Database Concurrency Control

SWEN304/SWEN435

Lecturer: Dr Hui Ma

Engineering and Computer Science

TE WHARE WĀNANGA O TE ŪPOKO O TE IKA A MĀUI
VICTORIA
UNIVERSITY OF WELLINGTON

Slides by: Pavle Morgan & Hui Ma

Outline

- Transaction schedules
- Basic locks and basic locking rules
- Lock conversion
- Lost update and locking
- Protocols to insure isolation property of concurrent transactions
- Dead lock and dead lock prevention protocols
- Starvation
- Phantom record
    - *Readings from the textbook:*
        - *Chapter 21: Section 21.5,*
        - *Chapter 22 : Sections 22.1, 22.2, and 22.5*
    - *PostgreSQL Manulas*

Database Concurrency Control

- **Purpose of Concurrency Control**
  - To enforce Isolation (through mutual exclusion) 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 T_1 conflicts with T_2 over a data item *A*, then the existing concurrency control decides if T_1 or T_2 should get *A* and if the other transaction is rolled-back or waits

Transaction Schedules

- **Transaction schedule or history:** When transactions are executing concurrently in an interleaved fashion, the order of execution of operations from the various transactions forms what is known as **a transaction schedule (or history)**

- A **schedule** (or **history**) S of n transactions $T_1, T_2, \ldots, T_n$ is an ordering of the operations of the transactions subject to the constraint that, for each transaction T_i that participates in S, the operations of T_i in S must appear in the same order in which they occur in T_i

- Note, however, that operations from other transactions T_j can be interleaved with the operations of T_i in S

Transaction Schedules based on Serializability

- **Serial schedule**: A schedule *S* is **serial** if, for every transaction *T* participating in the schedule, all the operations of *T* are executed consecutively in the schedule. Otherwise, the schedule is called **nonserial schedule**
- **Serializable schedule**: A schedule *S* is **serializable** if it is equivalent to some **serial schedule** of the same *n* transactions
- **Result equivalent**: Two schedules are called *result equivalent* if they produce the same final state of the database

Schedules based on Serializability (3)

- Being **serializable** is not the same as being **serial**
- Being serializable implies that the schedule is a correct schedule
  - It will leave the database in a consistent state
  - The interleaving is appropriate and will result in a state as if the transactions were **serially** executed
    - will achieve efficiency due to concurrent execution

Transaction Schedules based on Serializability

Practical approach:

- Come up with methods (protocols) to ensure serializability
- It is not possible to determine when a schedule begins and when it ends
- Hence, we reduce the problem of checking the whole schedule, to checking only a *committed project* of the schedule (i.e. operations from only the committed transactions.)
- Current approach used in most DBMSs:
  - Use of locks with two-phase locking

Locking

- Locking is the most frequent technique used to control concurrent execution of database transactions
- Operating systems provide a binary locking system (*lock* and *unlock*) that is too restrictive for database transactions
- That is why DBMS contains its own lock manager
- A lock_value(*X*) is variable associated with (each) database data item *X*
- The lock_value(*X*) describes the status of the data item *X*, by telling which operations can be applied to *X*

Kinds of Locks

- Generally, the lock manager of a DBMS offers two kinds of locks:
  - shared (read) lock
  - exclusive (write) lock
- If a transaction *T* issues a read_lock(*X*) command, it will be added to the list of transactions that share lock on item *X*, unless there is a transaction already holding write lock on *X*
- If a transaction *T* issues a write_lock(*X*) command, it will be granted an exclusive lock on *X*, unless another transaction is already holding lock on *X*
- Accordingly,

  $lock\_value \in \{read\_lock, write\_lock, unlocked\}$

Basic Locking Rules

- The basic locking rules are:
  - *T* must issue a **read_lock(*X*)** or **write_ lock(*X*)** command before any **read_item(*X*)** operation
  - *T* must issue a **write_lock(*X*)** command before any **write_item(*X*)** operation
  - *T* must issue an **unlock(*X*)** command when all **read_item(*X*)** or **write_item(*X*)** operations are completed
- Some DBMS lock managers perform automatic locking by granting an appropriate database item lock to a transaction when it attempts to read or write an item into database
- So, an item lock request can be either explicit, or implicit

Lock Conversion

- A transaction *T* that already holds a lock on item *X* can convert it to another state:
    - *T* can **upgrade** a **read_lock(*X*)** to a **write_lock(*X*)** if it is the only one that holds a lock on the item *X* (otherwise, *T* has to **wait**)
    - *T* can always **downgrade** a **write_lock(*X*)** to a **read_lock(*X*)**

