6.5830 Lecture 11
Two-phase Locking (recap) & Optimistic Concurrency Control

10/18/2022
Rules

• Reverse Jeopardy: The answers are the answers
• Create groups of 3-5 people
• You have to answer as a group
• Correct answers give candy points.
• Groups willing to answer the question have to raise their hand
• Among all hand-raised groups the moderator randomly picks a group.
• At the discretion of the moderator the answer is deemed correct or wrong. If correct, the group can choose the next question. If wrong, another group can answer (but no candy)
• You have to pick questions from the top down
<table>
<thead>
<tr>
<th>Terminology</th>
<th>Optimistic Concurrency Control</th>
<th>2 Phase Locking</th>
<th>Other Concurrency Techniques</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Candy</td>
<td>2 Candy</td>
<td>1 Candy</td>
<td>1 Candy</td>
</tr>
<tr>
<td>2 Candy</td>
<td>4 Candy</td>
<td>2 Candy</td>
<td>2 Candy</td>
</tr>
<tr>
<td>3 Candy</td>
<td>8 Candy</td>
<td>3 Candy</td>
<td>3 Candy</td>
</tr>
<tr>
<td>4 Candy</td>
<td>8 Candy</td>
<td>4 Candy</td>
<td>4 Candy</td>
</tr>
<tr>
<td>5 Candy</td>
<td>8 Candy</td>
<td>5 Candy</td>
<td>5 Candy</td>
</tr>
</tbody>
</table>
1 Candy

What is the connection between Atomicity, Isolation, and Durability with Concurrency Control, Logging?
Today Sam and I almost dressed alike. Let’s assume the process of dressing is done as part of a transaction, which ensures that we always dress differently.

What ACID properties were violated?
3 Candy

Name 2 types of serializability and give an example when one is valid under the definition but not the other?
## View vs Conflict Serializable

- Testing for view serializability is NP-Hard
  - Have to consider all possible orderings
- Conflict serializability used in practice
  - Not because of NP-Hardness
  - Because we have a way to enforce it as transactions run
- Example of schedule that is view serializable but not conflict serializable:

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>RA</td>
<td>RA</td>
<td>RA</td>
<td>RA</td>
</tr>
<tr>
<td>WA</td>
<td>WA</td>
<td>WA</td>
<td>WA</td>
</tr>
<tr>
<td>RB</td>
<td>RB</td>
<td>RB</td>
<td>RB</td>
</tr>
<tr>
<td>WB</td>
<td>WB</td>
<td>WB</td>
<td>WB</td>
</tr>
</tbody>
</table>

Equivalent to T1, T2, T3
Conflict serializability does not permit this
Only happens with *blind writes*

![Diagram showing the relationship between transactions and actions](image)
4 Candy

If a system ensures Atomicity, Durability, and Isolation, is it by definition not also Consistent? Explain by means of an example why/why not?
From the assigned reading from lecture 11:
WAL, LSN, Undo, Redo, physical, logical, physiological are all important terms for describing logging. What do they stand for?
2 Candy

What is the core idea of optimistic concurrency control?

What are the pros and cons?

(You should know that based on the assigned reading :)

---

2.1
Optimistic Concurrency Control (OCC)

- Alternative to locking for isolation

- Approach:
  - Store writes in a per-transaction buffer
  - Track read and write sets
  - At commit, check if transaction conflicted with earlier (concurrent) transactions
  - Abort transactions that conflict
  - Install writes at end of transaction

- “Optimistic” in that it does not block, hopes to “get lucky” arrive in serial interleaving
Tradeoffs

• In OCC:
  – Never have to wait for locks
  – no deadlocks

• But...
  – Transactions that conflict often have to be restarted
  – Transactions can "starve" -- e.g., be repeatedly restarted, never making progress

• OCC will do better when the restart rate is low
  – (Less contention)

• Recent work on high performance transaction processing has focused on OCC because
  – OCC checks can be done between individual transactions
    • Unlike global shared lock table
  – Modern OCC systems obtain insane throughput (> 10M xactions / sec)

