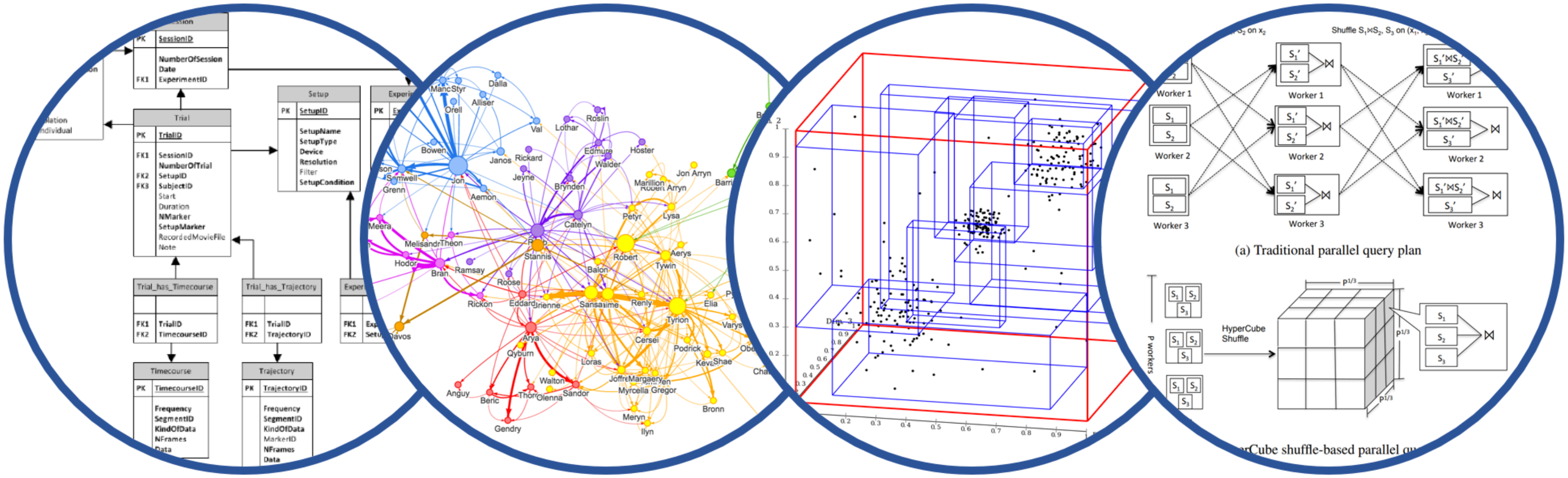


Database System Internals

Optimistic Concurrency Control

Paul G. Allen School of Computer Science and Engineering
University of Washington, Seattle

Pessimistic vs. Optimistic

- Pessimistic CC (locking)
  - Prevents unserializable schedules
  - Never abort for serializability (but may abort for deadlocks)
  - Best for workloads with high levels of contention

- Optimistic CC (timestamp, multi-version, validation)
  - Assume schedule will be serializable
  - Abort when conflicts detected
  - Best for workloads with low levels of contention

Outline

- Concurrency control by timestamps (18.8)
- Concurrency control by validation (18.9)
- Snapshot Isolation

Timestamps

- Each transaction receives 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 recoverable

Timestamps

With each element X, associate

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

Timestamps

With each element X, associate

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

If transactions abort, we must **reset the timestamps**

Main Idea

For any $r_T(X)$ or $w_T(X)$ request, check for conflicts:

- $w_U(X)$ . . . $r_T(X)$
- $r_U(X)$ . . . $w_T(X)$
- $w_U(X)$ . . . $w_T(X)$

How do we check if Read too late ?

Write too late ?

Main Idea

For any $r_T(X)$ or $w_T(X)$ request, check for conflicts:

- $w_U(X)$ . . . $r_T(X)$
- $r_U(X)$ . . . $w_T(X)$
- $w_U(X)$ . . . $w_T(X)$

How do we check if Read too late ?

Write too late ?

When T requests $r_T(X)$, need to check $TS(U) \le TS(T)$

Read Too Late

- T wants to read X

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

Read Too Late

- T wants to read X

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

If WT(X) > TS(T) then need to rollback T !
T tried to read **too late**

Write Too Late

- T wants to write X

$\text{START}(T) \ldots \text{START}(U) \ldots r_U(X) \ldots w_T(X)$

Write Too Late

- T wants to write X

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

If RT(X) > TS(T) then need to rollback T !