Lost Update Problem and Locking

	T_1	T_2	time
	read_lock(*X*)		
1000	**read_item (*X*)**		
	unlock(*X*)		
		write_lock(*X*)	
		read_item *X*	1000
		$X = X + M$	M= 2000
		write_item(*X*)	3000
		unlock(*X*)	
	write_lock(*X*)		
N= 200	**$X = X – N$**		
800	**write_item (*X*)**		
	unlock(*X*)		

- The problem is that T_1 releases lock on *X* too early, allowing T_2 to start updating *X*
- We need a protocol that will guarantee database consistency

2-Phase Locking Techniques: The algorithm

- **Two Phases**
  - **(a) Locking (Growing) Phase**: A transaction applies locks (read or write) on desired data items one at a time.
  - **(b) 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.

Strict 2-Phase Locking

- Protocol:
  - All lock operations of a transaction *T* must precede the first unlock operation
  - A transaction *T* does not release any of exclusive locks until after it commits or aborts
- Comments:
  - No other transaction can read or write an item *X* that is written by *T* unless *T* has committed
  - The strict 2-phase locking protocol is safe for all transaction anomalies mentioned so far
  - It is also called **read committed** protocol, because transactions are allowed to read only committed database items

Undesirable Effects of Locking

- 2-phase locking can introduce some undesirable effects:
  - waits,
  - deadlocks,
  - Starvation

- Waits relate to the fact that a transaction wanting to acquire a lock on a database item *X* has to wait if another transaction has already acquired an exclusive lock on *X*

Deadlock

- Deadlock is also called deadly embrace
- Typical sequence of operations is given in the following diagram

T_1	T_2
write_lock(*X*)	
	write_lock(*Y*)
write_lock(*Y*) **//has to wait**	
	write_lock(*X*) **//has to wait**

time ↓

- T_1 acquired exclusive lock on X
- T_2 acquired exclusive lock on Y
- No one can finish, because both are in the waiting state

Deadlock

- Deadlock occurs when:
  - Each transaction T_i in a set of *two or more transactions* $T = \{T_1, T_2, \ldots, T_n\}$ is waiting for some item X that is locked by some other transaction T_j
- In other words:
  - A number of transactions (greater than one) hold lock on one item and wait to acquire another
  - None of the waiting transactions can acquire locks on all necessary items

Deadlock Examples

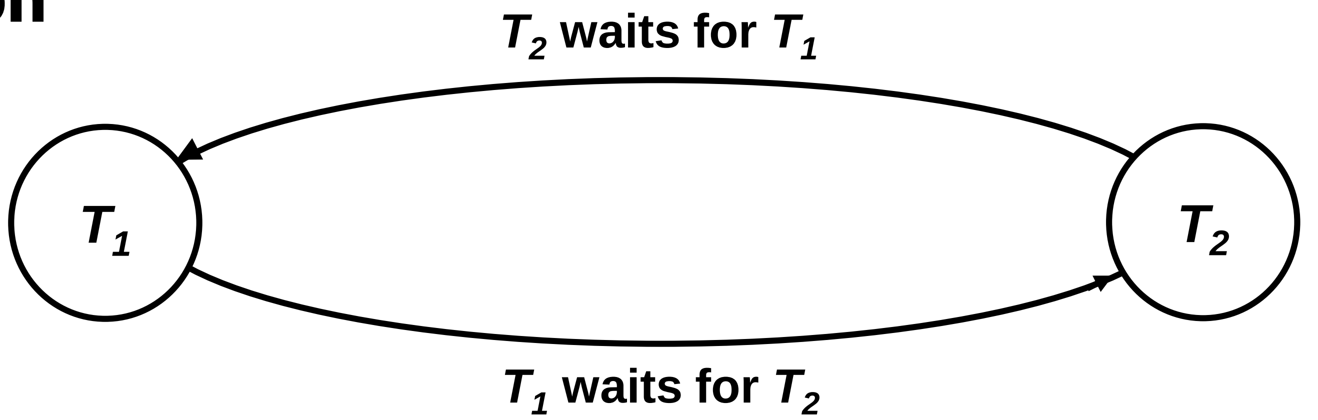
a)

- T_1 has locked X and waits to lock Y
- T_2 has locked Y and waits to lock Z
- T_3 has locked Z and waits to lock X

b)

- Both T_1 and T_2 have acquired sharable locks on X and wait to lock X exclusively

- A dead-lock may be represented using a cyclic **wait-for graph**



Deadlock Prevention Techniques (1)

