1}) > TS(X_{t-2}) > \ldots}$$

- Let T read an older version, with appropriate timestamp

Details

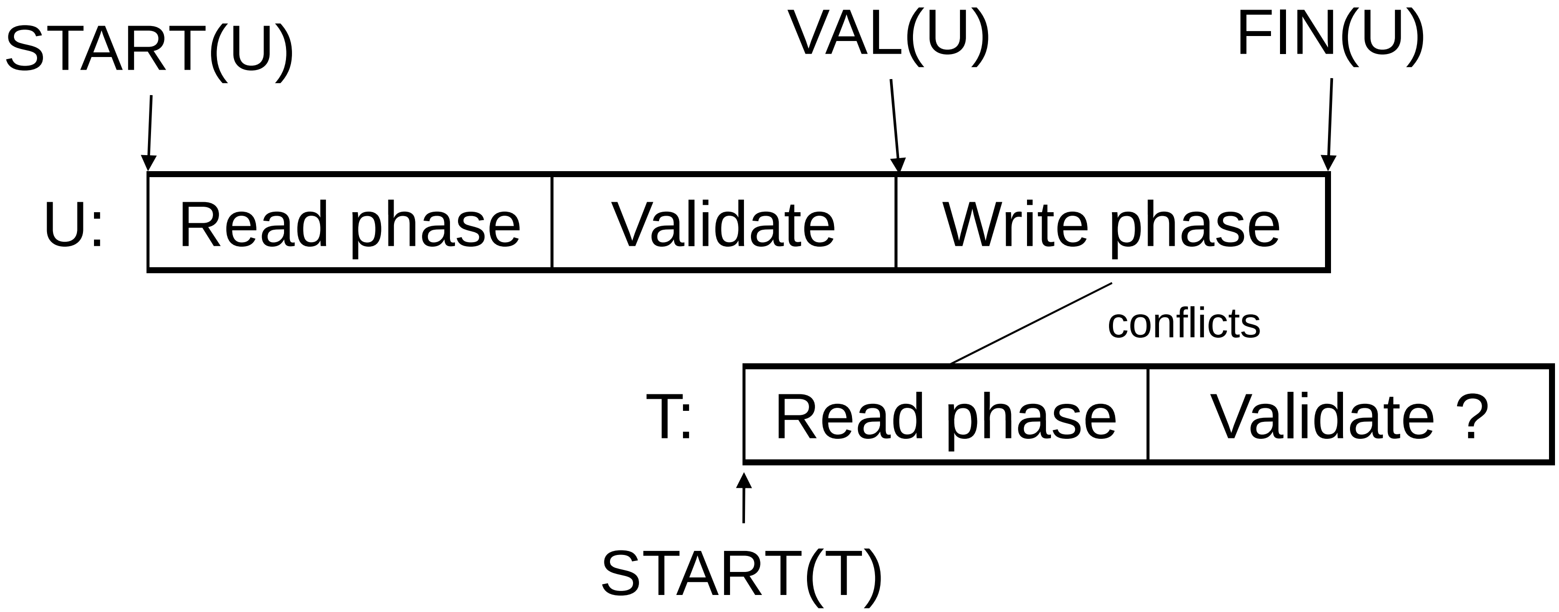
- When $w_T(X)$ occurs,
  create a new version, denoted X_t where $t = TS(T)$
- When $r_T(X)$ occurs,
  find most recent version X_t such that $t < TS(T)$
  Notes:
  - $WT(X_t) = t$ and it never changes
  - $RT(X_t)$ must still be maintained to check legality of writes
- Can delete X_t if we have a later version X_{t1} and all active transactions T have $TS(T) > t1$

Concurrency Control by Validation

- Each transaction T defines a *read set* RS(T) and a *write set* WS(T)
- Each transaction proceeds in three phases:
  - Read all elements in RS(T). Time = START(T)
  - Validate (may need to rollback). Time = VAL(T)
  - Write all elements in WS(T). Time = FIN(T)

Main invariant: the serialization order is VAL(T)

