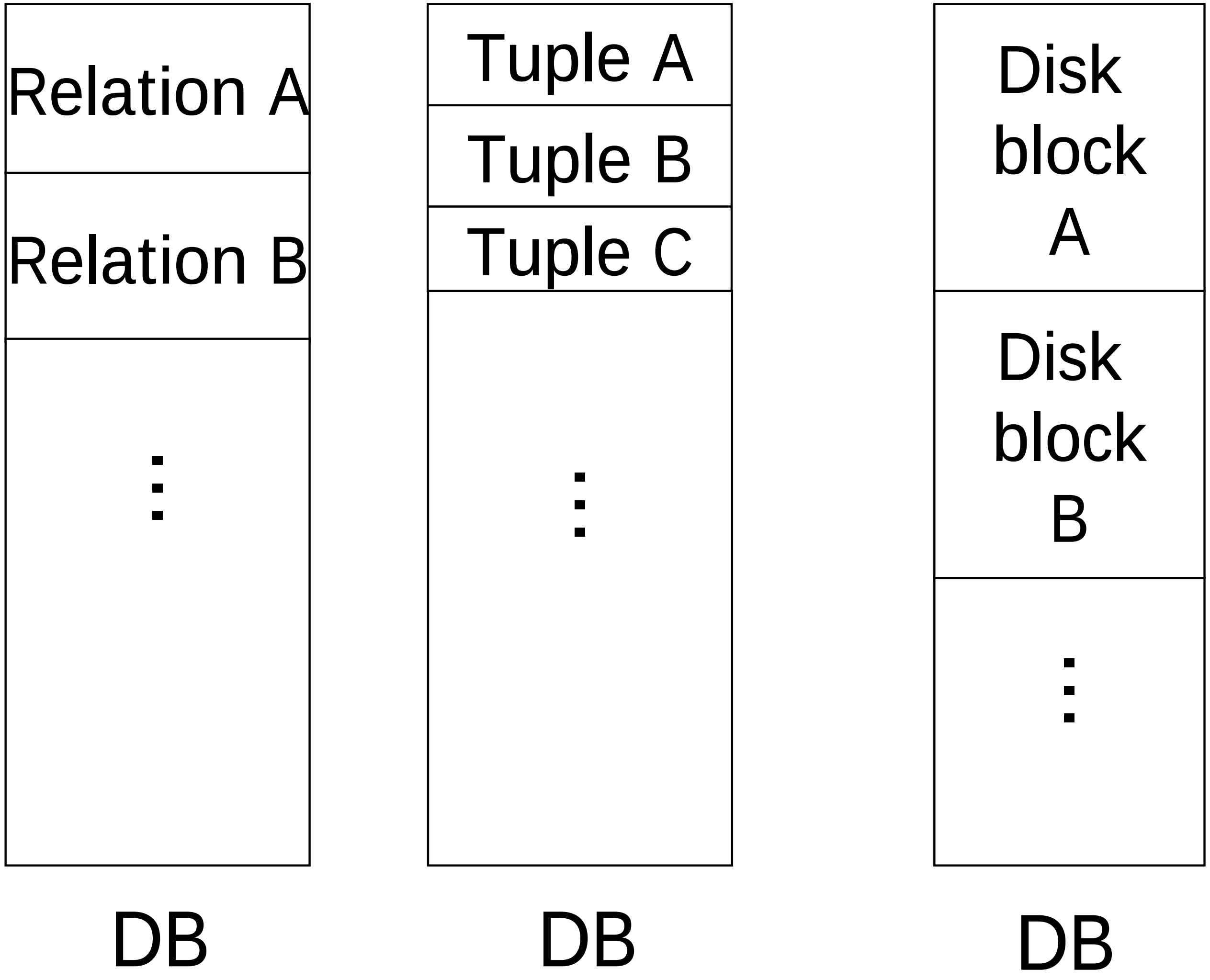
Transaction Management

Concurrency Control (4)

What are the Objects We Lock?

- *Database elements* can be tuples, blocks or entire relations.



