Chapter 9: Concurrency Control

- Concurrency, Conflicts, and Schedules
- Locking Based Algorithms
- Timestamp Ordering Algorithms
- Deadlock Management

Acknowledgements: I am indebted to Arturas Mazeika for providing me his slides of this course.

Concurrency

- **Concurrency control** is the problem of synchronizing concurrent transactions (i.e., order the operations of concurrent transactions) such that the following two properties are achieved:
  - the consistency of the DB is maintained
  - the maximum degree of concurrency of operations is achieved
- Obviously, the serial execution of a set of transaction achieves consistency, if each single transaction is consistent

Conflicts

- **Conflicting operations:** Two operations $O_{ij}(x)$ and $O_{kl}(x)$ of transactions T_i and T_k are in **conflict** iff at least one of the operations is a write, i.e.,
  - $O_{ij} = read(x)$ and $O_{kl} = write(x)$
  - $O_{ij} = write(x)$ and $O_{kl} = read(x)$
  - $O_{ij} = write(x)$ and $O_{kl} = write(x)$
- Intuitively, a conflict between two operations indicates that their order of execution is important.
- Read operations do not conflict with each other, hence the ordering of read operations does not matter.
- **Example:** Consider the following two transactions

T_1:	$Read(x)$	T_2:	$Read(x)$
	$x \leftarrow x + 1$		$x \leftarrow x + 1$
	$Write(x)$		$Write(x)$
	$Commit$		$Commit$

  - To preserve DB consistency, it is important that the $read(x)$ of one transaction is not between $read(x)$ and $write(x)$ of the other transaction.

Schedules

- A **schedule** (history) specifies a possibly interleaved order of execution of the operations O of a set of transactions $T = \{T_1, T_2, \ldots, T_n\}$, where T_i is specified by a partial order $(\Sigma_i, \prec_i)$. A schedule can be specified as a partial order over O, where
  - $\Sigma_T = \bigcup_{i=1}^{n} \Sigma_i$
  - $\prec_T \supseteq \bigcup_{i=1}^{n} \prec_i$
  - For any two conflicting operations $O_{ij}, O_{kl} \in \Sigma_T$, either $O_{ij} \prec_T O_{kl}$ or $O_{kl} \prec_T O_{ij}$

Schedules ...

- **Example:** Consider the following two transactions

T_1:	$Read(x)$	T_2:	$Read(x)$
	$x \leftarrow x + 1$		$x \leftarrow x + 1$
	$Write(x)$		$Write(x)$
	$Commit$		$Commit$

  – A possible schedule over $T = \{T_1, T_2\}$ can be written as the partial order $S = \{\Sigma_T, \prec_T\}$, where

$$\Sigma_T = \{R_1(x), W_1(x), C_1, R_2(x), W_2(x), C_2\}$$

$$\begin{aligned}\prec_T = \{&(R_1, W_1), (R_1, C_1), (W_1, C_1),\\ &(R_2, W_2), (R_2, C_2), (W_2, C_2),\\ &(R_2, W_1), (W_1, W_2), \ldots\}\end{aligned}$$

Schedules ...

- A schedule is **serial** if all transactions in T are executed serially.
- **Example:** Consider the following two transactions

T_1: $Read(x)$
$x \leftarrow x + 1$
$Write(x)$
$Commit$

T_2: $Read(x)$
$x \leftarrow x + 1$
$Write(x)$
$Commit$

  – The two serial schedules are $S_1 = \{\Sigma_1, \prec_1\}$ and $S_2 = \{\Sigma_2, \prec_2\}$, where

$$\Sigma_1 = \Sigma_2 = \{R_1(x), W_1(x), C_1, R_2(x), W_2(x), C_2\}$$
$$\prec_1 = \{(R_1, W_1), (R_1, C_1), (W_1, C_1), (R_2, W_2), (R_2, C_2), (W_2, C_2), (C_1, R_2), \dots\}$$
$$\prec_2 = \{(R_1, W_1), (R_1, C_1), (W_1, C_1), (R_2, W_2), (R_2, C_2), (W_2, C_2), (C_2, R_1), \dots\}$$

- We will also use the following notation:
  – $\{T_1, T_2\} = \{R_1(x), W_1(x), C_1, R_2(x), W_2(x), C_2\}$
  – $\{T_2, T_1\} = \{R_2(x), W_2(x), C_2, R_1(x), W_1(x), C_1\}$

