Unit 10.5
Transaction Processing: Concurrency
Concurrency in Context

User Level (View Level)
- Derived Tables
- Constraints, Privileges
  - Derived
  - Queries (DML)

Community Level (Base Level)
- Base Tables
- Constraints, Privileges
  - Implemented
  - Application Data Analysis (ER)
  - Normalization (NFs)
  - Schema Specification (DDL)
  - Queries (DML)

Physical Level
- Files
- Indexes, Distribution
  - Query Execution (B^+, ..., Execution Plan)

DBMS OS Level
- Concurrency
- Recovery
  - Transaction Processing (ACID, Sharding)

Centralized Or Distributed
- Standard OS
- Standard Hardware
Transactions

- Transaction is an execution of a user’s program
- In the *cleanest and most important* model a transaction is supposed to satisfy the **ACID** conditions
  - Atomic
  - Consistent
  - Isolated
  - Durable

- Some transactions may not satisfy all of these conditions in order to improve performance, as we will see later
Recovery and Concurrency Management

◆ The job of these recovery/concurrency modules of the database operating system is to assure the ACID properties, and handle other related issues

◆ Recovery does ACD, but can use help from Concurrency, though strictly Recovery is needed even if there is no Concurrency

◆ Concurrency does I while possibly (and in our description, definitely) supporting ACD
The Concurrency Problem

- Here we focus on *Isolation* in the presence of concurrently executing transactions
- Each transaction should run as if there were no other transactions in the system
- Our execution platform: a single centralized system, with concurrent execution of transactions
- Distributed databases more difficult to handle, as we will see briefly later in the class
Optimistic Concurrency Control

- Locking frequently can hurt performance
  - Overhead of acquiring locks
  - Blocking for reads

- It may not be necessary if there are few conflicts
- With *optimistic concurrency control*, the transaction uses data without acquiring locks
- Before committing, each transaction verifies that no other transaction modified data that it has read
- If there was a conflict, abort the transaction
Optimistic Concurrency Control

- Use a compare operation CMP to check if a value has changed
- System often stores a version number with the data to facilitate compare operations. We discuss later.

T1: READ x
T2: WRITE x
T1: CMP x
T2: COMMIT

abort
Serial Execution

◆ With *serial execution*, it appears as if transactions follow each other: no concurrency

◆ Example of a serial execution

T1: READ x  
T1: WRITE x  
T1: READ y  
T1: WRITE y

T2: READ x  
T2: WRITE x  
T2: READ y  
T2: WRITE y
Snapshot Isolation

- With *snapshot isolation*, each transaction has its own copy of the state (i.e., has its own snapshot)
- Before committing, compare write sets. A write-write conflicts may cause transactions to abort
- Resolve conflicts with “first committer wins” rule

T1: READ x  
T1: WRITE x  

T2: READ x  
T2: WRITE x  
T2: READ y  
T2: WRITE y

T1: READ y  
T1: WRITE y  
T1: CMP(x)  
T1: CMP(y)  

abort
Multiversion Concurrency Control

- Snapshot isolation is usually implemented with multiversion concurrency control
- When items are written to the database, the old value is not over-written
- The DBMS assigns each item a version number
- A transaction may not read the latest version. It reads the version from the time the transaction started
- The transaction sees a “snapshot” of the database from that time
**Snapshot Isolation Benefits**

- Performance similar to read committed, since read operations do not block
- No dirty reads (because of multiple versions)
- No lost reads
- No inconsistent reads
- No phantoms
- But, it *may have write skew*
**Example**

- A database consisting of two items: \( x, y \)
- Initially \( x = 100 \) and \( y = 100 \)
- The only criterion for correctness is the single integrity constraint:

\[
x + y \geq 0
\]

- Consider two simple transactions, T1 and T2
  - T1: \( x := x - 200 \)
  - T2: \( y := y - 200 \)

- Both transactions in isolation are correct: they preserve the consistency of the database
An Execution History

- An execution history

  T1

  \[ x := x - 200 \]

  \[ \text{check} \ (x + y \geq 0) \]

  T2

  \[ y := y - 200 \]

  \[ \text{check} \ (x + y \geq 0) \]

- In a **serial** execution, T2 should abort
- With **snapshot isolation**, T1 and T2 commit
- The is called **write skew**
Advanced Material
Locking Is Prone To Starvation

This can continue indefinitely: T7, ...

