17 Timestamp Ordering
ADMINISTRIVIA

Project 3 ongoing
→ Due Sunday, April 9th at 11:59 p.m.

Homework 4 released today
→ Due Friday, April 7th at 11:59 p.m.

Final exam Monday, May 1st, 8:30 – 11:30 a.m.
PROJECT #3 – QUERY EXECUTION

You will add support for executing queries in BusTub.

BusTub supports (basic) SQL with a rule-based optimizer for converting AST into physical plans.

Prompt: A realistic photo of a bath tub with wheels and cartoon eyes driving down a city street.

https://15445.courses.cs.cmu.edu/spring2023/project3/
PROJECT #3 – TASKS

Plan Node Executors
→ Access Methods: Sequential Scan, Index Scan
→ Modifications: Insert, Update, Delete
→ Joins: Nested Loop Join, Hash Join
→ Miscellaneous: Aggregation, Limit, Sort, Top-N

Optimizer Rules:
→ Convert Nested Loop Join into a Hash Join
→ Convert ORDER BY + LIMIT into a Top-N
The leaderboard requires you to add additional rules to the optimizer to generate query plans.

→ It will be impossible to get a top ranking by just having the fastest implementations in Project #1 + Project #2.
DEVELOPMENT HINTS

Implement the Insert and Sequential Scan executors first so that you can populate tables and read from it.

You do not need to worry about transactions.

The aggregation and hash join hash tables do not need to be backed by the buffer pool (i.e., use STL).

Gradescope is meant for grading, not debugging. Please write your own local tests.
THINGS TO NOTE

Do **not** change any file other than the ones that you submit to Gradescope.

Make sure you pull in the latest changes from the BusTub main branch.

Post your questions on Piazza or come to TA office hours.

Compare against our solution in your browser!
LAST TIME: TWO-PHASE LOCKING

Two-phase locking (2PL)
→ Regular 2PL
→ Strong strict 2PL

Deadlocks
→ Detection
→ Prevention

Hierarchical intention locks
INTENTION LOCKS

Intention-Shared (IS)
→ Indicates explicit locking at lower level with S locks.
→ Intent to get S lock(s) at finer granularity.

Intention-Exclusive (IX)
→ Indicates explicit locking at lower level with X locks.
→ Intent to get X lock(s) at finer granularity.

Shared+Intention-Exclusive (SIX)
→ The subtree rooted by that node is locked explicitly in S mode and explicit locking is being done at a lower level with X locks.
## COMPATIBILITY MATRIX

<table>
<thead>
<tr>
<th></th>
<th>IS</th>
<th>IX</th>
<th>S</th>
<th>SIX</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>IS</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td>IX</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>S</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>SIX</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>X</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
</tbody>
</table>
CONCURRENCY CONTROL APPROACHES

**Two-Phase Locking (2PL)**
→ Determine serializability order of conflicting operations at runtime while txns execute.

**Timestamp Ordering (T/O)**
→ Determine serializability order of txns before they execute.

*Pessimistic*

*Optimistic*
T/O CONCURRENCY CONTROL

Use timestamps to determine the serializability order of txns.

If $\text{TS}(T_i) < \text{TS}(T_j)$, then the DBMS must ensure that the execution schedule is equivalent to a serial schedule where $T_i$ appears before $T_j$. 

Each txn $T_i$ is assigned a unique fixed timestamp that is monotonically increasing

$\rightarrow$ Let $TS(T_i)$ be the timestamp allocated to txn $T_i$
$\rightarrow$ Different schemes assign timestamps at different times during the txn

Multiple implementation strategies:
$\rightarrow$ System/Wall Clock
$\rightarrow$ Logical Counter
$\rightarrow$ Hybrid
TODAY'S AGENDA

Basic Timestamp Ordering (T/O) Protocol
Optimistic Concurrency Control
The Phantom Problem (maybe)
BASIC TIMESTAMP ORDERING (T/O)

Txns read and write objects without locks.

Every object \( X \) is tagged with timestamp of the last txn that successfully did read/write:

\[ \rightarrow W-TS(X) \] – Write timestamp on \( X \)
\[ \rightarrow R-TS(X) \] – Read timestamp on \( X \)

Check timestamps for every operation:

\[ \rightarrow \] If txn tries to access an object written with a higher (future) timestamp, it aborts and restarts
BASIC T/O – READS

If $TS(T_i) < W-TS(X)$, this violates timestamp order of $T_i$ with regard to the writer of $X$.
→ Abort $T_i$ and restart it with a new TS.