Relation A

Relation B

⋮

DB

Tuple A

Tuple B

Tuple C

⋮

DB

Disk block A

Disk block B

⋮

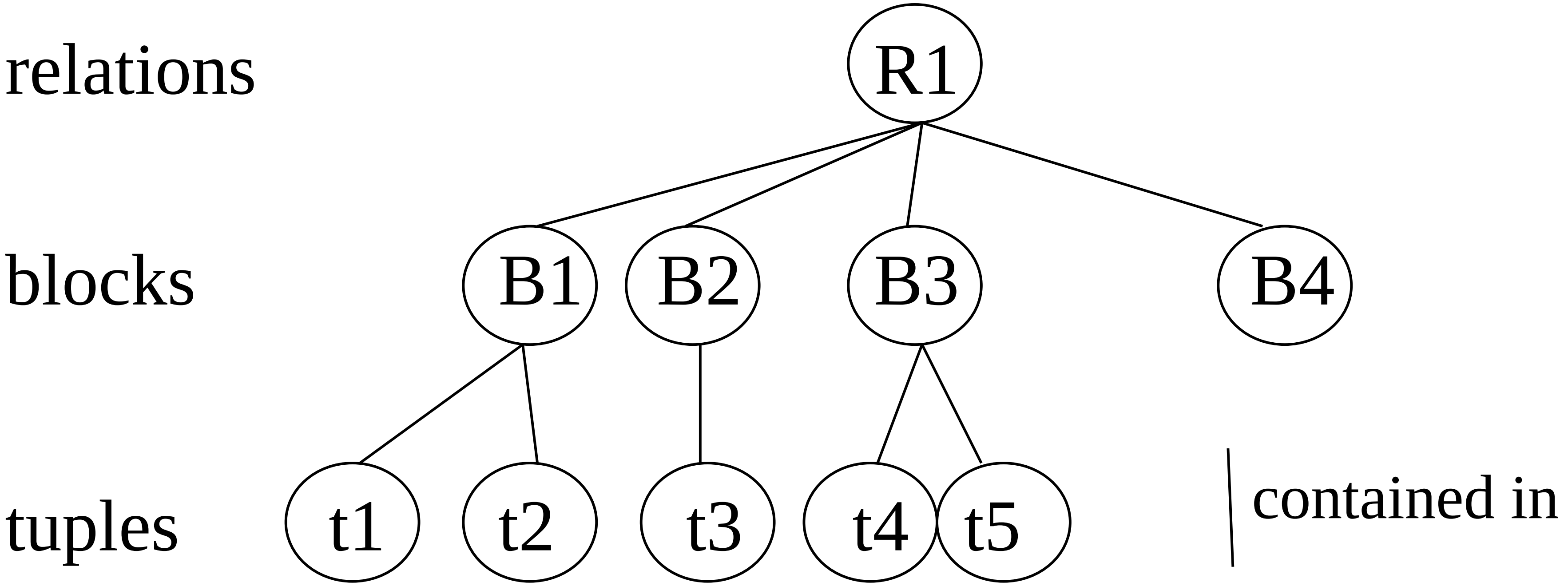
DB

Locks With Multiple Granularity

- Locking works in any case, but should we choose large objects or small objects? At which level of granularity shall we lock?
- There is a trade-off: the lower the level of granularity, the more concurrency, but the more locks and the higher the locking overhead.
- Best trade-off depends on application: e.g., lock blocks or tuples in bank database, and entire documents in document database.

Locks With Multiple Granularity

- Even within the same application, there may be a need for locks at multiple levels of granularity.
- Database elements are organized in a *hierarchy*:



Locks With Multiple Granularity

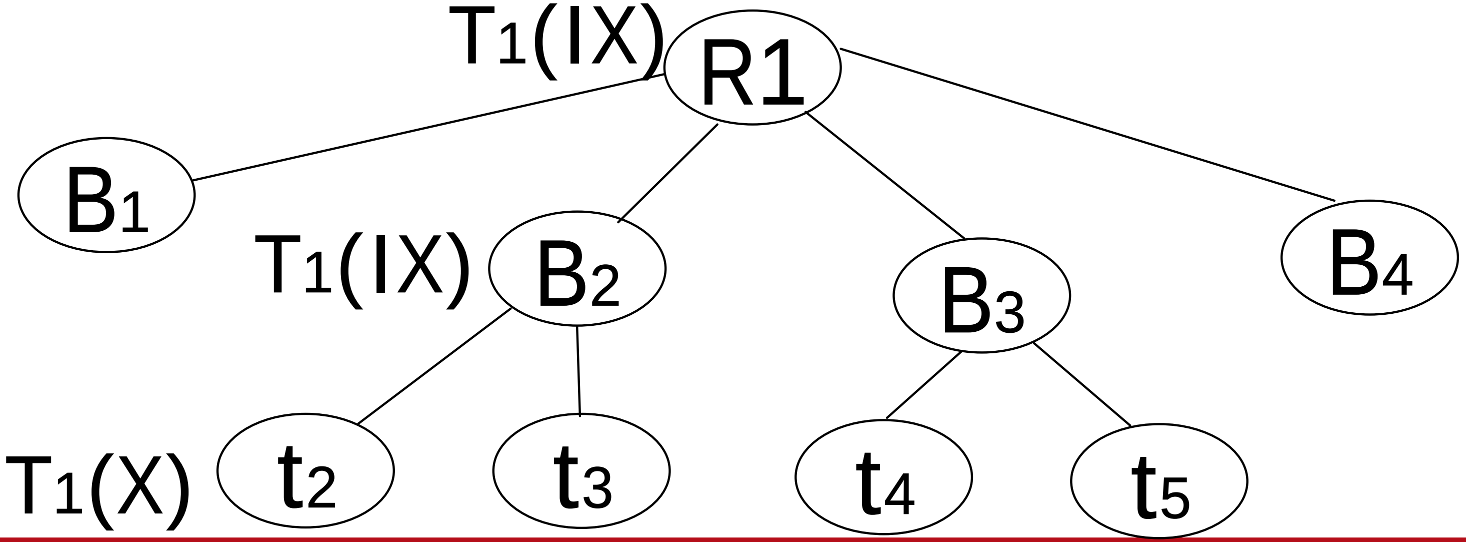
- The *warning protocol* manages locks on a hierarchy of database elements.
- We introduce two new types of locks:

- *IS*: *intention* to request an S lock and
- *IX*: *intention* to request an X lock.

- An IS (IX) lock expresses the intention to request an S (X) lock for a subelement further down in the hierarchy.

Locks With Multiple Granularity

- To request an S (or X) lock on some database element A, we traverse a path from the root of the hierarchy to element A.
- If we have reached A, we request the S (X) lock.
- Otherwise, we request an IS (IX) lock.
- As soon as we have obtained the requested lock, we proceed to the corresponding child (if necessary).



Locks With Multiple Granularity

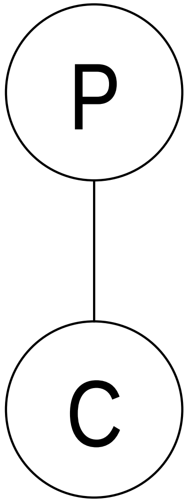
- Compatibility matrix

		Requester			
		IS	IX	S	X
Holder	IS	Yes	Yes	Yes	No
	IX	Yes	Yes	No	No
	S	Yes	No	Yes	No
	X	No	No	No	No

- If two transactions intend to read / write a subelement, we can grant both of them an I lock and resolve the potential conflict at a lower level.