Unless something is done, T2 will never gets the lock it wants

Obvious solution, stop granting S-locks and when the last S-lock is released, give X-lock to T2
Two-Phase Locking Is Prone To Deadlocks

瘤 Two transactions
瘤 T1: x := x + 1; y := y + 1
瘤 T2: y := 2y; x := 2x

1. T1 T2
2. X x
3. R x
4. W x
5. X y
6. R y
7. W y
8. X x (waits)
9. X y (waits)
瘤 We got a deadlock
瘤 *In fact this deadlock prevented a non-serializable history*
瘤 “Deadlocks are not a bug, but a feature”
Detecting And Avoiding Deadlocks

◆ Deadlocks are characterized by a cyclic “wait for” graph
◆ Ours was very simple, T1 waited for T2 and T2 waited for T1
◆ To detect if there is a deadlock, draw a “wait for” graph
  • Nodes: Transactions
  • Arc from T1 to T2 iff T1 waits for T2
◆ If there are cycles, some transaction need to be aborted
◆ There are protocols that avoid deadlock by aggressive abortion of transactions, sometimes not necessary
◆ They abort enough transactions, so that no cycles could ever appear in the “wait for” graph, so not need to draw it during execution
  • But they may abort transactions unnecessarily
Kill-Wait Protocol: Locking + More

◆ Each transaction, when entering the system is timestamped with the current time: timestamp of T is denoted by TS(T)

◆ If transaction Ti wants to lock x, which another transaction Tj holds in a conflicting mode (at least one of the two locks is an X-lock),
  - If TS(T1) < TS(T2), then abort T2 and give the lock to T1 (the older transaction kills the younger transaction)
  - If TS(T1) > TS(T2), then TS(T1) waits

◆ If a transaction unlocks a lock, the oldest from among the waiting transactions (all younger than the unlocking transaction) gets it

◆ In the “wait for” graph all the arcs are from a younger transaction to an older transaction, and therefore there cannot be a cycle
Wait-Die Protocol: Locking + More

◆ Each transaction, when entering the system is timestamped with the current time: timestamp of T is denoted by TS(T)

◆ If transaction Ti wants to lock x, which another transaction Tj holds in a conflicting mode (at least one of the two locks is an X-lock),
  - If TS(T1) < TS(T2), then T1 waits
  - If TS(T1) > TS(T2), TS(T1) abort T1 (T1 dies)

◆ If a transaction unlocks a lock, the youngest from among the waiting transactions (all older than the unlocking transaction) gets it

◆ In the “wait for” graph all the arcs are from an older transaction to a younger transaction, and therefore there cannot be a cycle
Timestamp-Based Protocol For Concurrency

◆ Each transaction is issued a timestamp when accepted by DB OS
◆ The first transaction gets the timestamp 1
◆ Every subsequent transaction gets the timestamp that is the previously largest assigned timestamp + 1
◆ So will can refer to transactions as “older” and “younger” based on their timestamps and also use timestamp value for transaction identification
◆ The system maintains for each item x two timestamps:
  • $RT(x)$ is the youngest transaction (largest timestamp) that read it
  • $WT(x)$ is the youngest transaction (largest timestamp) that wrote it
Timestamp-Based Protocol

◆ Assume that T1, T2, … arrive in this order and that the time stamp of Ti is i

◆ For simplicity assume that the database was created by transaction T0

◆ **We want to get schedules equivalent to the serial order T0, T1, T2, …, or some subsequence of this**, as some transactions can abort, so will not appear in the schedule

◆ **If we do this, our schedule will be serializable**

◆ Similarly to what we did during topological sort, we could say that Ti executed instantaneously at virtual time Ti
Equivalent Serial Schedule
**Scenario**

\[
\text{RT}(x) = 5 \quad \text{WT}(x) = 8
\]

**Virtual Time**

1  unknown
2  who
3  read
4
5  T5: Read x
6
7  nobody
8  T8: Write x
9  read
   | nobody
   | wrote
Scenario

RT(x) = 8
WT(x) = 5

Virtual Time

1 | unknown
2 | who
3 | read
4 | unknown
5 | who
5 | T5: Write x
6 | nobody
7 | wrote
8 | nobody
8 | T8: Read x
9 | read
Timestamp-Based Protocol: Reading

◆ RT(x) = 5; this is the youngest transaction that read it
◆ WT(x) = 8; this is the youngest transaction that wrote it
◆ If a transaction Ti with a timestamp of i ≤ 7, say T6, wants to read x
  • The value it wanted no longer exists (it had to be written by T0 (i.e., initial state of the DB), or by Ti with i ≤ 6)
  • T6 cannot read x and has to be aborted
◆ If a transaction with a timestamp of i ≥ 8, say T9, wants to read it,
  • T9 reads x, and RT(x) := 9
**Timestamp-Based Protocol: Reading**

- RT(x) = 8; this is the youngest transaction that read it
- WT(x) = 5; this is the youngest transaction that wrote it
- If a transaction Ti with a timestamp of i ≤ 4, say T3, wants to read x
  - T3 cannot read x and has to be aborted (as too new a value of x exists)
- If a transaction Ti with a timestamp i, 5 ≤ i ≤ 8, say T6, wants to read x
  - T6 reads x
- If a transaction Ti with a timestamp of i ≥ 9, say T9, wants to read it
  - T9 reads x and RT(x) := 9
Timestamp-Based Protocol: Writing