T tried to write **too late**

Thomas' Rule

But… we can still handle it in one case:

- T wants to write X

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

Thomas' Rule

But we can still handle it:

- T wants to write X

$$\text{START}(T)\ \ldots\ \text{START}(V)\ \ldots\ w_V(X)\ .\ .\ .\ w_T(X)$$

If $RT(X) \leq TS(T)$ and $WT(X) > TS(T)$
then don't write X at all !

Why does this work?

Thomas' Rule

But we can still handle it:

- T wants to write X

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

If $RT(X) \leq TS(T)$ and $WT(X) > TS(T)$
then don't write X at all !

Why does this work?

View-serializable:
V will have overwritted T!

View-Serializability

- By using Thomas' rule we do obtain a view-serializable schedule

Summary So Far

Only for transactions that do not abort

Otherwise, may result in non-recoverable schedule

Transaction wants to **READ** element X
If WT(X) > TS(T) then ROLLBACK
Else READ and update RT(X) to larger of TS(T) or RT(X)

Transaction wants to **WRITE** element X
If RT(X) > TS(T) then ROLLBACK
Else if WT(X) > TS(T) ignore write & continue (Thomas Write Rule)
Otherwise, WRITE and update WT(X) =TS(T)

Ensuring Recoverable Schedules

Recall:

- Schedule avoids cascading aborts if whenever 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
  (just a read will not change the commit bit)

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

Thomas' rule needs to be revised:

- 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 $r_T(X)$ or $w_T(X)$, 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

RULES including commit bit

- There are 4 long rules in Sec. 18.8.4
- You should be able to derive them yourself, based on the previous slides
- Make sure you understand them !

READING ASSIGNMENT:
Garcia-Molina et al. 18.8.4

Timestamp-based Scheduling (sec. 18.8.4)

Transaction wants to READ element X

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

Basic Timestamps with Commit Bit

T_1	T_2	T_3	T_4	A
1	2	3	4	RT=0 WT=0 C=true
	$W_2(A)$			

Time ↓

Basic Timestamps with Commit Bit

Time ↓

T_1	T_2	T_3	T_4	A	
1	2	3	4	RT=0	
				WT=0	C=true
	$W_2(A)$			WT=2	C=false
$R_1(A)$				RT=0	
		$R_3(A)$			
Abort					
		Delay			
	C				C=true
		$R_3(A)$		RT=3	
			$W_4(A)$	WT=4	C=false
		$W_3(A)$			
		delay			
			abort	WT=2	C=true
		W3(A)		WT=3	C=false

Summary of Timestamp-based Scheduling

- View-serializable
- Avoids cascading aborts (hence: recoverable)
- Does NOT handle phantoms
  - These need to be handled separately, e.g. predicate locks

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}, . . .

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

Details

- When $w_T(X)$ occurs,

  if the write is legal then

  create a new version, denoted X_t where t = TS(T)

Details

- When $w_T(X)$ occurs,
  if the write is legal then
  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 for that version
  - $RT(X_t)$ 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

Example (in class)

`Four versions of X:` X_3 X_9 X_{12} X_{18}

$R_6(X)$ -- Read X_3
$W_{21}(X)$ – Check read timestamp of X_{18}
$R_{15}(X)$ – Read X_{12}
$W_5(X)$ – Check read timestamp of X_3

When can we delete X_3?

Example w/ Basic Timestamps

	T_1	T_2	T_3	T_4	A
Timestamps:	150	200	175	225	RT=0 WT=0
	$R_1(A)$				RT=150
	$W_1(A)$				WT=150
		$R_2(A)$			RT=200
		$W_2(A)$			WT=200
			$R_3(A)$		
			Abort		
				$R_4(A)$	RT=225

Example w/ Multiversion

T_1	T_2	T_3	T_4	A_0	A_{150}	A_{200}
150	200	175	225			
$R_1(A)$				RT=150		
$W_1(A)$					Create	
	$R_2(A)$				RT=200	
	$W_2(A)$					Create
		$R_3(A)$			RT=200	
		$W_3(A)$				
		abort				
			$R_4(A)$			RT=225

Example w/ Multiversion

T_1	T_2	T_3	T_4	A_0	A_{150}	A_{200}
150	200	175	225			
$R_1(A)$				RT=150		
$W_1(A)$					Create	
	$R_2(A)$				RT=200	
	$W_2(A)$					Create
		$R_3(A)$			RT=200	
		$W_3(A)$				
		abort				
			$R_4(A)$			RT=225