Locks With Multiple Granularity

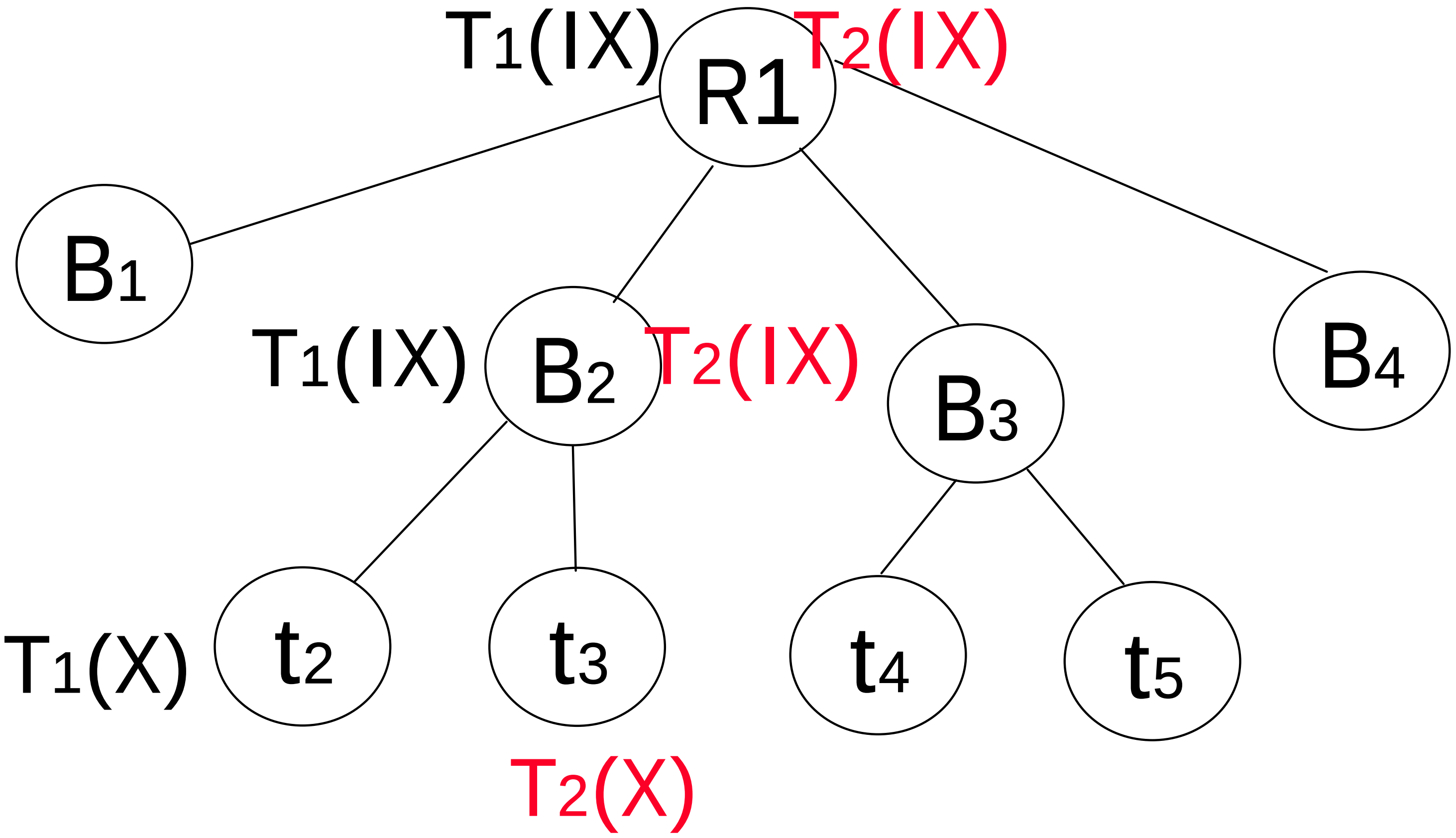
- An I lock for a superelement constrains the locks that the same transaction can obtain at a subelement.
- If Ti has locked the parent element P in IS, then Ti can lock child element C in IS, S.
- If Ti has locked the parent element P in IX, then Ti can lock child element C in IS, S, IX, X.



Locks With Multiple Granularity

- Example

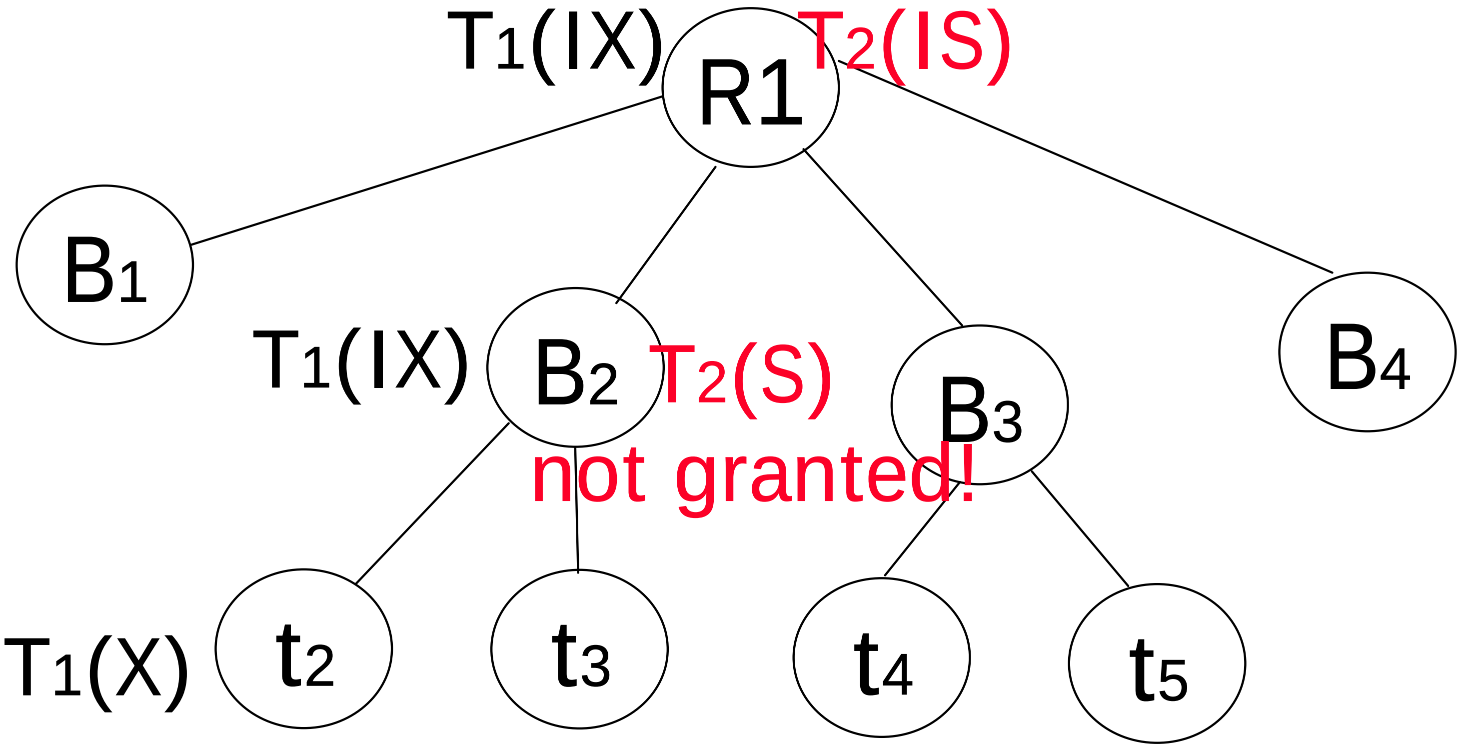
  T2 wants to request an X lock on tuple t3



Locks With Multiple Granularity

- Example

  T2 wants to request an S lock on block B2



How Does Locking Work In Practice?