- We distinguish between deadlock **prevention** and deadlock **detection** techniques
- Deadlock prevention techniques:
  - **Conservative 2-phase lock** protocol: lock all items in advance, if any of them cannot be obtained, none of the item are locked; try again later
  - *Timestamp techniques*:
    - **Wait–Die** protocol: if $TS(T_i) < TS(T_j)$ (T_i is older than T_j) then T_i is allowed to wait. Otherwise (T_i is younger than T_j) abort T_i (dies) and restart it later with the same timestamp
    - **Wound–Wait** protocol: if $TS(T_i) < TS(T_j)$, (T_i is older than T_j) then abort T_j (T_i wounds T_j) and restart it later with the same timestamp. Otherwise (T_i is younger than T_j) T_i is allowed to wait

Deadlock Prevention Techniques (2)

- **No Waiting (NW)** protocol: if unable to get a lock, immediately abort and restart again after a certain time

- **Cautious Waiting (CW)** protocol: if T_j is not blocked, then T_i is blocked and allowed to wait; otherwise abort T_i

Conservative 2-Phase Locking Protocol

- **Conservative 2-Phase Locking Protocol:**
  - A transaction has to lock all items it will access before it begins to execute
  - If it cannot acquire any of its locks, it releases all items, aborts, and tries again,
- Comments:
  - Deadlock can't occur because no hold-and-wait
  - Once it starts, a transaction can only release its locks
- Problems:
  - What if a transaction cannot predetermine all items it is going to use? (e.g. a sequence of interactive SQL statements comprising one database transaction)
  - What if a database item that is already locked by another transaction will be released very soon? (i.e. the transaction is aborted in vane)

Deadlock Detection Schemes

- Deadlock prevention is justified if transactions are long and use many items, or transaction load is very heavy
- In many practical situations it is advantages not to do deadlock prevention but to detect dead locks and then abort at least one of the transactions involved
- Deadlock detection schemes are:
  - Deadlock detection using wait-for graph
  - Timeouts protocol

Deadlock Detection Protocols

- Deadlock detection using a **wait-for graph**:
  - Construct a wait-for graph where each transaction has its node
  - If T_i waits on T_j, construct a directed edge from T_i to T_j
  - If there is a cycle detected, select a `victim' and abort it
  - Victim selecting algorithm should select and abort transactions that made the least number of updates
- **Timeouts** protocol:
  - If a transaction waits longer than a specified amount of time, it gets aborted
  - Here, deadlock is only supposed, not proved

Starvation

- **Starvation** occurs when a transaction can not make any progress for an indefinite period of time, while other transactions proceed
  - can occur when waiting protocol for locked items is unfair (used stacks instead of queues)
  - 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
- Wound-Wait and Wait-Die schemes can avoid starvation, because the aborted transactions restart with the same original timestamp

Granularity of Items

- Until now, we used the term `data item' without specifying its exact meaning
- In the context of the concurrency control, a data item can be:
  - A field of a database record,
  - A database record,
  - A disk block,
  - A whole file,
  - A whole database
- The coarser data item granularity is, the more contention between transactions will occur, and less productive the DBMS will be (more waits or aborts)

Granularity of Items (continued)

- The finer data granularity, the higher locking overhead of the DBMS lock manager (due to many locks and unlocks)
- The best item size depends on the type of a transaction:
  - If a transaction accesses a small number of records, than

    data item = record

  - If a transaction accesses a large number of records in the same file, then

    data item = file

  - Some DBMS automatically change granularity level with regard to the number of records a transaction is accessing (attempting to lock)

Phantom Record

- A transaction locks database items that satisfy certain selection condition and updates them
- During that update, another transaction inserts a new item that satisfies the same selection condition
- After the update, but inside the same transaction, we suddenly discover the existence of a database item that has not been updated although it should have been (since it satisfies the selection condition)
- This database item, called a "phantom record", appeared because it did not exist when locking has been done

Summary

- Basic locks:
  - Shareable,
  - Exclusive
- To avoid all update anomalies:
  - Lost Update,
  - Unrepeatable Read, and
  - Dirty Read

  locks should be released only just after the COMMIT point
- Two phase locking protocol may introduce:
  - Waits,
  - Deadlocks,
  - Starvation

Summary (continued)

- There are many deadlock prevention schemes, but no one is ideal
- In the context of the concurrency control, a database item can be:
  - a field of a database record (tuple),
  - a database record,
  - a disk block,
  - a whole table
  - a whole file,
  - a whole database
- Each data item granularity has advantages and disadvantages, but database record granularity is desirable
- Phantom record may appear if a finer granularity than a table is used