Second Example w/ Multiversion

T_1	T_2	T_3	T_4	T_5	A_0	A_1	A_2	A_3	A_4	A_5
1	2	3	4	5						
			$W_4(A)$							

Second Example w/ Multiversion

T_1	T_2	T_3	T_4	T_5	A_0	A_1	A_2	A_3	A_4	A_5
1	2	3	4	5						
			$W_4(A)$						Create	
W1(A)						Create				
	$R_2(A)$					RT=2				
		$R_3(A)$				RT=3				
	$W_2(A)$									
	abort			$R_5(A)$					RT=5	
				$W_5(A)$						Create
			$R_4(A)$						RT=5	
$R_1(A)$						RT=3				
C					X					
		C				X				

X means that we can delete this version

Outline

- Concurrency control by timestamps (18.8)
- Concurrency control by validation (18.9)
- Snapshot Isolation

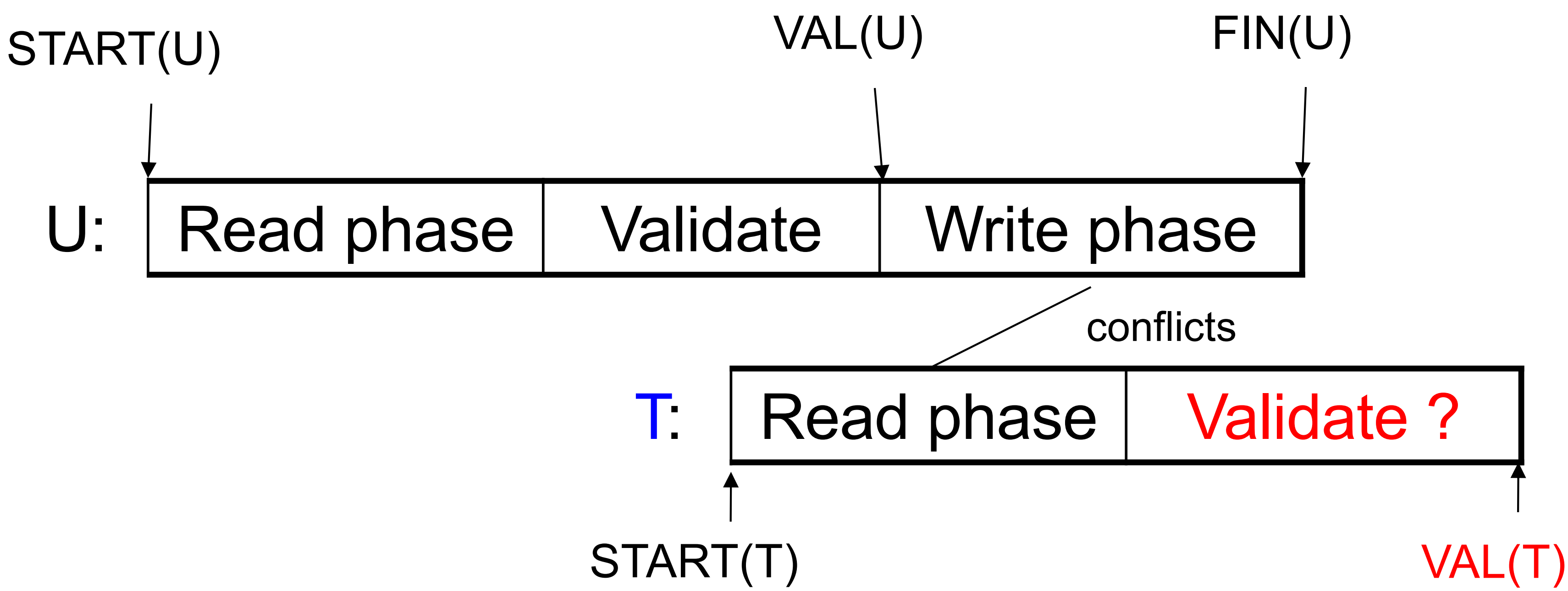
Concurrency Control by Validation

- Each transaction T defines:
  - Read set RS(T) = the elements it reads
  - Write set WS(T) = the elements it writes

- Each transaction T has three phases:
  - Read phase; time = START(T)
  - Validate phase (may need to rollback); time = VAL(T)
  - Write phase; time = FIN(T)