Else:
→ Allow $T_i$ to read $X$.
→ Update $R-TS(X)$ to $\max(R-TS(X), TS(T_i))$.
→ Make a local copy of $X$ to ensure repeatable reads for $T_i$. 
If $\text{TS}(T_i) < R-\text{TS}(X)$ or $\text{TS}(T_i) < W-\text{TS}(X)$

→ Abort and restart $T_i$.

Else:

→ Allow $T_i$ to write $X$ and update $W-\text{TS}(X)$
→ Also make a local copy of $X$ to ensure repeatable reads.
BASIC T/O – EXAMPLE #1

Schedule

<table>
<thead>
<tr>
<th>Time</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>BEGIN R(B)</td>
</tr>
<tr>
<td></td>
<td>R(B)</td>
</tr>
<tr>
<td></td>
<td>R(A)</td>
</tr>
<tr>
<td></td>
<td>R(A)</td>
</tr>
<tr>
<td></td>
<td>W(A)</td>
</tr>
<tr>
<td>T2</td>
<td>BEGIN R(B)</td>
</tr>
<tr>
<td></td>
<td>R(B)</td>
</tr>
<tr>
<td></td>
<td>W(B)</td>
</tr>
<tr>
<td></td>
<td>R(A)</td>
</tr>
<tr>
<td></td>
<td>W(A)</td>
</tr>
<tr>
<td></td>
<td>COMMIT</td>
</tr>
<tr>
<td></td>
<td>COMMIT</td>
</tr>
</tbody>
</table>

Database

<table>
<thead>
<tr>
<th>Object</th>
<th>R-TS</th>
<th>W-TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

TS(T₁) = 1
TS(T₂) = 2

TS(T₁) < TS(T₂)

No violations so both txns are safe to commit.
**BASIC T/O – EXAMPLE #2**

**Schedule**

- **T₁**: BEGIN R(A) W(A) COMMIT
- **T₂**: BEGIN W(A) COMMIT

**Database**

<table>
<thead>
<tr>
<th>Object</th>
<th>R-TS</th>
<th>W-TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Violation: \( TS(T₁) < W-TS(A) \)

\( T₁ \) cannot overwrite update by \( T₂ \), so the DBMS must abort it!
If $TS(T_i) < R - TS(X)$:
→ Abort and restart $T_i$.

If $TS(T_i) < W - TS(X)$:
→ Thomas Write Rule: Skip the write and allow the txn to continue executing without aborting.

Else:
→ Allow $T_i$ to write $X$ and update $W - TS(X)$. 

Creeper and Reaper

From Wikipedia, the free encyclopedia
(Please directed from Creeper (program))

Creeper was the first computer worm, while Reaper was the first antivirus software, designed to eliminate Creeper.

Creeper [edit]

Creeper was an experimental computer program written by Bob Thomas at BBN in 1971.[2] Its original iteration was designed to move between DEC PDP-10 mainframe computers running the TENEX operating system using the ARPANET, with a later version by Ray Tomlinson designed to copy itself between computers rather than simply move.[3] This self-replicating version of Creeper is generally accepted to be the first computer worm.[1][4] Creeper was a test created to demonstrate the possibility of a self-replicating computer program that could spread to other computers.

The program was not actively malicious software as it caused no damage to data, the only effect being a message it output to the teletype reading "I'M THE CREEPER. CATCH ME IF YOU CAN!".[5][4]
BASIC T/O – EXAMPLE #2

Schedule

T₁
BEGIN
R(A)

T₂
BEGIN
W(A)
COMMIT

Database

<table>
<thead>
<tr>
<th>Object</th>
<th>R-TS</th>
<th>W-TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

We do not update W-TS(A)

Skip this write, and allow T₁ to commit.
BASIC T/O

Generates a schedule that is conflict serializable if you do **not** use the **Thomas Write Rule**.
→ No deadlocks because no txn ever waits.
→ Possibility of starvation for long txns if short txns keep causing conflicts.

We’re not aware of any DBMS that uses the basic T/O protocol described here.
→ It provides the building blocks for OCC / MVCC.
PARTICIPATION EXERCISE

Why does no real database system use the basic timestamp ordering protocol?

OBSERVATION

If you assume that conflicts between txns are rare and that most txns are short-lived, then forcing txns to acquire locks or update timestamps adds unnecessary overhead.