Serializability

- Two schedules are said to be **equivalent** if they have the same effect on the DB.
- **Conflict equivalence:** Two schedules S_1 and S_2 defined over the same set of transactions $T = \{T_1, T_2, \ldots, T_n\}$ are said to be **conflict equivalent** if for each pair of conflicting operations O_{ij} and O_{kl}, whenever $O_{ij} <_1 O_{kl}$ then $O_{ij} <_2 O_{kl}$.
  - i.e., conflicting operations must be executed in the same order in both transactions.
- A concurrent schedule is said to be **(conflict-)serializable** iff it is conflict equivalent to a serial schedule
- A conflict-serializable schedule can be transformed into a serial schedule by swapping non-conflicting operations
- **Example:** Consider the following two schedules

T_1: $Read(x)$
$x \leftarrow x + 1$
$Write(x)$
$Write(z)$
$Commit$

T_2: $Read(x)$
$x \leftarrow x + 1$
$Write(x)$
$Commit$

  - The schedule $\{R_1(x), W_1(x), R_2(x), W_2(x), W_1(z), C_2, C_1\}$ is conflict-equivalent to $\{T_1, T_2\}$ but not to $\{T_2, T_1\}$

Serializability . . .

- The **primary function** of a concurrency controller is to generate a serializable schedule for the execution of pending transactions.
- In a DDBMS two schedules must be considered
  - Local schedule
  - Global schedule (i.e., the union of the local schedules)
- **Serializability** in DDBMS
  - Extends in a straightforward manner to a DDBMS if data is *not replicated*
  - Requires more care if data is *replicated*: It is possible that the local schedules are serializable, but the mutual consistency of the DB is not guaranteed.
    - Mutual consistency: All the values of all replicated data items are identical
- Therefore, a **serializable global schedule** must meet the following conditions:
  - Local schedules are serializable
  - Two conflicting operations should be in the same relative order in all of the local schedules they appear
    - Transaction needs to be run on each site with the replicated data item

Serializability . . .

- **Example:** Consider two sites and a data item x which is replicated at both sites.

$$T_1: \quad Read(x) \quad x \leftarrow x + 5 \quad Write(x)$$

$$T_2: \quad Read(x) \quad x \leftarrow x * 10 \quad Write(x)$$

  - Both transactions need to run on both sites
  - The following two schedules might have been produced at both sites (the order is implicitly given):
    - Site1: $S_1 = \{R_1(x), W_1(x), R_2(x), W_2(x)\}$
    - Site2: $S_2 = \{R_2(x), W_2(x), R_1(x), W_1(x)\}$
  - Both schedules are (trivially) serializable, thus are correct in the local context
  - But they produce different results, thus violate the mutual consistency

Concurrency Control Algorithms

- **Taxonomy** of concurrency control algorithms
  - **Pessimistic** methods assume that many transactions will conflict, thus the concurrent execution of transactions is synchronized early in their execution life cycle
    - Two-Phase Locking (2PL)
      - Centralized (primary site) 2PL
      - Primary copy 2PL
      - Distributed 2PL
    - Timestamp Ordering (TO)
      - Basic TO
      - Multiversion TO
      - Conservative TO
    - Hybrid algorithms
  - **Optimistic** methods assume that not too many transactions will conflict, thus delay the synchronization of transactions until their termination
    - Locking-based
    - Timestamp ordering-based

Locking Based Algorithms

- **Locking-based concurrency algorithms** ensure that data items shared by conflicting operations are accessed in a mutually exclusive way. This is accomplished by associating a "lock" with each such data item.
- Two types of **locks** (lock modes)
  - **read lock** (rl) – also called **shared** lock
  - **write lock** (wl) – also called **exclusive** lock
- **Compatibility matrix** of locks

	$rl_i(x)$	$wl_i(x)$
$rl_j(x)$	compatible	not compatible
$wl_j(x)$	not compatible	not compatible

- General locking algorithm
  1. Before using a data item x, transaction requests lock for x from the lock manager
  2. If x is already locked and the existing lock is incompatible with the requested lock, the transaction is delayed
  3. Otherwise, the lock is granted

Locking Based Algorithms

- **Example:** Consider the following two transactions

T_1:	$Read(x)$	T_2:	$Read(x)$
	$x \leftarrow x + 1$		$x \leftarrow x * 2$
	$Write(x)$		$Write(x)$
	$Read(y)$		$Read(y)$
	$y \leftarrow y - 1$		$y \leftarrow y * 2$
	$Write(y)$		$Write(y)$