- Every system is different

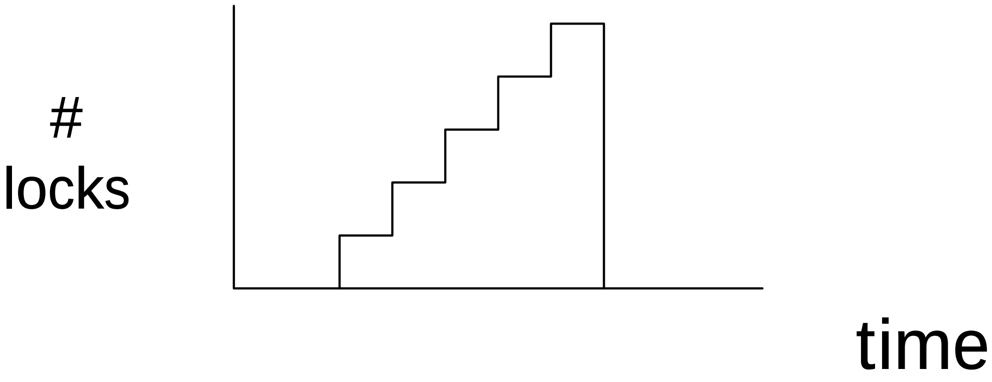
  (E.g., may not even provide CONFLICT-SERIALIZABLE schedules)

- But here is one (simplified) way ...

Sample Locking System:

(1) Don’t trust transactions to request/release locks;

(2) Hold all locks until transaction commits



Optimistic Concurrency Control

- *Optimistic approaches* to concurrency control assume that unserializable schedules are infrequent.
- Unlike in *pessimistic approaches* (locking), unserializable schedules are not prevented, but detected and some of the transactions aborted.
- The two main optimistic approaches are timestamping (not covered in class) and validation (next section).

Concurrency Control by Validation

- We allow transactions to proceed without locking.
- All DB modifications are made on a local copy.
- At the appropriate time, we check whether the transaction schedule is serializable.
- If so, the modifications of the local copy are applied to the global DB.
- Otherwise, the local modifications are discarded, and the transaction is re-started.

Concurrency Control by Validation

- For each transaction T, the scheduler maintains two sets of relevant database elements:
  - RS(T), the *read set* of T: the set of all database elements read by T.
  - WS(T), the *write set* of T: the set of all database elements written by T.
- This information is crucial to determine whether some schedule that has already been executed was indeed serializable.

Concurrency Control by Validation

- Transaction T is executed in three phases:

1. *Read*: transaction reads all elements in its read set from DB and is executes all its actions in its local address space.
2. *Validate*: the serializability of the schedule is checked by comparing RS(T) and WS(T) to the read / write sets of the concurrent transactions. If validation is unsuccessful, skip phase 3.
3. *Write*: write the new values of the elements in WS(T) back to the DB.

Concurrency Control by Validation

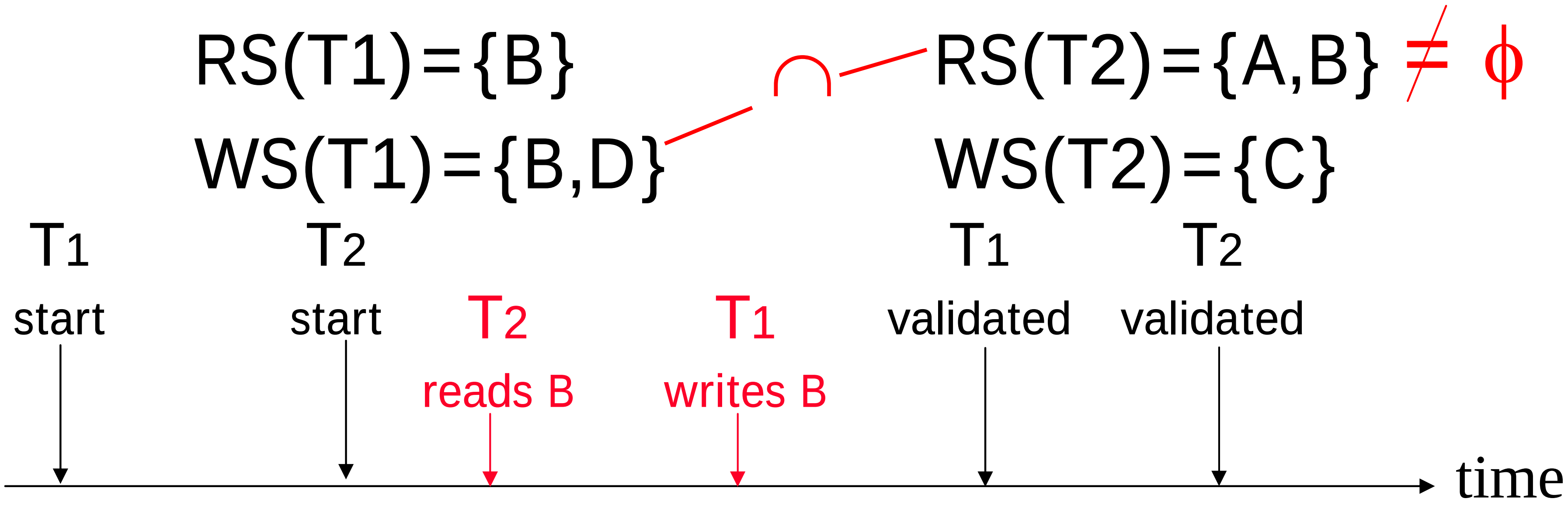
- At any time, the scheduler maintains three sets of transactions and some relevant information.
- *START*: set of transactions that have started, but have not yet completed their validation phase. For each element T of START, keep START(T).
- *VAL*: set of transactions that have completed validation, but not yet their write phase. For elements T of VAL, record VAL(T).
- *FIN*: set of transactions that have completed all three phases. For T in FIN, keep FIN(T).