A better approach is to optimize for the no-conflict case.
The DBMS creates a private workspace for each txn.

Any object read is copied into workspace.

Modifications are applied to workspace.

When a txn commits, the DBMS compares workspace write set to see whether it conflicts with other txns.

If there are no conflicts, the write set is installed into the "global" database.
OCC PHASES

#1 – Read Phase:
→ Track the read/write sets of txns and store their writes in a private workspace.

#2 – Validation Phase:
→ When a txn commits, check whether it conflicts with other txns.

#3 – Write Phase:
→ If validation succeeds, apply private changes to database. Otherwise abort and restart the txn.
OCC – EXAMPLE

Schedule

T₁

BEGIN
READ R(A)
VALIDATE
WRITE W(A)
COMMIT

TS(T₁) = 2

T₂

BEGIN
READ R(A)
VALIDATE
WRITE W(A)
COMMIT

TS(T₂) = 1

Database

<table>
<thead>
<tr>
<th>Object</th>
<th>Value</th>
<th>W-TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>456</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

T₁ Workspace

<table>
<thead>
<tr>
<th>Object Value</th>
<th>W-TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>456</td>
</tr>
<tr>
<td></td>
<td>∞</td>
</tr>
</tbody>
</table>

T₂ Workspace

<table>
<thead>
<tr>
<th>Object Value</th>
<th>W-TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>123</td>
</tr>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
OCC – READ PHASE

Track the read/write sets of txns and store their writes in-memory in a private workspace.

The DBMS copies every tuple that the txn accesses from the shared database to its workspace ensure repeatable reads.

→ We can ignore for now what happens if a txn reads/writes tuples via indexes.
When txn $T_i$ invokes **COMMIT**, the DBMS checks if it conflicts with other txns.

→ The DBMS needs to guarantee only serializable schedules are permitted.

→ Checks other txns for RW and WW conflicts and ensure that conflicts are in one direction (e.g., older$\rightarrow$younger).

**Approach #1: Backward Validation**

**Approach #2: Forward Validation**
OCC – BACKWARD VALIDATION

Check whether the committing txn intersects its read/write sets with those of any txns that have already committed.
OCC – FORWARD VALIDATION

Check whether the committing txn intersects its read/write sets with any active txns that have \textbf{not} yet committed.
OCC – FORWARD VALIDATION

Each txn's timestamp is assigned at the beginning of the validation phase.

Check the timestamp ordering of the committing txn with all other running txns.

If $TS(T_i) < TS(T_j)$, then one of three cases:
OCC – FORWARD VALIDATION CASE #1

\( T_i \) completes all three phases before \( T_j \) begins its execution.

This is a serial ordering, so there is no conflict.
OCC – FORWARD VALIDATION CASE #2

$T_i$ completes before $T_j$ starts its Write phase.

If $T_i$ does not write to any object read by $T_j$, then there is no conflict.

Abort $T_i$ if $\text{WriteSet}(T_i) \cap \text{ReadSet}(T_j) \neq \emptyset$
OCC - FORWARD VALIDATION CASE #2

T₁ must abort even though T₂ will never write to the database.
OCC - FORWARD VALIDATION CASE #2

Safe to commit $T_1$ because $T_2$ commits logically before $T_1$
OCC – FORWARD VALIDATION CASE #3

$T_i$ completes its **Read** phase before $T_j$ completes its **Read** phase.

If $T_i$ does not write to any object that is either read or written by $T_j$, then there is no conflict.

Abort $T_i$ if $\text{WriteSet}(T_i) \cap \text{ReadSet}(T_j) \neq \emptyset$

or if $\text{WriteSet}(T_i) \cap \text{WriteSet}(T_j) \neq \emptyset$
OCC - FORWARD VALIDATION CASE #3

Schedule

```
T1
BEGIN
READ
R(A)
W(A)
VALIDATE
WRITE
COMMIT

T2
BEGIN

TS(T1)=1
```

Database

<table>
<thead>
<tr>
<th>Object</th>
<th>Value</th>
<th>W-TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>456</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>XYZ</td>
<td>0</td>
</tr>
</tbody>
</table>

T1 Workspace

<table>
<thead>
<tr>
<th>Object</th>
<th>Value</th>
<th>W-TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>456</td>
<td>∞</td>
</tr>
</tbody>
</table>

Safe to commit T1 because T2 sees the DB after T1 has executed.