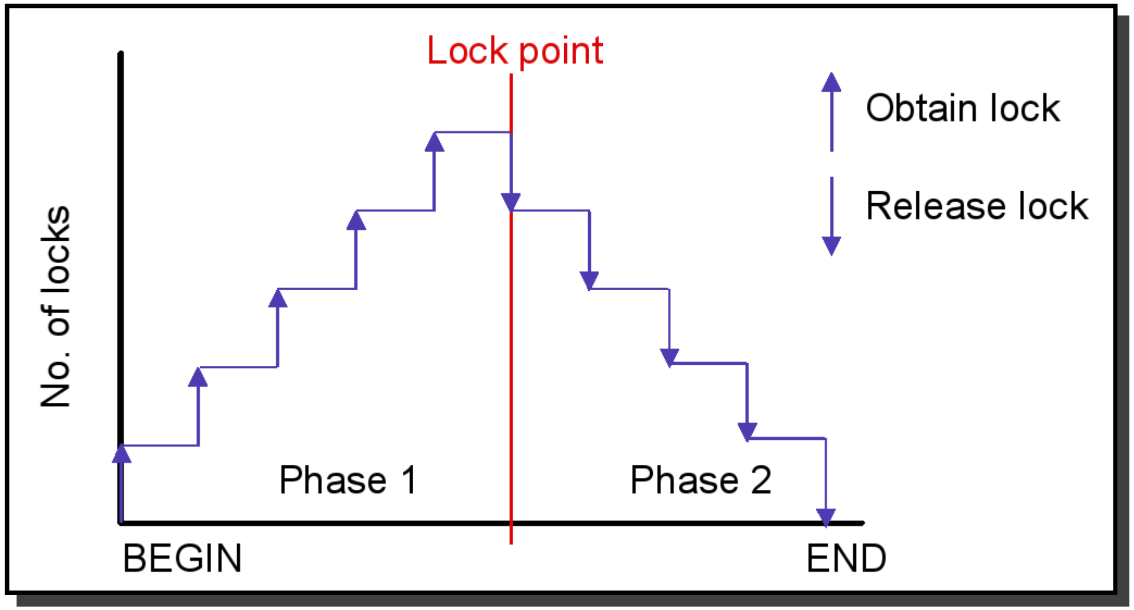
  - The following schedule is a valid locking-based schedule ($lr_i(x)$ indicates the release of a lock on x):

$$
\begin{aligned}
S = \{ & wl_1(x), R_1(x), W_1(x), lr_1(x) \\
& wl_2(x), R_2(x), W_2(x), lr_2(x) \\
& wl_2(y), R_2(y), W_2(y), lr_2(y) \\
& wl_1(y), R_1(y), W_1(y), lr_1(y)\}
\end{aligned}
$$

  - However, S is not serializable
    - S cannot be transformed into a serial schedule by using only non-conflicting swaps
    - The result is different from the result of any serial execution

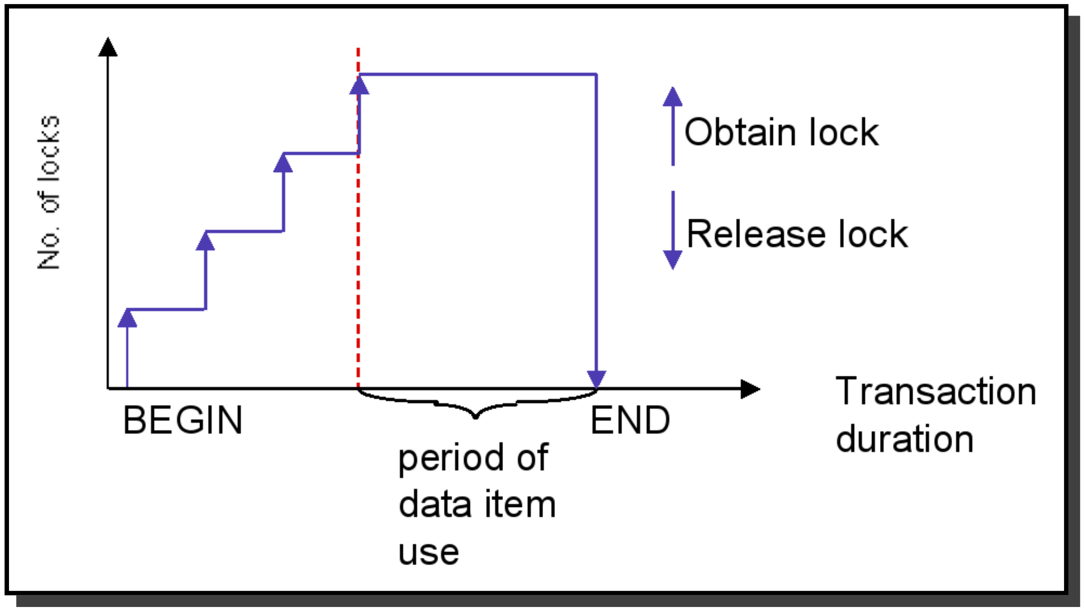
Two-Phase Locking (2PL)

- **Two-phase locking** protocol
  - Each transaction is executed in two phases
    - ∗ **Growing phase:** the transaction obtains locks
    - ∗ **Shrinking phase:** the transaction releases locks
  - The **lock point** is the moment when transitioning from the growing phase to the shrinking phase



Two-Phase Locking (2PL) . . .