- $RT(x) = 5$; this is the youngest transaction that read it
- $WT(x) = 8$; this is the youngest transaction that wrote it
- If a transaction $Ti$ with a timestamp of $i < 5$ wants to write, say $T3$,
  - $T3$ has to be aborted
  - Because there was a read of a value of $x$ by transaction $T5$, and maybe this was a value produced actually by $T2$. If we allow $T3$ to write, this would have meant that $T5$ read a value that was produced by a transaction that was too old
- If a transaction $Ti$ with a timestamp of $i > 8$ wants to write
  - $T9$ writes and $WT(x) = 9$
- If a transaction $Ti$ with a timestamp of $i \in [6, 7]$, say $T7$, wants to write
  - We just throw out the write and let $T7$ proceed
  - This was a blind write, nobody read it and it is obsolete (and nobody will be allowed to read it as described above; we will not go back to re-examine this case and check this out)
**Timestamp-Based Protocol: Writing**

- $RT(x) = 8$; this is the youngest transaction that read it
- $WT(x) = 5$; this is the youngest transaction that wrote it
- If a transaction $Ti$ with a timestamp of $i < 8$, say $T6$, wants to write
  - $T6$ has to be aborted
  - Because there was a read of a value of $x$ by transaction $T8$, and if we allow $T6$ to write $x$, this would mean that $T8$ read a value that was too old
- If a transaction $Ti$ with a timestamp of $i > 8$, say $T9$, wants to write
  - $T9$ writes and $WT(x) = 9$
Conflict Serializability And Deadlock Freedom

In the conflict graph all the arcs will be from an older transaction to a younger transaction.

Therefore the history will be conflict-serializable.

And as transactions never wait, there will be no deadlocks.

But the history may not even be recoverable.

We can make it strict, or even rigorous, by having transactions wait until the relevant transactions commit.

There still will not be any deadlocks, because younger transactions wait for older transactions to commit, but not the other way around.
Granularity Of Locks

◆ The problem of phantoms can be avoided, by say, locking the file that has all the accounts, and therefore no account can be added during the processing.

◆ Sometimes we may want to lock all the accounts (logically, so no new accounts can be added).

◆ Sometimes we may want to lock an account, to add money to it, for instance.
  - And of course, it is not efficient to lock all the accounts in order to modify one account only.

◆ So “lockable” objects are no longer disjoint items.

◆ This can be handled using somewhat more complex types of locks (called intention locks).

◆ You can read about Intention Locks and the protocol using them at http://en.wikipedia.org/wiki/Multiple_granularity_locking.
Granularity Of Locks
(Simplified Granularity-Based Locking)

◆ See [https://docs.oracle.com/cd/E17952_01/refman-5.1-en/innodb-lock-modes.html](https://docs.oracle.com/cd/E17952_01/refman-5.1-en/innodb-lock-modes.html)

◆ There are 4 types of locks and their compatibility matrix is

<table>
<thead>
<tr>
<th>Type</th>
<th>N</th>
<th>S</th>
<th>X</th>
<th>IS</th>
<th>IX</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>S</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>X</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>IS</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>IX</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

◆ IS stands for “Intent Shared”
◆ IX stands for “Intent eXclusive”

◆ As usual, the matrix states what two transactions can hold simultaneously, a transaction can hold any set of locks on an item (as long as not conflicting with other transactions)
Granularity Of Locks
(Simplified Granularity-Based Locking)

◆ Assume one table R and five rows 1, 2, 3, 4, 5.
◆ The items that can be locked are R, 1, 2, 3, 4, 5.
◆ To have write access to all of R, a transaction needs an X-lock on R
◆ To have a read access on all of R, a transaction needs an X-lock or an S-lock on R
◆ To have write access to a row of R, a transaction needs an X-lock on that row
◆ To have a read access on a row of R, a transaction needs an X-lock or an S-lock on that row
◆ To set an X-lock on a row of R, a transaction needs an IX-lock on R
◆ To set an S-lock on a row of R, a transaction needs an IX-lock or an IS-lock on R
◆ Locking is top to bottom; unlocking is bottom to top
Granularity Of Locks
(Simplified Granularity-Based Locking)

◆ The goal is to prevent situations such as
  ● T1 holds X-lock on R and T2 holds an S-lock on 1
  ● T1 holds X-lock on R and T2 holds an X-lock on 1
  ● T1 holds S-lock on R and T2 holds an X-lock on 1
◆ The goal is to permit situations such as
  ● T1 holds S-lock on R and T2 holds an S-lock on 1
  ● T1 holds S-lock on 1 and T2 holds an X-lock on 2
  ● T1 holds X-lock on 1 and T2 holds an X-lock on 2