Concurrency Control Techniques

Chapter 18

Chapter Objectives

- Discusses a number of concurrency control techniques that are used to insure the noninterference or isolation property (one of the **ACID** properties) of concurrently executing transactions.

	SID	Name	Major	YOB	GPA
A →	221234	Ali	ICS	1984	3.2
	221543	Ahmed	COE	1983	3.3
	221965	Emad	SE	1985	3.4
B →	222785	Fahd	SWE	1984	3.5
	223542	Lutfi	ICS	1984	3.6
	229851	Basam	COE	1985	3.7

- Chapter Outline

- Purpose of Concurrency Control
- Two-Phase Locking Based Concurrency Control
- Timestamp Based Concurrency Control
- Multiversion Concurrency Control Technique

- Purpose of Concurrency Control

- To enforce Isolation or noninterference among conflicting transactions.
  - To preserve database consistency through consistency preserving execution of transactions.
  - To resolve read-write and write-write conflicts

Example: In concurrent execution environment if T1 conflicts with T2 over a data item A, then the existing concurrency control decides if T1 or T2 should get the A and if the other transaction is rolled-back or waits.

... - Two-Phase Locking (2PL) ...

- A **lock** is a variable associated with a data item that describes the status of the item with respect to possible operations that can be applied to it.

- Locking is an operation which secures a permission to Read or a permission to Write a data item for a transaction.
  - Example: **Lock (X):** Data item X is locked in behalf of the requesting transaction

- Unlocking is an operation which removes these permissions from the data item.
  - Example: **Unlock (X):** Data item X is made available to all other transactions.

- Lock and Unlock are **Atomic** operations.

-- 2PL: Essential components ...

- Two locks modes:

Shared mode: shared lock (X). More than one transaction can apply share lock on X for reading its value but no write lock can be applied on X by any other transaction.

Exclusive mode: Write lock (X). Only one write lock on X can exist at any time and no shared lock can be applied by any other transaction on X.

Conflict matrix

Lock		
	Yes	No
	No	No

-- 2PL: Essential components ...

T_1 →

← T_3

T_2 →

	SID	Name	Major	YOB	GPA
	221234	Ali	ICS	1984	3.2
	221543	Ahmed	COE	1983	3.3
	221965	Emad	SE	1985	3.4
	222785	Fahd	SWE	1984	3.5
	223542	Lutfi	ICS	1984	3.6
	229851	Basam	COE	1985	3.7

... -- 2PL: Essential components ...

- **Lock Manager**: Managing locks on data items.
- **Lock table:** Lock manager uses it to store the identify of transaction locking a data item, the data item, lock mode and pointer to the next data item locked. One simple way to implement a lock table is through linked list

Transaction ID	Data item id	lock mode	Ptr to next data item
T1	X1	Read	Next

... -- 2PL: Essential components ...

- Database requires that all transactions should be well-formed. A transaction is well-formed if:
  - It must lock the data item before it reads or writes to it.
  - It must unlock the data item after it is done with it.
  - It must not lock an already locked data item.
  - It must not try to unlock a free data item.

... -- 2PL: Essential components ...

- The following code performs the **read-lock** operation:

```
B: if LOCK (X) = "unlocked" then
     begin LOCK (X) ← "read-locked";
       no_of_reads (X) ← 1;
     end
     else if LOCK (X) ← "read-locked" then
            no_of_reads (X) ← no_of_reads (X) +1
     else begin wait (until LOCK (X) = "unlocked" and
            the lock manager wakes up the transaction);
            go to B
     end;
```

... -- 2PL: Essential components ...

- The following code performs the **write-lock** operation:

```
B: if LOCK (X) = "unlocked" then
        begin LOCK (X) ← "write-locked";
    else begin
            wait (until LOCK (X) = "unlocked" and
            the lock manager wakes up the transaction);
            go to B
        end;
```

... -- 2PL: Essential components ...