- **Properties** of the 2PL protocol
  - Generates **conflict-serializable** schedules
  - But schedules may cause **cascading aborts**
    - ∗ If a transaction aborts after it releases a lock, it may cause other transactions that have accessed the unlocked data item to abort as well
- **Strict 2PL locking** protocol
  - Holds the locks till the end of the transaction
  - Cascading aborts are avoided



- **Example:** The schedule S of the previous example is not valid in the 2PL protocol:

$$
\begin{aligned}
S = \{ & wl_1(x), R_1(x), W_1(x), lr_1(x) \\
& wl_2(x), R_2(x), W_2(x), lr_2(x) \\
& wl_2(y), R_2(y), W_2(y), lr_2(y) \\
& wl_1(y), R_1(y), W_1(y), lr_1(y)\}
\end{aligned}
$$

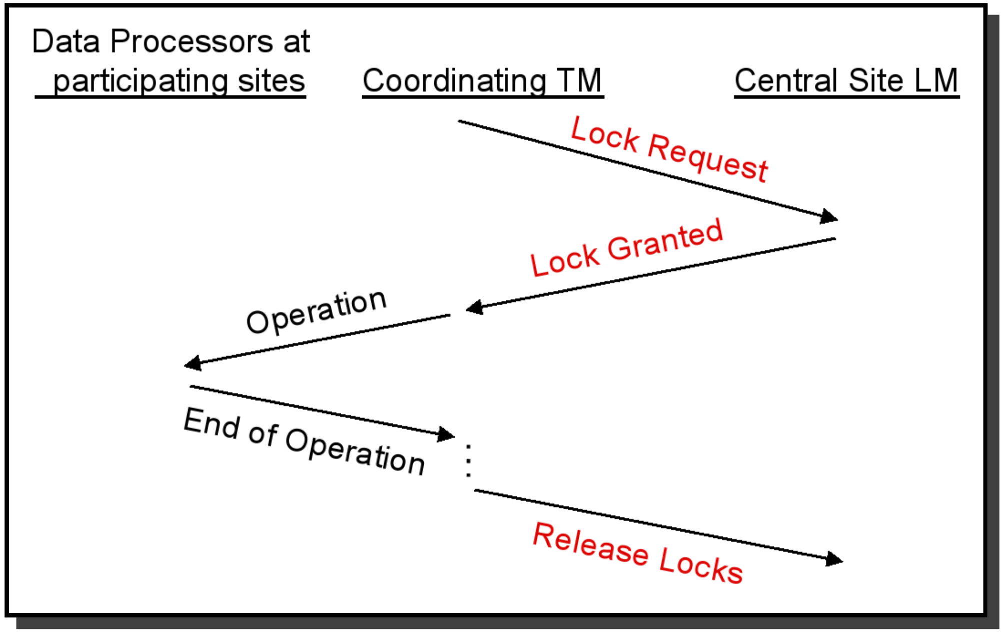
  - e.g., after $lr_1(x)$ (in line 1) transaction T_1 cannot request the lock $wl_1(y)$ (in line 4).
  - Valid schedule in the 2PL protocol

$$
\begin{aligned}
S = \{ & wl_1(x), R_1(x), W_1(x), \\
& wl_1(y), R_1(y), W_1(y), lr_1(x), lr_1(y) \\
& wl_2(x), R_2(x), W_2(x), \\
& wl_2(y), R_2(y), W_2(y), lr_2(x), lr_2(y)\}
\end{aligned}
$$

2PL for DDBMS

- Various extensions of the 2PL to DDBMS
- **Centralized 2PL**
  - A single site is responsible for the lock management, i.e., one lock manager for the whole DDBMS
  - Lock requests are issued to the lock manager
  - Coordinating transaction manager (TM at site where the transaction is initiated) can make all locking requests on behalf of local transaction managers

- Advantage: Easy to implement
- Disadvantages: Bottlenecks and lower reliability
- Replica control protocol is additionally needed if data are replicated (see also primary copy 2PL)



2PL for DDBMS . . .