Concurrency Control by Validation

- Make validation an atomic operation.
- If T1, T2, T3, … is validation order, then the resulting schedule will be conflict equivalent to serial schedule S = T1, T2, T3.
- Can think of each transaction that successfully validates as executing entirely at the moment that it validates.

Concurrency Control by Validation

- Example



$RS(T1)=\{B\}$

$WS(T1)=\{B,D\}$ $\cap$ $RS(T2)=\{A,B\} \neq \phi$

$WS(T2)=\{C\}$

T_1 start

T_2 start

T_2 reads B

T_1 writes B

T_1 validated

T_2 validated

time

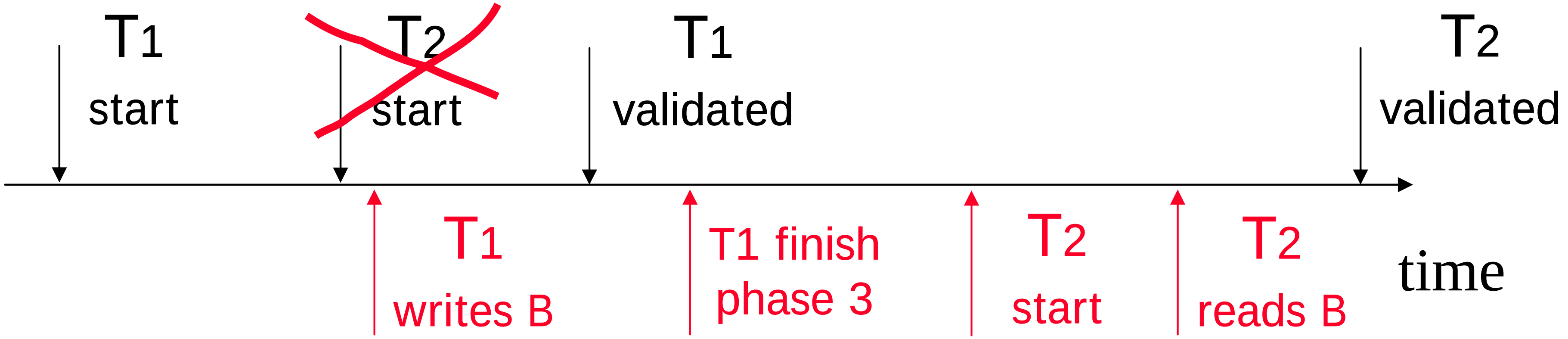
- It is possible that T1 wrote database element B after T2 has read it.
- Schedule is not conflict-equivalent to T1,T2.

Concurrency Control by Validation

- Example

  $RS(T_1)=\{B\}$ $\quad \cap \quad$ $RS(T_2)=\{A,B\}$ $\neq \phi$

  $WS(T_1)=\{B,D\}$ $\quad$ $WS(T_2)=\{C\}$



T_1 start

~~T_2 start~~

T_1 validated

T_2 validated

T_1 writes B

T1 finish phase 3

T_2 start

T_2 reads B

time

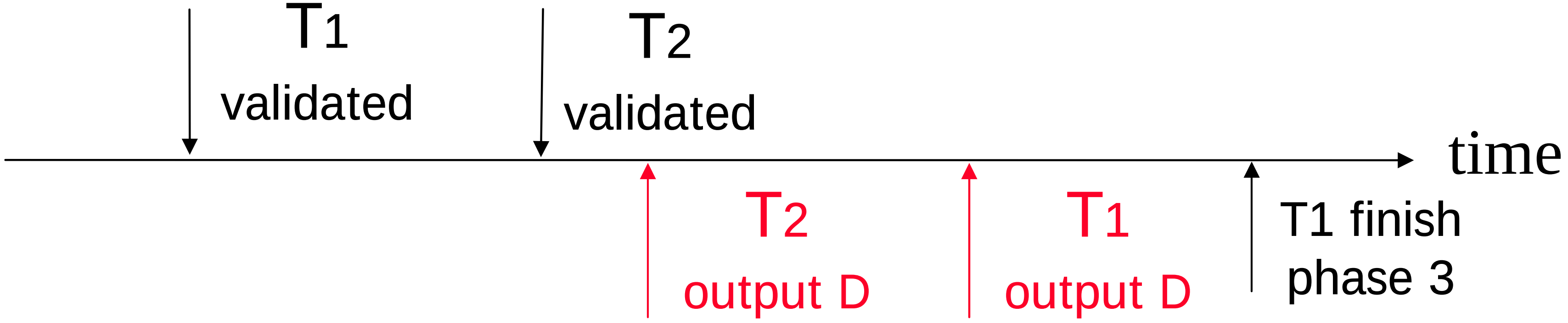
- New value of B written by T1 must have been written back to the DB before T2 has read B.
- Schedule is conflict-equivalent to T1, T2.

Concurrency Control by Validation

- Example

$RS(T_1)=\{A\}$ $RS(T_2)=\{A,B\}$

$WS(T_1)=\{D,E\} - \cap - WS(T_2)=\{C,D\} \neq \phi$



T_1 validated

T_2 validated

time

T_2 output D

T_1 output D

T1 finish phase 3

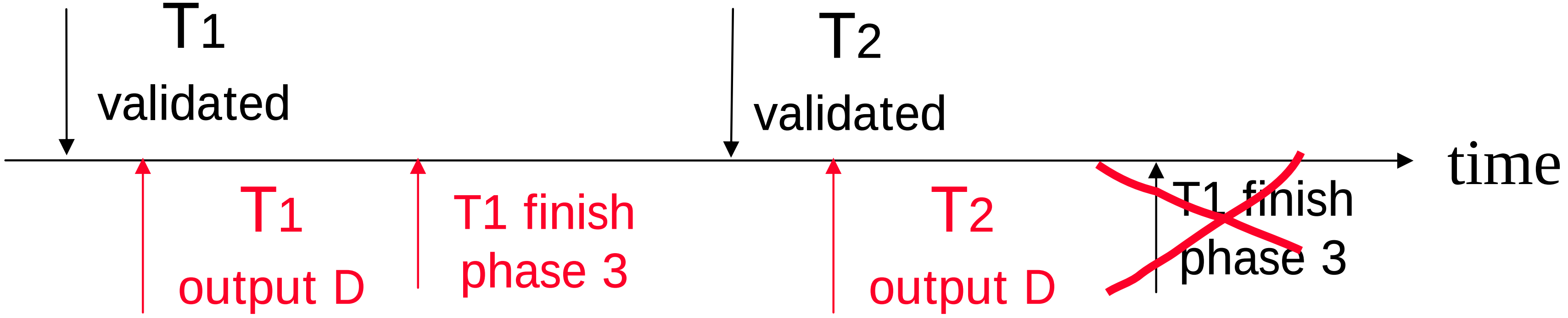
- The new value of D written by T1 may be output to the DB later than the new value written by T2.
- Schedule is not conflict-equivalent to T1, T2.

