ADVANCED DATABASE SYSTEMS

Optimistic Concurrency Control

@Andy_Pavlo // 15-721 // Spring 2018
Project #1 is due Monday Jan 29\textsuperscript{th} @ 11:59pm

Project #2 will be released on Wednesday.
→ You need a group of three people.
→ I will send out a sign-up sheet.
CMU Database Group Meetings
→ Mondays @ 4:30pm in GHC 8102
→ http://db.cs.cmu.edu/events

Peloton Developer Team Meetings
→ Tuesdays @ 12:00pm in GHC 7501
→ There will be food. Everyone loves food.
TODAY’S AGENDA

Stored Procedures
Optimistic Concurrency Control
Modern OCC Implementations
Disk stalls are (almost) gone when executing txns in an in-memory DBMS.

There are still other stalls when an app uses conversational API to execute queries on DBMS → ODBC/JDBC → DBMS-specific wire protocols
CONVERSATIONAL DATABASE API

Application

BEGIN

\text{SQL}

Program Logic

\text{SQL}

Program Logic : :

\text{COMMIT}
CONVERSATIONAL DATABASE API

Application

BEGIN

SQL

Program Logic

SQL

Program Logic

;

COMMIT
CONVERSATIONAL DATABASE API

Application

BEGIN

SQL
Program Logic

SQL
Program Logic

;

COMMIT

Parser
Planner
Optimizer
Query Execution
CONVERSATIONAL DATABASE API

Application

BEGIN
  SQL
  Program Logic
  SQL
  Program Logic
  ;
  COMMIT

Parser
Planer
Optimizer
Query Execution
CONVERSATIONAL DATABASE API

Application

BEGIN

SQL
Program Logic

SQL
Program Logic

;

COMMIT

Parser
Planner
Optimizer
Query Execution
SOLUTIONS

**Prepared Statements**
→ Removes query preparation overhead.

**Query Batches**
→ Reduces the number of network roundtrips.

**Stored Procedures**
→ Removes both preparation and network stalls.
A **stored procedure** is a group of queries that form a logical unit and perform a particular task on behalf of an application directly inside of the DBMS.

Programming languages:
→ **SQL/PSM** (standard)
→ **PL/SQL** (Oracle / IBM / MySQL)
→ **PL/pgSQL** (Postgres)
→ **Transact-SQL** (Microsoft / Sybase)
STORED PROCEDURES

Application

CALL PROC(x=99)

PROC(x)

BEGIN
SQL
Program Logic
SQL
Program Logic :
COMMIT
CREATE PROCEDURE testProc
    (num INT, name VARCHAR) RETURNS INT
BEGIN
    DECLARE cnt INT DEFAULT 0;
    LOOP
        INSERT INTO student VALUES (cnt, name);
        SET cnt := cnt + 1;
        IF (cnt > num) THEN
            RETURN cnt;
        END IF;
    END LOOP;
END;
ADVANTAGES

Reduce the number of round trips between application and database servers.

Increased performance because queries are pre-compiled and stored in DBMS.

Procedure reuse across applications.

Server-side txn restarts on conflicts.
Not as many developers know how to write SQL/PSM code.
→ Safe Languages vs. Sandbox Languages

Outside the scope of the application so it is difficult to manage versions and hard to debug.

Probably not be portable to other DBMSs.

DBAs usually don’t give permissions out freely...
CONCURRENCY CONTROL

The protocol to allow txns to access a database in a multi-programmed fashion while preserving the illusion that each of them is executing alone on a dedicated system.

→ The goal is to have the effect of a group of txns on the database’s state is equivalent to any serial execution of all txns.

Provides Atomicity + Isolation in ACID
**CONCURRENCY CONTROL SCHEMES**

**Two-Phase Locking (2PL)**

→ Assume txns will conflict so they must acquire locks on database objects before they are allowed to access them.

**Timestamp Ordering (T/O)**

→ Assume that conflicts are rare so txns do not need to first acquire locks on database objects and instead check for conflicts at commit time.
TWO-PHASE LOCKING

Txn #1

Growing Phase

Shrinking Phase
TWO-PHASE LOCKING

**Txn #1**

```
BEGIN
  LOCK(A)
  READ(A)
  LOCK(B)
  WRITE(B)
  UNLOCK(A)
  UNLOCK(B)
  COMMIT
```

**Txn #2**

```
BEGIN
  LOCK(B)
  WRITE(B)
  LOCK(A)
  WRITE(A)
  UNLOCK(A)
  UNLOCK(B)
  COMMIT
```
TWO-PHASE LOCKING

**Txn #1**

- **BEGIN**
  - LOCK(A)
  - READ(A)
  - LOCK(B)
  - WRITE(B)
  - UNLOCK(A)
  - UNLOCK(B)
  - COMMIT

**Txn #2**

- **BEGIN**
  - LOCK(B)
  - WRITE(B)
  - LOCK(A)
  - WRITE(A)
  - UNLOCK(A)
  - UNLOCK(B)
  - COMMIT
TWO-PHASE LOCKING

**Txn #1**

- **BEGIN**
  - LOCK(A)
  - READ(A)
  - LOCK(B)
  - WRITE(B)
  - UNLOCK(A)
  - UNLOCK(B)
  - COMMIT

**Txn #2**

- **BEGIN**
  - LOCK(B)
  - WRITE(B)
  - LOCK(A)
  - WRITE(A)
  - UNLOCK(A)
  - UNLOCK(B)
  - COMMIT
**TWO-PHASE LOCKING**