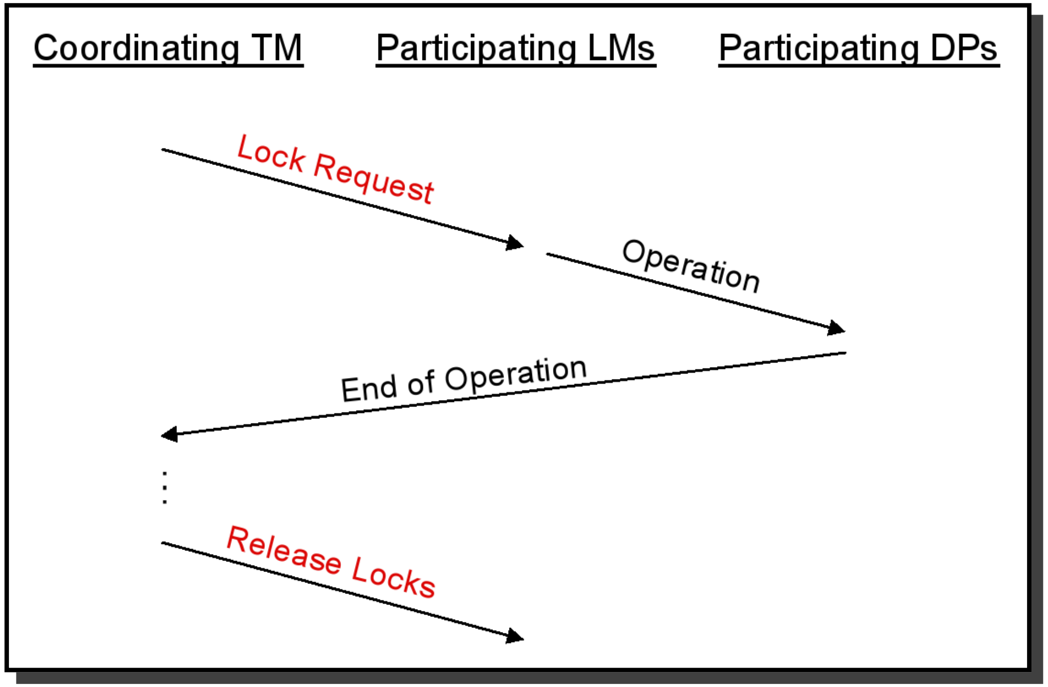
- **Primary copy 2PL**
  - Several lock managers are distributed to a number of sites
  - Each lock manager is responsible for managing the locks for a set of data items
  - For replicated data items, one copy is chosen as primary copy, others are slave copies
  - Only the primary copy of a data item that is updated needs to be write-locked
  - Once primary copy has been updated, the change is propagated to the slaves
- Advantages
  - Lower communication costs and better performance than the centralized 2PL
- Disadvantages
  - Deadlock handling is more complex

2PL for DDBMS . . .

- **Distributed 2PL**
  - Lock managers are distributed to all sites
  - Each lock manager responsible for locks for data at that site
  - If data is not replicated, it is equivalent to primary copy 2PL
  - If data is replicated, the Read-One-Write-All (ROWA) replica control protocol is implemented
    * $Read(x)$: Any copy of a replicated item x can be read by obtaining a read lock on the copy
    * $Write(x)$: All copies of x must be write-locked before x can be updated
- Disadvantages
  - Deadlock handling more complex
  - Communication costs higher than primary copy 2PL

2PL for DDBMS . . .

- Communication structure of the distributed 2PL
  - The coordinating TM sends the lock request to the lock managers of all participating sites
  - The LMs pass the operations to the data processors
  - The end of the operation is signaled to the coordinating TM



Timestamp Ordering

- **Timestamp-ordering** based algorithms do not maintain serializability by mutual exclusion, but select (a priori) a serialization order and execute transactions accordingly.
  - Transaction T_i is assigned a globally unique timestamp $ts(T_i)$
  - Conflicting operations O_{ij} and O_{kl} are resolved by timestamp order, i.e., O_{ij} is executed before O_{kl} iff $ts(T_i) < ts(T_k)$.
- To allow for the scheduler to check whether operations arrive in correct order, each data item is assigned a write timestamp (wts) and a read timestamp (rts):
  - $rts(x)$: largest timestamp of any read on x
  - $wts(x)$: largest timestamp of any write on x
- Then the scheduler has to perform the following checks:
  - Read operation, $R_i(x)$:
    * If $ts(T_i) < wts(x)$: T_i attempts to read overwritten data; abort T_i
    * If $ts(T_i) \geq wts(x)$: the operation is allowed and $rts(x)$ is updated
  - Write operations, $W_i(x)$:
    * If $ts(T_i) < rts(x)$: x was needed before by other transaction; abort T_i
    * If $ts(T_i) < wts(x)$: T_i writes an obsolete value; abort T_i
    * Otherwise, execute $W_i(x)$

Timestamp Ordering . . .