E.g., https://people.eecs.berkeley.edu/~wzheng/silo.pdf
What are the 3 phases of optimistic concurrency control?
OCC Implementation

• Divide transaction execution in 3 phases
  – **Read**: transaction executes on DB, stores local state
  – **Validate**: transaction checks if it can commit
  – **Write**: transaction writes state to DB
Read Phase

• Transactions execute, with updates affecting local copies of the data
• Build a list of data items that were read/written
  – Read and write sets

• Modify functions to read and write data from DB
OCC Write

twrite(object, value):
    if object not in write_set:  // never written, make copy
        m = read(object)
        copies[object] = m
        write_set = write_set U {object}
        write(copies[object], value)

By writing to local copies, we ensure dirty results aren’t visible to other concurrent transactions
OCC Read

tread(object):
    read_set = read_set U {object};
    if object in write_set:
        return read(copies[object]);
    else:
        return read(object);

Allows us to read our own writes!
Validation Phase

• How do we know whether a transaction can commit?
  – Validation Rules
  – Check concurrent transactions for conflicts

• How do we implement validation efficiently?
  – Validation Algorithm

• But first... How do we order transactions?
Goal: assign transaction ids T1, ... Tn, such that this is the serial equivalent order.

When do you assign the Transaction Identifier?

a) At the beginning of the trx,
b) At the start of the validation phase
c) At the start of the write phase
d) At the end of the write phase
Transaction Identifier Assignment

• Goal: assign transaction ids T1, ... Tn, such that this is the serial equivalent order

• When should we assign transaction identifiers?
  • At start of read phase?
    – No! Would be “pessimistic” – don’t want to pre-assign the transaction order before transactions finish running
    – Long running transactions would have to commit before later short transactions

• Assign at end of read phase, just before validation starts
Under OCC there are 4 conditions. When Tj completes its read phase, require that for all Ti < Tj, **one of the following** conditions must be true for validation to succeed (Tj to commit) to ensure serializability:

1) Ti completes its write phase before Tj starts its read phase
2) W(Ti) does not intersect with R(Tj) or W(tj)
3) W(Ti) does not intersect R(Tj), and Ti completes its write phase before Tj starts its write phase.
4) W(Ti) does not intersect R(Tj) or W(Tj), and Ti completes its read phase before Tj completes its read phase.
5) W(Ti) does not intersect R(Tj) or W(Tj), and W(Tj) does not intersect R(Ti) [no conflicts]

One of the conditions is not sufficient! Which one?
Validation Rules

When Tj completes its read phase, require that for all Ti < Tj, one of the following conditions must be true for validation to succeed (Tj to commit):

1) Ti completes its write phase before Tj starts its read phase

2) W(Ti) does not intersect R(Tj), and Ti completes its write phase before Tj starts its write phase.

3) W(Ti) does not intersect R(Tj) or W(Tj), and Ti completes its read phase before Tj completes its read phase.

4) W(Ti) does not intersect R(Tj) or W(Tj), and W(Tj) does not intersect R(Ti) [no conflicts]

These rules will ensure serializability, with Tj being ordered after Ti with respect to conflicts

XXX confusion about W and R as Read / write sets vs read / write phases
Condition 1

Ti completes its write phase before Tj starts its read phase

Don't overlap at all.
**Condition 2**

$W(T_i)$ does not intersect $R(T_j)$, and $T_i$ completes its write phase before $T_j$ starts its write phase.

$W(T_i) \cap R(T_j) = \{ \}$ \hspace{1cm} $R(T_i) \cap W(T_j) \neq \{ \}$ \hspace{1cm} $W(T_i) \cap W(T_j) \neq \{ \}$

$T_j$ doesn’t read anything $T_i$ wrote.

Anything $T_j$ wrote that $T_i$ also wrote will be installed afterwards.