T2 Workspace

<table>
<thead>
<tr>
<th>Object</th>
<th>Value</th>
<th>W-TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>XYZ</td>
<td>0</td>
</tr>
<tr>
<td>A</td>
<td>456</td>
<td>1</td>
</tr>
</tbody>
</table>
Propagate changes in the txn's write set to database to make them visible to other txns.

**Serial Commits:**
→ Use a global latch to limit a single txn to be in the **Validation/Write** phases at a time.

**Parallel Commits:**
→ Use fine-grained write latches to support parallel **Validation/Write** phases.
→ Txns acquire latches in primary key order to avoid deadlocks.
OCC works well when the # of conflicts is low:
→ All txns are read-only (ideal).
→ Txns access disjoint subsets of data.

If the database is large and the workload is not skewed, then there is a low probability of conflict, then locking is wasteful.
OCC – PERFORMANCE ISSUES

High overhead for copying data locally.

Validation/Write phase bottlenecks.

Aborts are more wasteful than in 2PL because they only occur after a txn has already executed.
Recall that so far, we have only dealt with transactions that read and update existing objects in the database.

But now if txns perform insertions, updates, and deletions, we have new problems…
THE PHANTOM PROBLEM

Schedule

```
BEGIN
SELECT COUNT(age)
FROM people
WHERE status='lit'
COMMIT
```

```
BEGIN
SELECT COUNT(age)
FROM people
WHERE status='lit'
COMMIT
```

```
CREATE TABLE people (  
id SERIAL,  
name VARCHAR,  
age INT,  
status VARCHAR  
);
```

```sql
INSERT INTO people (age=30, status='lit')
```

99

100

TIME
HOW DID THIS HAPPEN?

Because $T_1$ locked only existing records and not ones under way!

Conflict serializability on reads and writes of individual items guarantees serializability only if the set of objects is fixed.
THE PHANTOM PROBLEM

Approach #1: Re-Execute Scans
→ Run queries again at commit to see whether they produce a different result to identify missed changes.

Approach #2: Predicate Locking
→ Logically determine the overlap of predicates before queries start running.

Approach #3: Index Locking
→ Use keys in indexes to protect ranges.
RE-EXECUTE SCANS

The DBMS tracks the WHERE clause for all queries that the txn executes.
→ Retain the scan set for every range query in a txn.

Upon commit, re-execute just the scan portion of each query and check whether it generates the same result.
→ Example: Run the scan for an UPDATE query but do not modify matching tuples.
PREDICATE LOCKING

Proposed locking scheme from System R.
→ Shared lock on the predicate in a `WHERE` clause of a `SELECT` query.
→ Exclusive lock on the predicate in a `WHERE` clause of any `UPDATE`, `INSERT`, or `DELETE` query.

Never implemented in any system except for HyPer (precision locking).
SELECT COUNT(age) 
FROM people 
WHERE status='lit'

INSERT INTO people VALUES 
(age=30, status='lit')

Records in Table "people"
INDEX LOCKING SCHEMES

Key-Value Locks
Gap Locks
Key-Range Locks
Hierarchical Locking
**KEY-VALUE LOCKS**

Locks that cover a single key-value in an index. Need “virtual keys” for non-existent values.

**B+Tree Leaf Node**

```
10  12  [14, 14]  16
```

**Key [14, 14]**
GAP LOCKS

Each txn acquires a key-value lock on the single key that it wants to access. Then get a gap lock on the next key gap.

B+Tree Leaf Node

10  {Gap}  12  {Gap}  14  {Gap}  16

Gap (14, 16)
KEY-RANGE LOCKS

A txn takes locks on ranges in the key space.
→ Each range is from one key that appears in the relation, to the next that appears.
→ Define lock modes so conflict table will capture commutativity of the operations available.
KEY-RANGE LOCKS

Locks that cover a key value and the gap to the next key value in a single index.
→ Need “virtual keys” for artificial values (infinity)
Hierarchical Locking

Allow for a txn to hold wider key-range locks with different locking modes.
→ Reduces the number of visits to lock manager.
LOCKING WITHOUT AN INDEX

If there is no suitable index, then to avoid phantoms the txn must obtain:
→ A lock on every page in the table to prevent a record’s status='lit' from being changed to lit.
→ The lock for the table itself to prevent records with status='lit' from being added or deleted.
CONCLUSION

Every concurrency control can be broken down into the basic concepts that I've described in the last two lectures.

Every protocol has pros and cons.
NEXT CLASS

Multi-Version Concurrency Control