- Generation of **timestamps** (TS) in a distributed environment
  - TS needs to be locally and globally **unique** and **monotonically increasing**
  - System clock, incremental event counter at each site, or global counter are unsuitable (difficult to maintain)
  - Concatenate local timestamp/counter with a unique site identifier: $<$*local timestamp, site identifier*$>$
    * site identifier is in the least significant position in order to distinguish only if the local timestamps are identical
- Schedules generated by the basic TO protocol have the following **properties**:
  - Serializable
  - Since transactions never wait (but are rejected), the schedules are deadlock-free
  - The price to pay for deadlock-free schedules is the potential restart of a transaction several times

Timestamp Ordering . . .

- Basic timestamp ordering is "**aggressive**": It tries to execute an operation as soon as it receives it
- **Conservative** timestamp ordering delays each operation until there is an assurance that it will not be restarted, i.e., that no other transaction with a smaller timestamp can arrive
  - For this, the operations of each transaction are buffered until an ordering can be established so that rejections are not possible
- If this condition can be guaranteed, the scheduler will never reject an operation
- However, this delay introduces the possibility for deadlocks

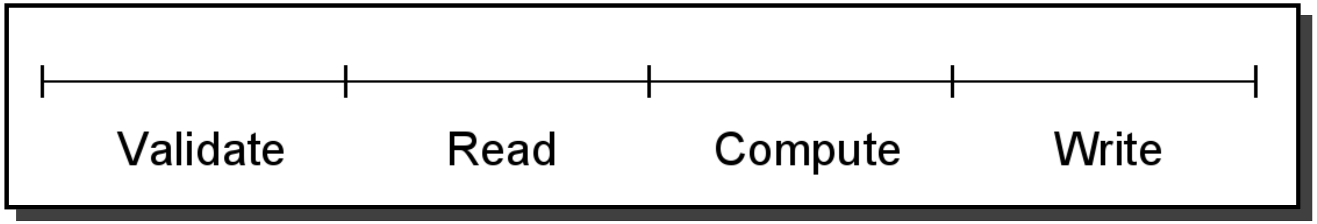
- **Multiversion timestamp ordering**
  - Write operations do not modify the DB; instead, a new version of the data item is created: $x_1, x_2, \ldots, x_n$
  - $R_i(x)$ is always successful and is performed on the appropriate version of x, i.e., the version of x (say x_v) such that $wts(x_v)$ is the largest timestamp less than $ts(T_i)$
  - $W_i(x)$ produces a new version x_w with $ts(x_w) = ts(T_i)$ if the scheduler has not yet processed any $R_j(x_r)$ on a version x_r such that

    $$ts(T_i) < rts(x_r)$$

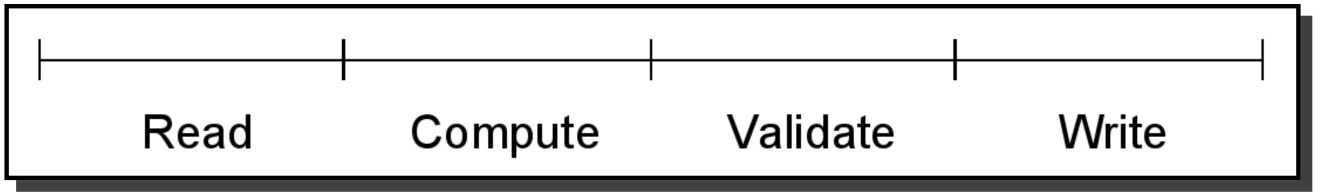
    i.e., the write is too late.
  - Otherwise, the write is rejected.

Timestamp Ordering . . .

- The previous concurrency control algorithms are pessimistic



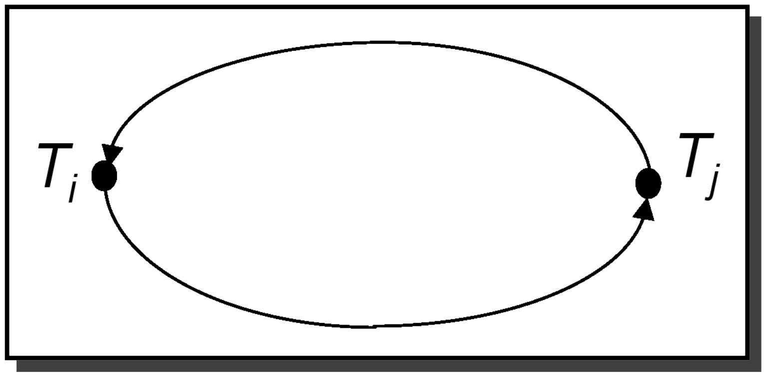
- **Optimistic concurrency control algorithms**
  - Delay the validation phase until just before the write phase
  - T_i run independently at each site on local copies of the DB (without updating the DB)
  - Validation test then checks whether the updates would maintain the DB consistent:
    * If yes, all updates are performed
    * If one fails, all T_i's are rejected