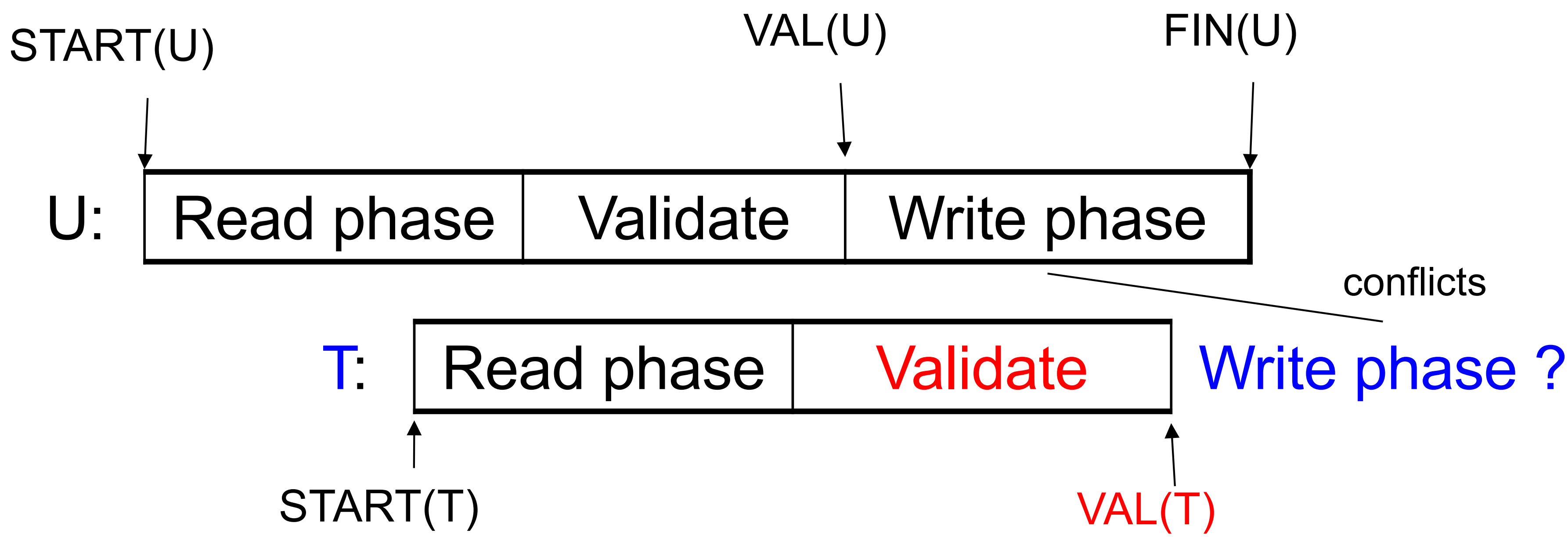
Main invariant: the serialization order is VAL(T)

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) ∩ WS(U) and FIN(U) > VAL(T)
(U has validated and U has not finished before T validates)
Then ROLLBACK(T)

Outline

- Concurrency control by timestamps (18.8)
- Concurrency control by validation (18.9)
- Snapshot Isolation
  - Not in the book, but good overview in Wikipedia

Snapshot Isolation

- A type of multiversion concurrency control algorithm
- Provides yet another level of isolation

- Very efficient, and very popular
  - Oracle, PostgreSQL, SQL Server 2005

- Prevents many classical anomalies BUT…
- Not serializable (!), yet ORACLE and PostgreSQL use it even for SERIALIZABLE transactions!
  - But "serializable snapshot isolation" now in PostgreSQL

Snapshot Isolation Overview

- Each transactions receives a timestamp TS(T)
- Transaction T sees snapshot at time TS(T) of the database
- W/W conflicts resolved by "**first committer wins**" rule
  - Loser gets aborted
- R/W conflicts are ignored

Snapshot Isolation Details

- Multiversion concurrency control:
  - Versions of X: $X_{t1}, X_{t2}, X_{t3}, \ldots$
- When T reads X, return $X_{TS(T)}$.
- When T writes X (to avoid lost update):
  - If latest version of X is TS(T) then proceed
  - Else if C(X) = true then abort
  - Else if C(X) = false then wait
- When T commits, write its updates to disk

What Works and What Not

- No dirty reads (Why ?)
  - Start each snapshot with consistent state
- No inconsistent reads (Why ?)
  - Two reads by the same transaction will read same snapshot
- No lost updates (“first committer wins”)
- Moreover: no reads are ever delayed

- However: read-write conflicts not caught!
  - A txn can read and commit even though the value had changed in the middle

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.

Discussion: Tradeoffs

- Pessimistic CC: Locks
  - Great when there are many conflicts
  - Poor when there are few conflicts
- Optimistic CC: Timestamps, Validation, SI
  - Poor when there are many conflicts (rollbacks)
  - Great when there are few conflicts
- Compromise
  - READ ONLY transactions → timestamps
  - READ/WRITE transactions → locks

Commercial Systems

Always check documentation!

- DB2: Strict 2PL
- SQL Server:
  - Strict 2PL for standard 4 levels of isolation
  - Multiversion concurrency control for snapshot isolation
- PostgreSQL: SI; recently: seralizable SI (!)
- Oracle: SI