- The following code performs the **unlock** operation:

```
if LOCK (X) = "write-locked" then
      begin LOCK (X) ← "unlocked";
          wakes up one of the transactions, if any
      end
      else if LOCK (X) ← "read-locked" then
         begin
              no_of_reads (X) ← no_of_reads (X) -1
              if  no_of_reads (X) = 0 then
              begin
                  LOCK (X) = "unlocked";
                 wake up one of the transactions, if any
              end
         end;
```

... -- 2PL: Essential components ...

- Lock conversion
  - Lock upgrade: existing read-lock to write-lock

if Ti has a read-lock (X) and Tj has no read-lock (X) (i ≠ j) then
convert read-lock (X) to write-lock (X)
else
force Ti to wait until Tj unlocks X

  - Lock downgrade: existing write-lock to read-lock

Ti has a write-lock (X) (*no transaction can have any lock on X*)
convert write-lock (X) to read-lock (X)

... -- 2PL: Essential components ...

- A transaction is said to follow 2PL protocol if all its locking operations precede its first unlock operation.
- 2PL algorithm
  - 2 Phases
    1. **Locking (Growing) Phase**: A transaction applies locks (read or write) on desired data items one at a time
    2. **Unlocking (Shrinking) Phase**: A transaction unlocks its locked data items one at a time.
  - **Requirement:** For a transaction these two phases must be mutually exclusively, that is, during locking phase unlocking phase must not start and during unlocking phase locking phase must not begin.

... -- 2PL: Essential components ...

T1	T2
read_lock (Y);	read_lock (X);
read_item (Y);	read_item (X);
unlock (Y);	unlock (X);
write_lock (X);	Write_lock (Y);
read_item (X);	read_item (Y);
X:=X+Y;	Y:=X+Y;
write_item (X);	write_item (Y);
unlock (X);	unlock (Y);

T1 and T2 are **NOT** following 2PL protocol

... -- 2PL: Essential components ...

T3	T4
read_lock (Y); read_item (Y); write_lock (X); unlock (Y); read_item (X); X:=X+Y; write_item (X); unlock (X);	read_lock (X); read_item (X); Write_lock (Y); unlock (X); read_item (Y); Y:=X+Y; write_item (Y); unlock (Y);

T3 and T4 are following 2PL protocol

-- 2PL Algorithms

- Two-phase policy generates two locking algorithms:

1. **Conservative**: Prevents deadlock by locking all desired data items before transaction begins execution.

2. **Basic**: Transaction locks data items incrementally. This may cause deadlock which is dealt with

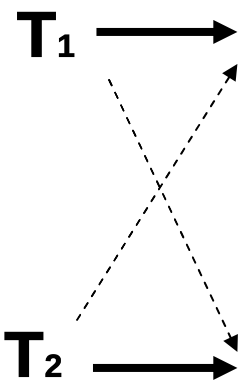
    - **Strict**: A more stricter version of Basic algorithm where unlocking is performed after a transaction terminates (commits or aborts and rolled-back). This is the most commonly used two-phase locking algorithm

-- Dealing with Deadlock and Starvation ...

T1	T2
read_lock (Y); read_item (Y);	
	read_lock (X); read_item (Y);
write_lock (X); (waits for X)	
	write_lock (Y); (waits for Y)

- T1 and T2 did follow two-phase policy but they are deadlock

... -- Dealing with Deadlock and Starvation ...



	SID	Name	Major	YOB	GPA
	221234	Ali	ICS	1984	3.2
	221543	Ahmed	COE	1983	3.3
	221965	Emad	SE	1985	3.4
	222785	Fahd	SWE	1984	3.5
	223542	Lutfi	ICS	1984	3.6
	229851	Basam	COE	1985	3.7

→ : Holds ⇢ : Requests

... -- Dealing with Deadlock and Starvation ...

- Three techniques to solve deadlock problems
  - **Deadlock prevention**
    - A transaction locks all data items it refers to before it begins execution
  - **Deadlock detection and resolution**
    - A wait-for-graph is created using the lock table. As soon as a transaction is blocked, it is added to the graph. When a chain like: Ti waits for Tj waits for Tk waits for Ti or Tj occurs, then this creates a cycle. One of the transaction of the cycle is selected and rolled back
  - **Deadlock avoidance**
    - As soon as the algorithm discovers that blocking a transaction is likely to create a cycle, it rolls back the transaction

... -- Dealing with Deadlock and Starvation ...