Concurrency Control by Validation

- Example

$RS(T_1)=\{A\}$ $RS(T_2)=\{A,B\}$

$WS(T_1)=\{D,E\} \cap WS(T_2)=\{C,D\} \neq \phi$



T_1 validated

T_2 validated

time

T_1 output D

T1 finish phase 3

T_2 output D

~~T1 finish phase 3~~

- The new value of D written by T1 must be output to the DB earlier than the new value of D written by T2.
- Schedule is conflict-equivalent to T1, T2.

Concurrency Control by Validation

- The above examples motivate the following two *validation rules* for a given transaction T2.
- We consider all transactions T1 that have validated before T2.
- For all T1 with FIN(T1) > START(T2):

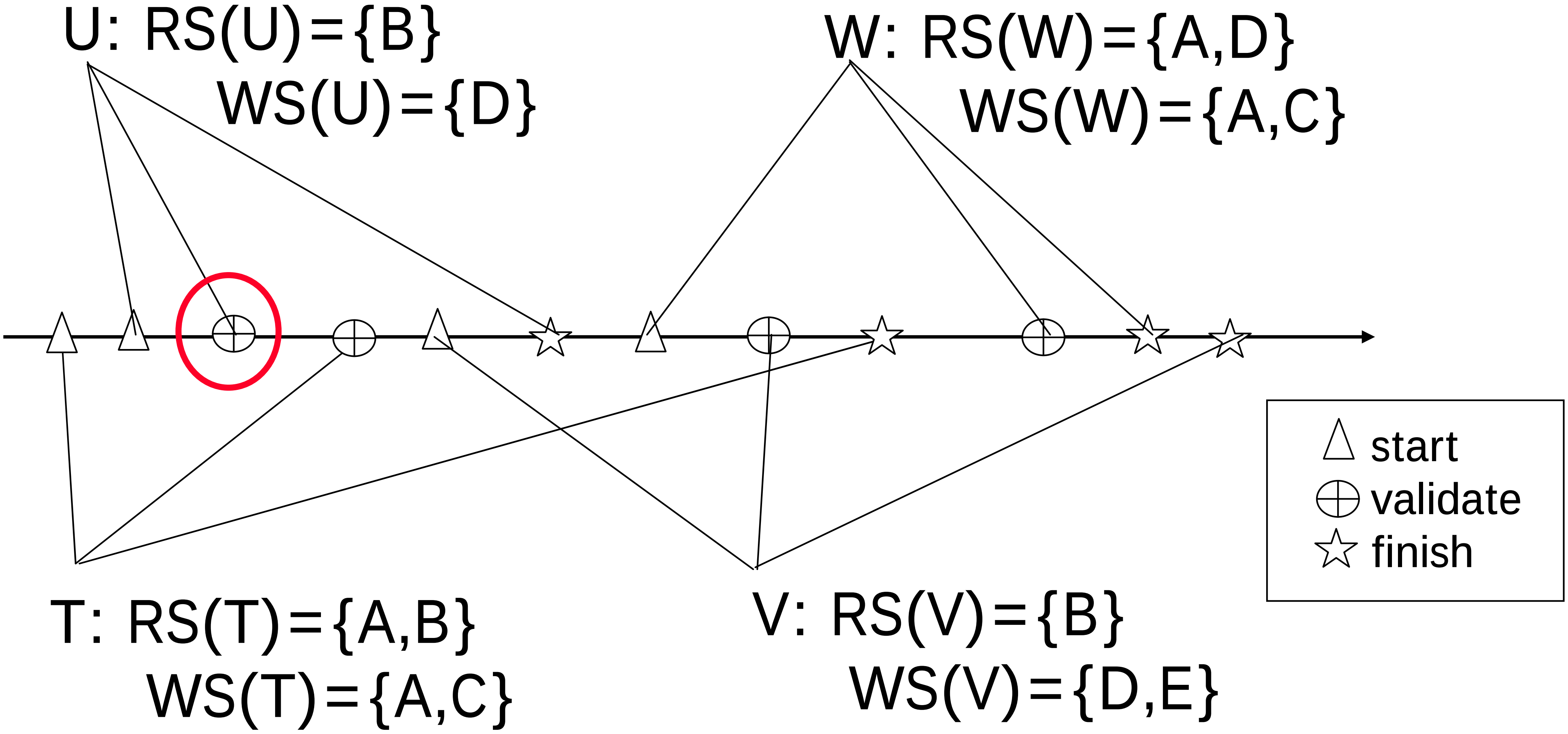
$$RS(T2) \cap WS(T1) = \emptyset.$$

- For all T1 with FIN(T1) > VAL(T2):

$$WS(T2) \cap WS(T1) = \emptyset.$$

- If T2 does successfully validate, if the two validation rules are satisfied for all these T1.