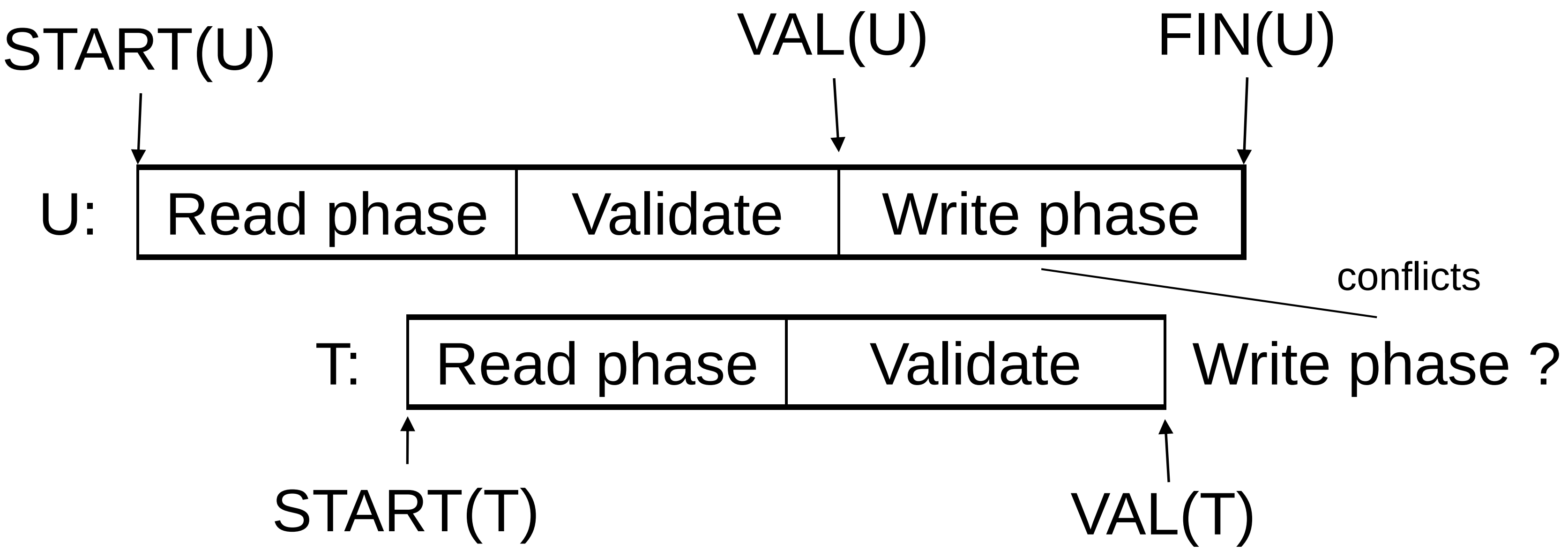
Avoid $r_T(X)$ - $w_U(X)$ Conflicts



IF RS(T) ∩ WS(U) and FIN(U) > START(T)
(U has validated and U has not finished before T begun)

Then ROLLBACK(T)

Avoid $w_T(X)$ - $w_U(X)$ Conflicts



IF WS(T) $\cap$ WS(U) and FIN(U) > VAL(T)

(U has validated and U has not finished before T validates)

Then ROLLBACK(T)

Snapshot Isolation

- Another optimistic concurrency control method
- Very efficient, and very popular
  - Oracle, PostgreSQL, SQL Server 2005
- Prevents many classical anomalies BUT…
- Not serializable (!), yet ORACLE uses it even for SERIALIZABLE transactions!

Snapshot Isolation Rules

- Each transactions receives a timestamp TS(T)
- Transaction T sees database snapshot at time TS(T)
- When T commits, updated pages are written to disk
- Write/write conflicts resolved by “first committer wins” rule
- Read/write conflicts are ignored

Snapshot Isolation (Details)

- Multiversion concurrency control:
  - Versions of X: Xt1, Xt2, Xt3, . . .
- When T reads X, return $X_{TS(T)}$.
- When T writes X: if other transaction updated X, abort
  - Not faithful to "first committer" rule, because the other transaction U might have committed after T. But once we abort T, U becomes the first committer ☺

What Works and What Not

- No dirty reads (Why ?)
- No inconsistent reads (Why ?)
  - A: Each transaction reads a consistent snapshot

- No lost updates (“first committer wins”)

- Moreover: no reads are ever delayed

- However: read-write conflicts not caught !

Write Skew

```
T1:
  READ(X);
  if X >= 50
      then Y = -50; WRITE(Y)
  COMMIT
```

```
T2:
  READ(Y);
  if Y >= 50
      then X = -50; WRITE(X)
  COMMIT
```

In our notation:

$R_1(X), R_2(Y), W_1(Y), W_2(X), C_1,C_2$

Starting with X=50,Y=50, we end with X=-50, Y=-50.
Non-serializable !!!

Write Skews Can Be Serious

- Acidicland had two viceroys, Delta and Rho
- Budget had two registers: taXes, and spendYng
- They had high taxes and low spending…

```
Delta:
  READ(taXes);
  if taXes = ‘High’
      then { spendYng = ‘Raise’;
              WRITE(spendYng) }
  COMMIT
```

```
Rho:
  READ(spendYng);
  if spendYng = ‘Low’
      then {taXes = ‘Cut’;
              WRITE(taXes) }
  COMMIT
```

… and they ran a deficit ever since.

Tradeoffs

- Pessimistic Concurrency Control (Locks):
  - Great when there are many conflicts
  - Poor when there are few conflicts (overhead)
- Optimistic Concurrency Control (Timestamps):
  - Poor when there are many conflicts (rollbacks)
  - Great when there are few conflicts

- Compromise
  - READ ONLY transactions → timestamps
  - READ/WRITE transactions → locks