Anything $T_i$ read will not reflect $T_j$’s writes.
**Condition 3**

W(Ti) does not intersect R(Tj) or W(Tj), and Ti completes its read phase before Tj completes its read phase.

\[
\begin{align*}
W(Ti) \cap R(Tj) &= \{\} & R(Ti) \cap W(Tj) &\neq \{\} & W(Ti) \cap W(Tj) &= \{\}
\end{align*}
\]

Tj doesn’t read or write anything Ti wrote (but Ti may read something Tj writes).

Ti definitely won’t see any of Tj’s writes, because it finishes reading before Tj starts validation, so Ti ordered before Tj.

Ti will always complete its read phase before Tj b/c xaction IDs assigned after read phase
If no conditions apply, abort!

Restating previous rules, aborts required if:

1) $W(T_i) \cap R(T_j) \neq \{\}$, and $T_i$ does not finish writing before $T_j$ starts, $T_j$ must abort, because $T_j$ may have only seen some of what $T_i$ wrote

or

2) $W(T_i) \cap (W(T_j) \cup R(T_j)) \neq \{\}$, and $T_j$ overlaps with $T_i$ validation or write phase, $T_j$ must abort because it needs its writes to all appear after $T_i$’s writes
8 Candy

Validate Implementation
Validate Implementation

• Several different implementations designed to provide different levels of concurrency during writeback

• Transaction initialization:

```java
tnc = 0; // current transaction id
void tbegin {
    read_set = new Set();
    write_set = new Set();
    start_tn = tnc; // the transaction that finished just before this one started
}
```
Serial Validation

validateAndWrite(pastT[], start_tn, my_read_set, my_write_set) {

    lock();
    int finish_tn = tnc;  //prior transaction
    bool valid = true;
    for(int t = start_tn + 1; t <= finish_tn; t++)
        if(pastT[t].write_set intersects with my_read_set)
            valid = false;
    if (valid) {
        write_phase();
        tnc = tnc+1;
        tn = tnc;
    }
    unlock();
}
Example

Last transaction that validated before this transaction started

Transactions that validated and wrote while this transaction was in read phase

Have to compare against T6, T7, and T8
Which of the following transactions would serial validation allow to commit, assuming the transactions are concurrent and \( T_i \) completes its write phase before \( T_j \) starts its write phase \( \forall i < j \)?

Aborts required if:

1) \( W(T_i) \cap R(T_j) \neq \{ \} \), and \( T_i \) does not finish writing before \( T_j \) starts, \( T_j \) must abort, because \( T_j \) may have only some of what \( T_i \) wrote.

or

2) \( W(T_i) \cap (W(T_j) \cup R(T_j)) \neq \{ \} \), and \( T_j \) overlaps with \( T_i \) validation or write phase, \( T_j \) must abort because it needs its writes to all appear after \( T_i \)'s writes.

A. 

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>RA</td>
<td></td>
<td>RC</td>
</tr>
<tr>
<td>WA</td>
<td></td>
<td>WA</td>
</tr>
<tr>
<td>RB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

B. 

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>RA</td>
<td></td>
<td>RB</td>
<td></td>
</tr>
<tr>
<td>WA</td>
<td></td>
<td>WB</td>
<td></td>
</tr>
</tbody>
</table>

C. 

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>RA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WC</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Blind write
### 1 Candy

**Is the following schedule permitted by two-phase locking?**

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ A</td>
<td></td>
<td>READ A</td>
</tr>
<tr>
<td></td>
<td>WRITE A</td>
<td></td>
</tr>
<tr>
<td>WRITE A</td>
<td></td>
<td>WRITE B</td>
</tr>
<tr>
<td>WRITE A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>COMMIT</td>
<td>WRITE B</td>
<td>COMMIT</td>
</tr>
<tr>
<td>COMMIT</td>
<td></td>
<td>COMMIT</td>
</tr>
</tbody>
</table>

**Is this schedule view serializable? Is it conflict serializable?**
2 Candy

Which of the following schedules would rigorous 2PL permit?