- Starvation
  - Starvation occurs when a particular transaction consistently waits or restarted and never gets a chance to proceed further. In a deadlock resolution it is possible that the same transaction may consistently be selected as victim and rolled-back. This limitation is inherent in all priority based scheduling mechanisms. In Wound-Wait scheme a younger transaction may always be wounded (aborted) by a long running older transaction which may create starvation.

- Timestamp based concurrency control algorithm ...

- A **timestamp** is a unique identifier created by a DBMS to identify a transaction.
- A timestamp is a monotonically increasing variable (integer) indicating the age a transaction. A larger timestamp value indicates a younger transaction.
- Timestamp based algorithm uses timestamp to serialize the execution of concurrent transactions.

…- Timestamp based concurrency control algorithm …

- In order to use timestamp values for serializable scheduling of transactions, the transaction manager of a DBMS associates with each database item X two timestamp (TS) values:
  - **Read_TS(X):** The timestamp (identifier) of the youngest transaction that has read X successfully.
  - **Write_TS(X):** The timestamp (identifier) of the youngest transaction that has written X successfully.

… - Timestamp based concurrency control algorithm …

- Basic Timestamp Ordering

1. Transaction T issues a **write_item(X)** operation:
   a) If read_TS(X) > TS(T) or if write_TS(X) > TS(T), then an younger transaction has already read the data item so abort and roll-back T and reject the operation
   b) If the condition in part (a) does not exist, then execute write_item(X) of T and set write_TS(X) to TS(T).

2. Transaction T issues a **read_item(X)** operation:
   a) If write_TS(X) > TS(T), then an younger transaction has already written to the data item so abort and roll-back T and reject the operation.
   b) If write_TS(X) ≤ TS(T), then execute read_item(X) of T and set read_TS(X) to the larger of TS(T) and the current read_TS(X).

… - Timestamp based concurrency control algorithm …

- Strict Timestamp Ordering (for ease of recoverability)

1. Transaction T issues a write_item(X) operation:
   - If TS(T) > read_TS(X), then delay T until the transaction T' that wrote or read X has terminated (committed or aborted).
2. Transaction T issues a read_item(X) operation:
   - If TS(T) > write_TS(X), then delay T until the transaction T' that wrote or read X has terminated (committed or aborted).

- Multiversion concurrency control technique Concept ...

- This approach maintains a number of versions of a data item and allocates the right version to a read operation of a transaction. Thus unlike other mechanisms a read operation in this mechanism is never rejected.

- **Side effect**: Significantly more storage (RAM and disk) is required to maintain multiple versions. To check unlimited growth of versions, a garbage collection is run when some criteria is satisfied

- Multiversion concurrency control technique Concept ...

- Assume X1, X2, ..., Xn are the version of a data item X created by a write operation of transactions. With each Xi a read_TS (read timestamp) and a write_TS (write timestamp) are associated.
- **read_TS(Xi)**: The read timestamp of Xi is the largest of all the timestamps of transactions that have successfully read version Xi
- **write_TS(Xi)**: The write timestamp of Xi that wrote the value of version Xi.
- A new version of Xi is created only by a write operation.

- Multiversion concurrency control technique Concept ...

- To ensure serializability, the following two rules are used.

  1. If transaction T issues write_item (X) and version i of X has the highest write_TS(Xi) of all versions of X that is also less than or equal to TS(T), and read _TS(Xi) > TS(T), then abort and roll-back T; otherwise create a new version Xi and read_TS(X) = write_TS(Xj) = TS(T).

  2. If transaction T issues read_item (X), find the version i of X that has the highest write_TS(Xi) of all versions of X that is also less than or equal to TS(T), then return the value of Xi to T, and set the value of read _TS(Xi) to the largest of TS(T) and the current read_TS(Xi).

- Rule 2 guarantees that a read will never be rejected.

END