# Transactions: Concurrency Control

Amol Deshpande
CMSC424

# Spring 2020 – Online Instruction Plan

- Week 1: File Organization and Indexes
- Week 2: Query Processing
- Week 3: Query Optimization; Parallel Databases 1
- Week 4: Parallel Databases; Mapreduce; Transactions 1
- Week 5: Transactions 2 (Homework Due May 1)
  - Transactions: Serializability, Recoverability
  - Transactions: Concurrency 1
  - Transactions: Concurrency 2: Other Concurrency Schemes
  - ~~Transactions: Recovery~~ (MOVED TO NEXT WEEK)
- Week 6: Transactions: Recovery; Distributed Transactions; Miscellaneous Topics (Homework Due May 8)

# Transactions: Concurrency 2

- Book Chapters
  - 15.4, 15.5, 15.7, 15.9
- Key topics:
  - Timestamp-based concurrency schemes
  - Optimistic (validation-based) concurrency control
  - Snapshot isolation
  - Phantom Problem
  - Weak levels of consistency in SQL

# Other CC Schemes: Time-stamp Based

- Time-stamp based
  - Transactions are issued time-stamps when they enter the system
  - The time-stamps determine the *serializability* order
  - So if T1 entered before T2, then T1 should be before T2 in the serializability order
  - Say *timestamp(T1) < timestamp(T2)*
  - If T1 wants to read data item A
    - If any transaction with larger time-stamp wrote that data item, then this operation is not permitted, and T1 is *aborted*
  - If T1 wants to write data item A
    - If a transaction with larger time-stamp already read that data item or written it, then the write is *rejected* and T1 is aborted
  - Aborted transaction are restarted with a new timestamp
    - Possibility of *starvation*

# Other CC Schemes: Time-stamp Based

- Example

| $T_1$ | $T_2$ | $T_3$ | $T_4$ | $T_5$ |
|---|---|---|---|---|
| | | | | read ($X$) |
| | read ($Y$) | | | |
| read ($Y$) | | | | |
| | | write ($Y$) | | |
| | | write ($Z$) | | |
| | | | | read ($Z$) |
| | read ($Z$) | | | |
| | abort | | | |
| read ($X$) | | | | |
| | | | read ($W$) | |
| | | write ($W$) | | |
| | | abort | | |
| | | | | write ($Y$) |
| | | | | write ($Z$) |

# Other CC Schemes: Time-stamp Based

- Time-stamp based
  - As discussed here, has too many problems
    - Starvation
    - Non-recoverable
    - Cascading rollbacks required
  - Most can be solved fairly easily
    - Read up
  - Remember: We can always put more and more restrictions on what the transactions can do to ensure these things
    - The goal is to find the minimal set of restrictions to as to not hinder concurrency

# Other Schemes: Optimistic Concurrency Control

- Optimistic concurrency control
  - Also called validation-based
  - Intuition
    - Let the transactions execute as they wish
    - At the very end when they are about to commit, check if there might be any problems/conflicts etc
      - If no, let it commit
      - If yes, abort and restart
  - Optimistic: The hope is that there won’t be too many problems/aborts

# Other Schemes: Optimistic Concurrency Control

- Each transaction $T_i$ has 3 timestamps
  - Start($T_i$) : the time when $T_i$ started its execution
  - Validation($T_i$): the time when $T_i$ entered its validation phase
  - Finish($T_i$) : the time when $T_i$ finished its write phase
- Serializability order is determined by timestamp given at validation time, to increase concurrency.
  - Thus TS($T_i$) is given the value of Validation($T_i$).
- This protocol is useful and gives greater degree of concurrency if probability of conflicts is low.
  - because the serializability order is not pre-decided, and
  - relatively few transactions will have to be rolled back.

# Other Schemes: Optimistic Concurrency Control

- If for all $T_i$ with TS ($T_i$) < TS ($T_j$) either one of the following condition holds:
  - **finish**($T_i$) < **start**($T_j$)
  - **start**($T_j$) < **finish**($T_i$) < **validation**($T_j$) **and** the set of data items written by $T_i$ does not intersect with the set of data items read by $T_j$.

  then validation succeeds and $T_j$ can be committed. Otherwise, validation fails and $T_j$ is aborted.
- *Justification*: Either the first condition is satisfied, and there is no overlapped execution, or the second condition is satisfied and
  - the writes of $T_j$ do not affect reads of $T_i$ since they occur after $T_i$ has finished its reads.
  - the writes of $T_i$ do not affect reads of $T_j$ since $T_j$ does not read any item written by $T_i$.