T1
RA
WA

T2
RA
WA

T1
RA
WA

T2
RC
WC

T1
RA
WA

T2
RB
WB
COMMIT

T1 does not release lock on A until COMMIT (note under rigorous 2PL T2 would wait so this schedule would not happen)
### 3.3 Candy

Consider the following banking schema (a typical design):

<table>
<thead>
<tr>
<th>account_nb</th>
<th>owner</th>
<th>status</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Karin</td>
<td>open</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>id</th>
<th>account_nb</th>
<th>description</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>2</td>
<td>Candy for Tim</td>
<td>$-40</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>Ski pass</td>
<td>$-50</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

You have decided with your girl-/boyfriend to consolidate the accounts by closing ACCOUNT_NB=1 with the following trx:

\[
\text{acct\_sum} = \text{SELECT SUM(amount) FROM bank\_trx WHERE account\_nb = 1}
\]

\[
\text{INSERT (100, 2, \text{"Transfer from Acct 1"}, acct\_sum) INTO bank\_trx}
\]

\[
\text{INSERT (101, 1, \text{"Transfer to Acct 2"}, -acct\_sum) INTO bank\_trx}
\]

\[
\text{UPDATE account SET status=closed WHERE account\_nb = 1}
\]

**Assuming transfers always check the account status, why is 2-Phase Locking from class not sufficient to ensure that the balance of account 1 is 0 after closing it? How would you prevent the problem?**
Final Wrinkle: Phantoms

- T1 scans a range; T2 later inserts into that range
- If T1 scans the range again, it will see a new value

If we are just locking, e.g., records, this insertion would be allowed in all 2PL algos we have studied, but is not serializable (since this couldn’t happen in a serial execution).
Solving Phantoms

• Need a way to lock ranges
• Common approach: next key locking

On insert(val), Xlock ij next pointer if val > max(page i) and < min(page(j))

Only works for ranges with indexes
For unindexed tables, must read the whole table, so just use a table lock
More details next lecture!
4 Candy

How would you implement 2PL and prevent Phantoms in GoDB without a Btree?
Implementing 2PL

- **GoDB**: Lock Table
  - Buffer pool maintains a table of locks per page
  - Transactions acquire locks on reads/writes of pages
  - Release locks at commit

- Access to lock table will need to be synchronized
What is the conflict graph for this schedule, and is it serializable?

For an operation pair that conflicts in T1 and T2, the operation happens first in T1.
1 Candy

What if Serializability isn’t needed?
What If Serializability Isn’t Needed?

• E.g., application only needs to read committed data
• Databases provide different isolation levels
  – READ UNCOMMITTED
    • Ok to read other transaction’s dirty data
  – READ COMMITTED
    • Only read committed values
  – REPEATABLE READS
    • If R1 reads A=x, R2 will read A=x ∀ A

• Many database systems default to READ COMMITTED
READ UNCOMMITTED w/ Locking

• If OK reading uncommitted data, no need to check if records that are read are locked

• However, to prevent other transactions from seeing dirty data, need to hold write locks for the duration of the transaction

• May be OK if, e.g., just reporting some statistic, like number of users or views
READ COMMITTED w/ Locking

• To ensure that a transaction only reads committed values, need to acquire locks before reading
  – If all other transactions hold write locks (as in READ UNCOMMITTED), it will never read a dirty value

• Since we doesn’t care about always reading the same value, OK to release locks after a value is read

• As in READ UNCOMMITTED, write locks still need to be held for the duration of the transaction
Is this schedule permitted under (a) read uncommitted, (b) read committed, (c) repeatable read read?
**READ COMMITTED Example**

This schedule is permitted by READ COMMITTED

These reads see different values of A

Short duration read lock

Additional concurrency!
REPEATABLE READ w/ Locking

• If we want to always read the same value, need to hold read locks for transaction duration

• So how is this different from SERIALIZABLE?

• SERIALIZABLE also needs to prevent phantoms
2 Candy

Multi-version concurrency control (MVCC) / Snapshot Isolation (SI)
REPEATABLE READ vs SERIALIZABLE

• Some systems, e.g., Postgres implement REPEATABLE READ through a different mechanism based on database snapshots taken at the start of transaction
  – Called “multiversion concurrency control” – yet another way of achieving isolation!