Concurrency Control by Validation



U: $RS(U)=\{B\}$
$WS(U)=\{D\}$

W: $RS(W)=\{A,D\}$
$WS(W)=\{A,C\}$

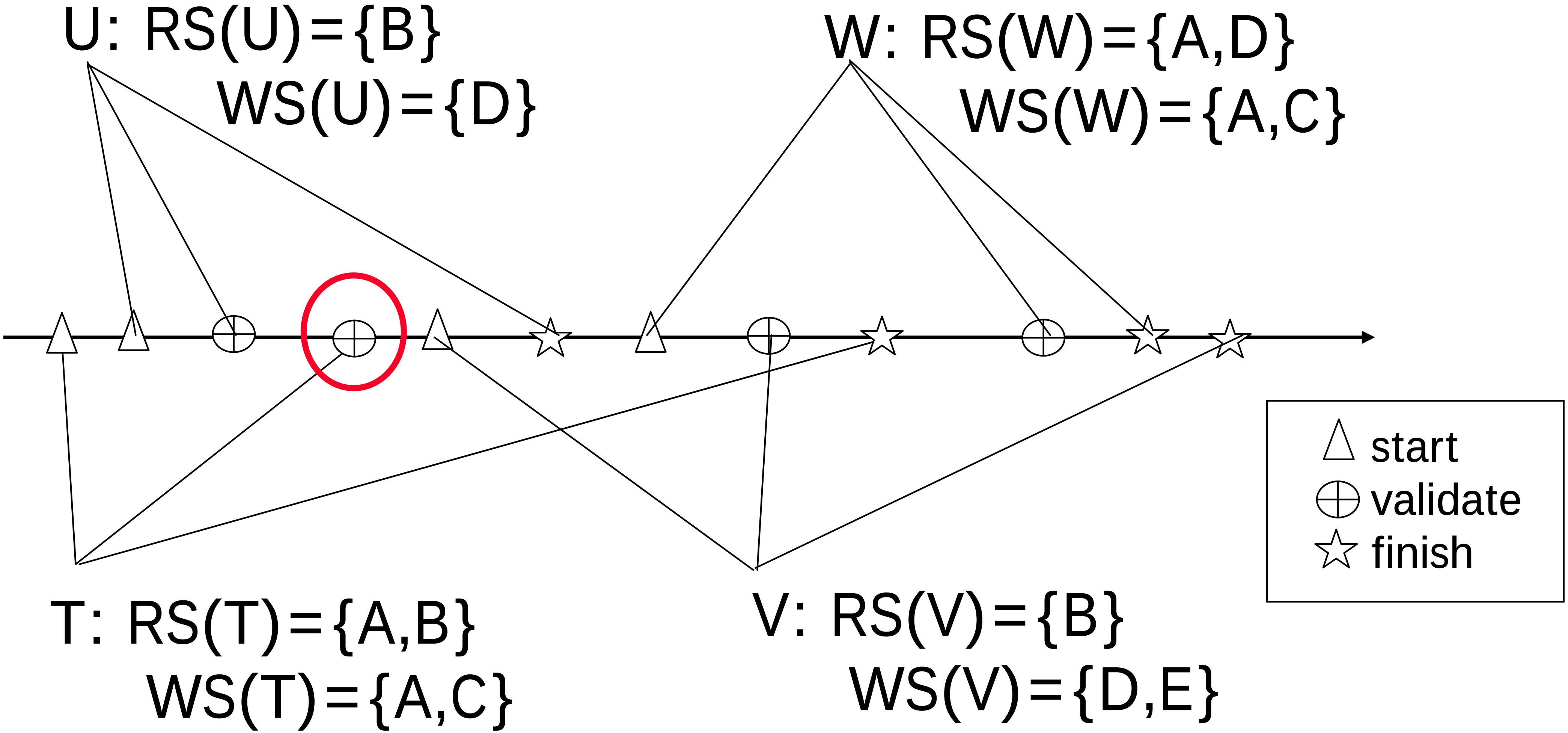
T: $RS(T)=\{A,B\}$
$WS(T)=\{A,C\}$

V: $RS(V)=\{B\}$
$WS(V)=\{D,E\}$

△ start
⊕ validate
☆ finish

- U validates successfully, since there are no other transactions that have validated before U.