# Other Schemes: Optimistic Concurrency Control

- Example of schedule produced using validation

| $T_{25}$ | $T_{26}$ |
|---|---|
| read ($B$) | |
| | read ($B$) |
| | $B := B \quad 50$ |
| | read ($A$) |
| | $A := A + 50$ |
| read ($A$) | |
| ⟨ *validate* ⟩ | |
| display ($A + B$) | |
| | ⟨ *validate* ⟩ |
| | write ($B$) |
| | write ($A$) |

# Other CC Schemes: Snapshot Isolation

- Very popular scheme, used as the primary scheme by many systems including Oracle, PostgreSQL etc…
  - Several others support this in addition to locking-based protocol
- A type of “optimistic concurrency control”
- Key idea:
  - For each object, maintain past “versions” of the data along with timestamps
    - Every update to an object causes a new version to be generated

# Other CC Schemes: Snapshot Isolation

- Read queries:
  - Let “t” be the “time-stamp” of the query, i.e., the time at which it entered the system
  - When the query asks for a data item, provide a version of the data item that was latest as of “t”
    - Even if the data changed in between, provide an old version
  - No locks needed, no waiting for any other transactions or queries
  - The query executes on a consistent snapshot of the database
- Update queries (transactions):
  - Reads processed as above on a snapshot
  - Writes are done in private storage
  - At commit time, for each object that was written, check if some other transaction updated the data item since this transaction started
    - If yes, then abort and restart
    - If no, make all the writes public simultaneously (by making new versions)

# Snapshot Isolation

- A transaction T1 executing with Snapshot Isolation
  - takes snapshot of committed data at start
  - always reads/modifies data in its own snapshot
  - updates of concurrent transactions are not visible to T1
  - writes of T1 complete when it commits
  - **First-committer-wins rule**:
    - Commits only if no other concurrent transaction has already written data that T1 intends to write.

| T1 | T2 | T3 |
|---|---|---|
| W(Y := 1)<br>Commit | | |
| | Start<br>R(X) → 0<br>R(Y)→ 1 | |
| | | W(X:=2)<br>W(Z:=3)<br>Commit |
| | R(Z) → 0<br>R(Y) → 1<br>W(X:=3)<br>Commit-Req<br>Abort | |

Concurrent updates not visible
Own updates are visible
Not first-committer of X
Serialization error, T2 is rolled back

# Other CC Schemes: Snapshot Isolation

- Advantages:
  - Read query don’t block at all, and run very fast
  - As long as conflicts are rare, update transactions don’t abort either
  - Overall better performance than locking-based protocols

- Major disadvantage:
  - Not serializable
  - Inconsistencies may be introduced
  - See the wikipedia article for more details and an example
    - http://en.wikipedia.org/wiki/Snapshot_isolation

# Snapshot Isolation

- Example of problem with SI
  - T1: x:=y
  - T2: y:= x
  - Initially x = 3 and y = 17
    - Serial execution:  x = ??, y = ??
    - if both transactions start at the same time, with snapshot isolation:  x = ?? , y = ??
- Called **skew write**
- Skew also occurs with inserts
  - E.g:
    - Find max order number among all orders
    - Create a new order with order number = previous max + 1

# SI In Oracle and PostgreSQL

- **Warning**: SI used when isolation level is set to serializable, by Oracle, and PostgreSQL versions prior to 9.1
  - PostgreSQL's implementation of SI (versions prior to 9.1) described in Section 26.4.1.3
  - Oracle implements "first updater wins" rule (variant of "first committer wins")
    - concurrent writer check is done at time of write, not at commit time
    - Allows transactions to be rolled back earlier
    - Oracle and PostgreSQL < 9.1 do not support true serializable execution
  - PostgreSQL 9.1 introduced new protocol called "Serializable Snapshot Isolation" (SSI)
    - Which guarantees true serializabilty including handling predicate reads (coming up)

# The "Phantom" problem

- An interesting problem that comes up for dynamic databases
- Schema: *accounts(acct_no, balance, zipcode, …)*
- Transaction 1: Find the number of accounts in *zipcode = 20742,* and divide $1,000,000 between them
- Transaction 2: Insert *<acctX, …, 20742, …>*
- Execution sequence:
  - T1 locks all tuples corresponding to "zipcode = 20742", finds the total number of accounts (= num_accounts)
  - T2 does the insert
  - T1 computes 1,000,000/num_accounts
  - When T1 accesses the relation again to update the balances, it finds one new ("phantom") tuples (the new tuple that T2 inserted)
- Not serializable
- See this for another example

# Weak Levels of Consistency

- **Degree-two consistency**: differs from two-phase locking in that S-locks may be released at any time, and locks may be acquired at any time
  - X-locks must be held till end of transaction
  - Serializability is not guaranteed, programmer must ensure that no erroneous database state will occur]

- **Cursor stability**:
  - For reads, each tuple is locked, read, and lock is immediately released
  - X-locks are held till end of transaction
  - Special case of degree-two consistency

# Weak Levels of Consistency in SQL

- SQL allows non-serializable executions
  - **Serializable:** is the default
  - **Repeatable read**: allows only committed records to be read, and repeating a read should return the same value (so read locks should be retained)
    - However, the phantom phenomenon need not be prevented
      - T1 may see some records inserted by T2, but may not see others inserted by T2
  - **Read committed**: same as degree two consistency, but most systems implement it as cursor-stability
  - **Read uncommitted**: allows even uncommitted data to be read

- In many database systems, read committed is the default consistency level
  - has to be explicitly changed to serializable when required
    - **set isolation level serializable**

# Summary

- Concurrency control schemes help guarantee isolation while allowing for concurrent transactions
- Many different schemes developed over the years
  - Lock-based, Timestamp-based, Snapshot Isolation, Optimistic
- Lot of new work in the recent years because of shifting hardware trends
  - E.g., locking performance overheads quite significant
- Many NoSQL systems still have limited concurrency
- Important to consider recovery schemes at the same time