- Potentially allow for a higher level of concurrency

Deadlock Management

- **Deadlock:** A set of transactions is in a deadlock situation if several transactions wait for each other. A deadlock requires an outside intervention to take place.
- Any locking-based concurrency control algorithm may result in a deadlock, since there is mutual exclusive access to data items and transactions may wait for a lock
- Some TO-based algorihtms that require the waiting of transactions may also cause deadlocks
- A **Wait-for Graph** (WFG) is a useful tool to identify deadlocks
  - The nodes represent transactions
  - An edge from T_i to T_j indicates that T_i is waiting for T_j
  - If the WFG has a cycle, we have a deadlock situation

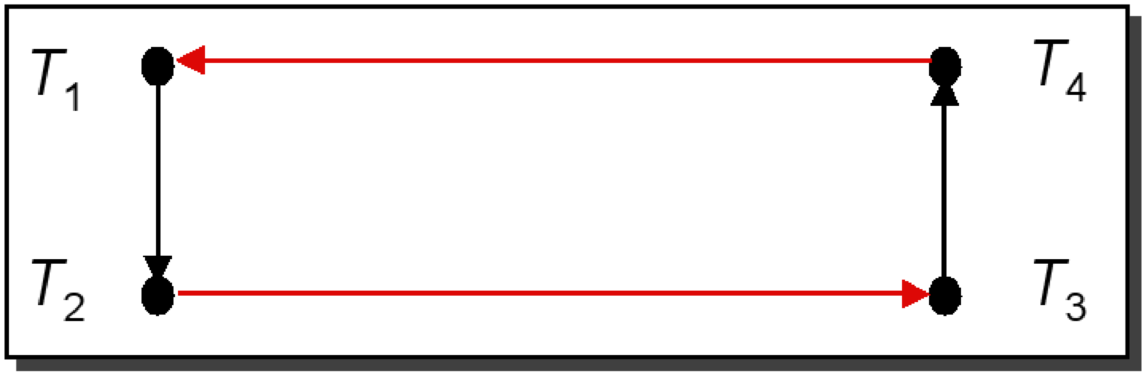


Deadlock Management . . .

- Deadlock management in a DDBMS is more complicate, since lock management is not centralized
- We might have **global deadlock**, which involves transactions running at different sites
- A Local Wait-for-Graph (LWFG) may not show the existence of global deadlocks
- A Global Wait-for Graph (GWFG), which is the union of all LWFGs, is needed

- **Example:** Assume T_1 and T_2 run at site 1, T_3 and T_4 run at site 2, and the following wait-for relationships between them: $T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_4 \rightarrow T_1$. This deadlock cannot be detected by the LWFGs, but by the GWFG which shows intersite waiting.
  - Local WFG:

  - Global WFG:



Deadlock Prevention

- **Deadlock prevention**: Guarantee that deadlocks never occur
  - Check transaction when it is initiated, and start it only if all required resources are available.
  - All resources which may be needed by a transaction must be predeclared
- Advantages
  - No transaction rollback or restart is involved
  - Requires no run-time support
- Disadvantages
  - Reduced concurrency due to pre-allocation
  - Evaluating whether an allocation is safe leads to added overhead
  - Difficult to determine in advance the required resources

Deadlock Avoidance

- **Deadlock avoidance:** Detect potential deadlocks in advance and take actions to ensure that a deadlock will not occur. Transactions are allowed to proceed unless a requested resource is unavailable
- Two different approaches:
  - **Ordering of data items**: Order data items and sites; locks can only be requested in that order (e.g., graph-based protocols)
  - **Prioritize transactions:** Resolve deadlocks by aborting transactions with higher or lower priority. The following schemes assume that T_i requests a lock hold by T_j:
    - **Wait-Die Scheme: if** $ts(T_i) < ts(T_j)$ **then** T_i waits **else** T_i dies
    - **Wound-Wait Scheme: if** $ts(T_i) < ts(T_j)$ **then** T_j wounds (aborts) **else** T_i waits
- Advantages
  - More attractive than prevention in a database environment
  - Transactions are not required to request resources a priori
- Disadvantages
  - Requires run time support