Concurrency Control by Validation



U: RS(U)={B}
WS(U)={D}

W: RS(W)={A,D}
WS(W)={A,C}

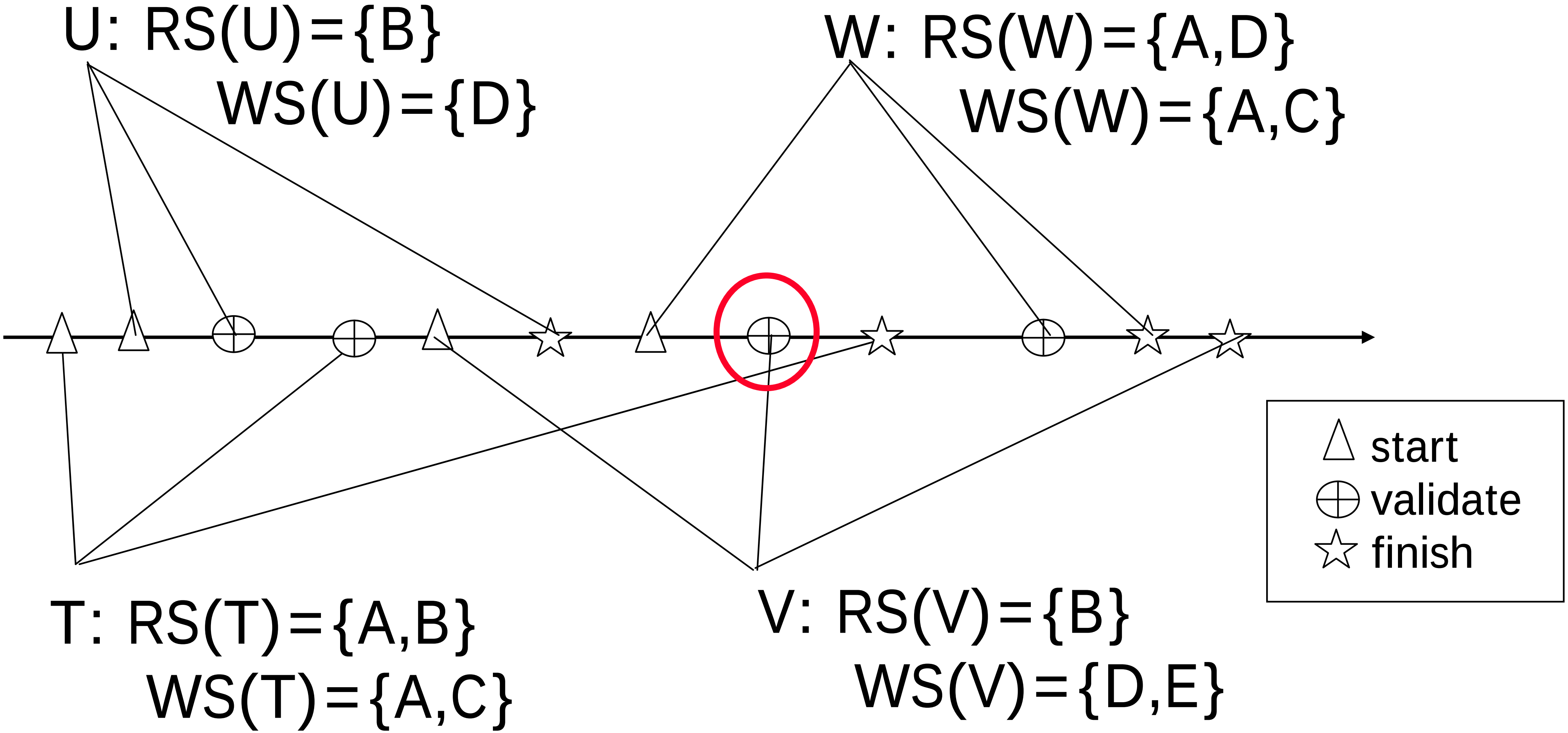
T: RS(T)={A,B}
WS(T)={A,C}

V: RS(V)={B}
WS(V)={D,E}

△ start
⊕ validate
☆ finish

- T validates successfully, since RS(T) and WS(T) have no intersection with WS(U).

Concurrency Control by Validation



U: RS(U)={B}
WS(U)={D}

W: RS(W)={A,D}
WS(W)={A,C}

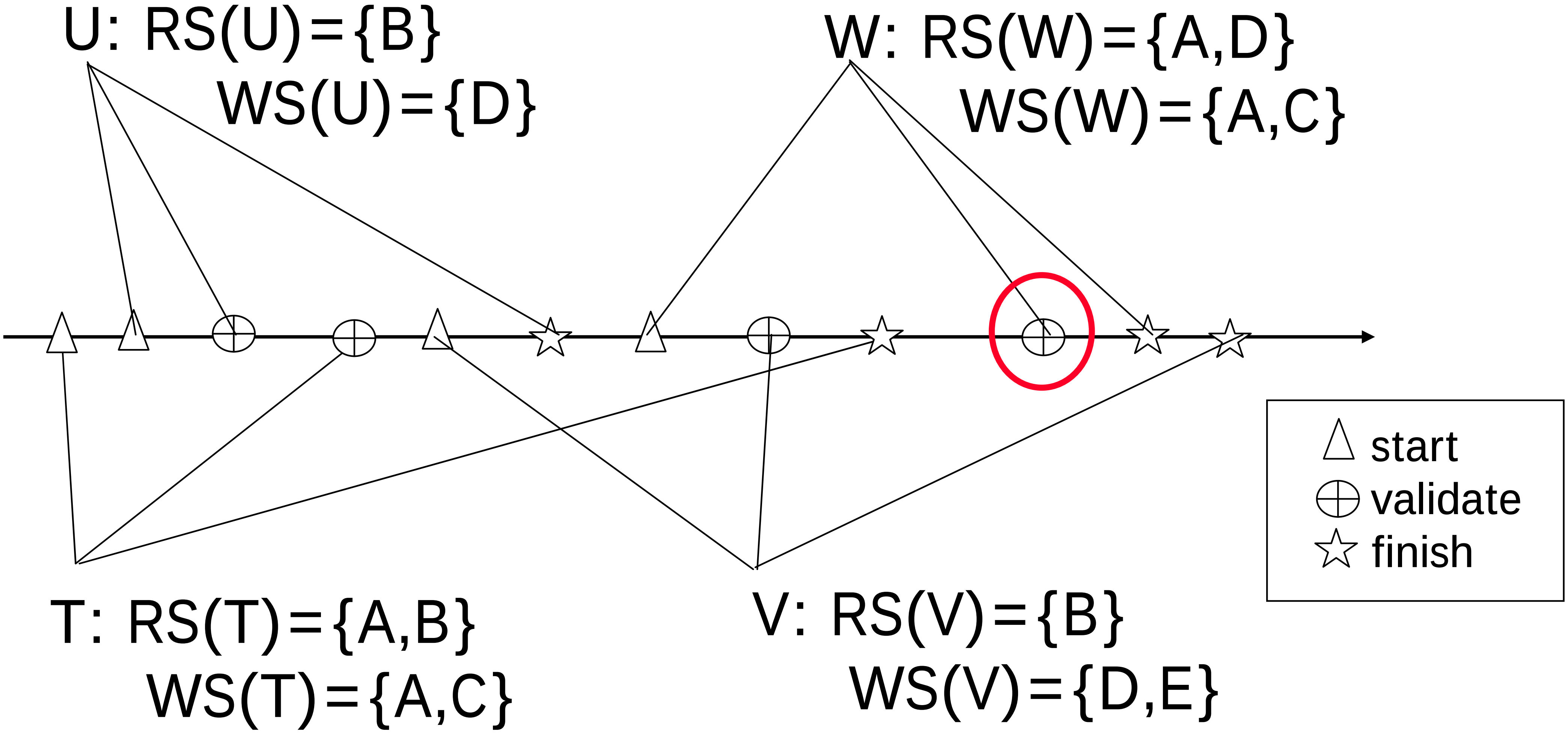
T: RS(T)={A,B}
WS(T)={A,C}

V: RS(V)={B}
WS(V)={D,E}

△ start
⊕ validate
☆ finish

- V validates successfully, since RS(V) has no intersection with WS(U) and FIN(U) < VAL(V) and neither RS(V) nor WS(V) have intersection with WS(T).

Concurrency Control by Validation



U: RS(U)={B}
WS(U)={D}

W: RS(W)={A,D}
WS(W)={A,C}

T: RS(T)={A,B}
WS(T)={A,C}

V: RS(V)={B}
WS(V)={D,E}

- W validates unsuccessfully, since RS(W) has intersection with WS(V) and FIN(V) > START(W).

Concurrency Control Mechanisms

- We conclude by comparing pessimistic and optimistic concurrency control mechanisms.
- Locking delays transactions, but avoids rollbacks.
- Validation does not delay transactions, but can cause a rollback (and re-start).
- Rollbacks may waste a lot of resources.
- If interactions between transactions are infrequent, then there will be few rollbacks, and validation will be more efficient.

Next to Discuss

- Serialibility and Recoverability (Chapter 19.1)
- Deadlocks (Chapter 19.2)

To-Do-List

- Is Validation = 2PL?