• This has other problems besides phantoms – so called “read skew anomalies”
  – See: https://www.cockroachlabs.com/blog/what-write-skew-looks-like/
Snapshot Isolation

• When a TA starts it receives a timestamp, T.
• All reads are carried out as of the DB version of T.
  – Need to keep historic versions of all objects!!!
• All writes are carried out in a separate buffer.
  – Writes only become visible after a commit.
• When TA commits, DBMS checks for conflicts
  – Abort TA1 with timestamp T1 if exists TA2 such that
    • TA2 committed after T1 and before TA1
    • TA1 and TA2 updated the same object
### UNDER SI

A) Does T2 abort or commit?  
B) Does T3 abort or commit?

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>R(Y)</td>
</tr>
<tr>
<td>W(Y)</td>
<td>Commit</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R(Y)</td>
<td>Start</td>
<td>R(X)</td>
<td></td>
</tr>
<tr>
<td>R(Y)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>W(X)</td>
<td></td>
<td></td>
<td>W(X)</td>
</tr>
<tr>
<td>W(Z)</td>
<td></td>
<td></td>
<td>W(Z)</td>
</tr>
<tr>
<td></td>
<td>R(Z)</td>
<td></td>
<td>Commit/Abort</td>
</tr>
<tr>
<td></td>
<td>R(Y)</td>
<td>W(X)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>W(X)</td>
<td>/Commit?</td>
</tr>
</tbody>
</table>
Is the following schedule valid under SI?

- 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.

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>R(Y)→v0</td>
</tr>
<tr>
<td></td>
<td>W(Y := v1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Commit</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Start</td>
<td></td>
<td>Start</td>
<td></td>
</tr>
<tr>
<td></td>
<td>R(X) → v0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>R(Y)→ v1</td>
<td>W(X:=v2)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>W(Z:=v1)</td>
<td>W(Z:=v1)</td>
<td>Commit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R(Z) → v0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>R(Y) → v1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>W(X:=v3)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Abort</td>
</tr>
</tbody>
</table>

Concurrent updates not visible
Own updates are visible
Not first-committer of X
Serialization error, T2 is rolled back
For what type of workloads should you use 2PL for what type of workloads SI?
Create a real-world-inspired example, (e.g., from banking, hotel booking, ...) for which two transactions executed with Snapshot Isolation guarantees would violate serializability.
• **Observation:** Snapshot isolation does not prevent Write-Read conflicts, since it doesn’t check whether it read something another transaction wrote

• This leads to so-called write-skew, i.e.:

```
  T1   T2
 RX
   RY
 WY
  WX
```

• Neither transaction saw the other’s write; this would not be permitted under serializability

Real world example:
• Employee X and employee Y are signing up for shifts
• They hate each other and don’t want to work together
• EY checks that EX isn’t working (RX)
• EX checks that EY isn’t working (RY)
• Both update their schedule to work on the same day
Does SI isolation have the phantom problem?
Under 2-Phase Locking, the following locks got requested from the Lock Manager. Is there a deadlock? Can you think of a single change to create/resolve the deadlock?

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>RA</td>
<td>RB</td>
<td>RC</td>
</tr>
<tr>
<td>WB</td>
<td>RC</td>
<td>RA</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>time</th>
<th>Type</th>
<th>TID</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1</td>
<td>Req. ReadLock A</td>
<td>T1</td>
</tr>
<tr>
<td>t2</td>
<td>Req. ReadLock B</td>
<td>T2</td>
</tr>
<tr>
<td>t3</td>
<td>Req. ReadLock C</td>
<td>T3</td>
</tr>
<tr>
<td>t4</td>
<td>Req. ReadLock A</td>
<td>T3</td>
</tr>
<tr>
<td>t5</td>
<td>Req. WriteLock B</td>
<td>T1</td>
</tr>
<tr>
<td>t6</td>
<td>Req. ReadLock C</td>
<td>T2</td>
</tr>
</tbody>
</table>