Deadlock Detection

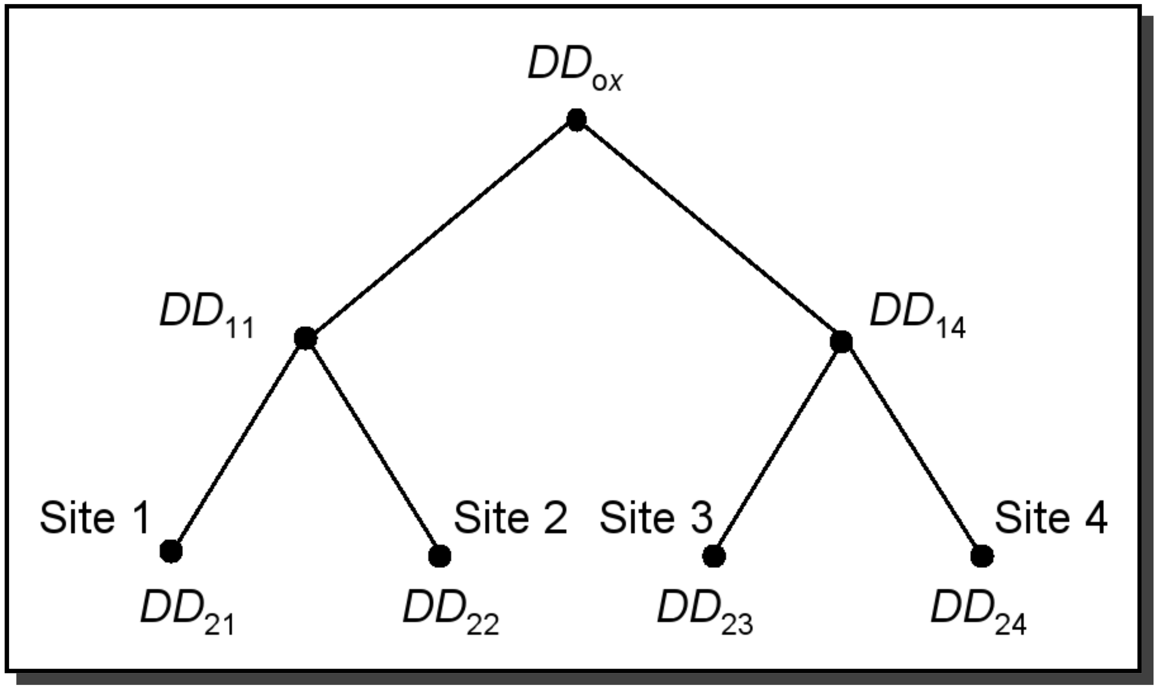
- **Deadlock detection and resolution:** Transactions are allowed to wait freely, and hence to form deadlocks. Check global wait-for graph for cycles. If a deadlock is found, it is resolved by aborting one of the involved transactions (also called the victim).
- Advantages
  - Allows maximal concurrency
  - The most popular and best-studied method
- Disadvantages
  - Considerable amount of work might be undone
- Topologies for deadlock detection algorithms
  - Centralized
  - Distributed
  - Hierarchical

Deadlock Detection . . .

- **Centralized deadlock detection**
  - One site is designated as the deadlock detector (DDC) for the system
  - Each scheduler periodically sends its LWFG to the central site
  - The site merges the LWFG to a GWFG and determines cycles
  - If one or more cycles exist, DDC breaks each cycle by selecting transactions to be rolled back and restarted
- This is a reasonable choice if the concurrency control algorithm is also centralized

Deadlock Detection . . .

- **Hierarchical deadlock detection**
  - Sites are organized into a hierarchy
  - Each site sends its LWFG to the site above it in the hierarchy for the detection of deadlocks
  - Reduces dependence on centralized detection site



Deadlock Detection . . .

- **Distributed deadlock detection**
  - Sites cooperate in deadlock detection
  - The local WFGs are formed at each site and passed on to the other sites.
  - Each local WFG is modified as follows:
    - Since each site receives the potential deadlock cycles from other sites, these edges are added to the local WFGs
    - i.e., the waiting edges of the local WFG are joined with waiting edges of the external WFGs
  - Each local deadlock detector looks for two things:
    - If there is a cycle that does not involve the external edge, there is a local deadlock which can be handled locally
    - If there is a cycle involving external edges, it indicates a (potential) global deadlock.

Conclusion

- Concurrency orders the operations of transactions such that two properties are achieved: (i) the database is always in a consistent state and (ii) the maximum concurrency of operations is achieved
- A schedule is some order of the operations of the given transactions. If a set of transactions is executed one after the other, we have a serial schedule.
- There are two main groups of serializable concurrency control algorithms: locking based and timestamp based
- A transaction is deadlocked if two or more transactions are waiting for each other. A Wait-for graph (WFG) is used to identify deadlocks
- Centralized, distributed, and hierarchical schemas can be used to identify deadlocks