**Txn #1**

- **BEGIN**
  - LOCK(A)
  - READ(A)
  - LOCK(B)
  - WRITE(B)
  - UNLOCK(A)
  - UNLOCK(B)
  - COMMIT

**Txn #2**

- **BEGIN**
  - LOCK(B)
  - WRITE(B)
  - LOCK(A)
  - WRITE(A)
  - UNLOCK(A)
  - UNLOCK(B)
  - COMMIT
TWO-PHASE LOCKING

Txn #1

BEGIN

LOCK(A) READ(A) LOCK(B) WRITE(B) UNLOCK(A) UNLOCK(B) COMMIT

Txn #2

BEGIN

LOCK(B) WRITE(B) LOCK(A) WRITE(A) UNLOCK(A) UNLOCK(B) COMMIT
TWO-PHASE LOCKING

**Txn #1**

BEGIN
- LOCK(A)
- READ(A)
- LOCK(B)
- WRITE(B)
- UNLOCK(A)
- UNLOCK(B)
- WRITE(B)
- COMMIT

**Txn #2**

BEGIN
- LOCK(B)
- WRITE(B)
- LOCK(A)
- READ(A)
- UNLOCK(A)
- UNLOCK(B)
- COMMIT
TWO-PHASE LOCKING

Deadlock Detection
→ Each txn maintains a queue of the txns that hold the locks that it waiting for.
→ A separate thread checks these queues for deadlocks.
→ If deadlock found, use a heuristic to decide what txn to kill in order to break deadlock.

Deadlock Prevention
→ Check whether another txn already holds a lock when another txn requests it.
→ If lock is not available, the txn will either (1) wait, (2) commit suicide, or (3) kill the other txn.
Txn #1

BEGIN

READ(A)

WRITE(B)

WRITE(A)

COMMIT
TIMESTAMP ORDERING

#1

BEGIN
READ(A)
WRITE(B)

... ... ...

WRITE(A)

COMMIT
**TIMESTAMP ORDERING**

**Record** | **Read Timestamp** | **Write Timestamp**
---|---|---
A | 10000 | 10000
B | 10000 | 10000
TIMESTAMP ORDERING

10001

#1

BEGIN

READ(A)

WRITE(B)

WRITE(A)

COMMIT

<table>
<thead>
<tr>
<th>Record</th>
<th>Read Timestamp</th>
<th>Write Timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>10000</td>
<td>10000</td>
</tr>
<tr>
<td>B</td>
<td>10000</td>
<td>10000</td>
</tr>
</tbody>
</table>
**TIMESTAMP ORDERING**

```
#1

BEGIN
READ(A)
WRITE(B)

COMMIT
WRITE(A)

<table>
<thead>
<tr>
<th>Record</th>
<th>Read Timestamp</th>
<th>Write Timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>10001</td>
<td>10000</td>
</tr>
<tr>
<td>B</td>
<td>10000</td>
<td>10000</td>
</tr>
</tbody>
</table>
```
TIMESTAMP ORDERING

#1

<table>
<thead>
<tr>
<th>Record</th>
<th>Read Timestamp</th>
<th>Write Timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>10001</td>
<td>10000</td>
</tr>
<tr>
<td>B</td>
<td>10000</td>
<td>10000</td>
</tr>
</tbody>
</table>
TIMESTAMP ORDERING

BEGIN
READ(A)
WRITE(B)

... ...

WRITE(A)

COMMIT

<table>
<thead>
<tr>
<th>Record</th>
<th>Read Timestamp</th>
<th>Write Timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>10001</td>
<td>10000</td>
</tr>
<tr>
<td>B</td>
<td>10000</td>
<td>10001</td>
</tr>
</tbody>
</table>

10001
Record   Read Timestamp | Write Timestamp
-------- | -------------------
A        | 10001               | 10005
B        | 10000               | 10001
TIMESTAMP ORDERING

### #1

| Txn #1 | BEGIN | READ(A) | WRITE(B) | COMMIT |

- **A**
  - Read Timestamp: 10001
  - Write Timestamp: 10005

- **B**
  - Read Timestamp: 10000
  - Write Timestamp: 10001
Basic T/O
→ Check for conflicts on each read/write.
→ Copy tuples on each access to ensure repeatable reads.

Optimistic Currency Control (OCC)
→ Store all changes in private workspace.
→ Check for conflicts at commit time and then merge.
OPTIMISTIC CONCURRENcy CONTROL

Timestamp-ordering scheme where txns copy data read/write into a private workspace that is not visible to other active txns.

When a txn commits, the DBMS verifies that there are no conflicts.

First proposed in 1981 at CMU by H.T. Kung.
OPTIMISTIC CONCURRENCY CONTROL

**Txn #1**

BEGIN

READ(A)

WRITE(A)

WRITE(B)

COMMIT

<table>
<thead>
<tr>
<th>Record</th>
<th>Value</th>
<th>Write Timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>123</td>
<td>10000</td>
</tr>
<tr>
<td>B</td>
<td>456</td>
<td>10000</td>
</tr>
</tbody>
</table>
**Optimistic Concurrency Control**

*Txn #1*

**Read Phase**

<table>
<thead>
<tr>
<th>Record</th>
<th>Value</th>
<th>Write Timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>123</td>
<td>10000</td>
</tr>
<tr>
<td>B</td>
<td>456</td>
<td>10000</td>
</tr>
</tbody>
</table>
**Optimistic Concurrency Control**

**Txn #1**

<table>
<thead>
<tr>
<th>Record</th>
<th>Value</th>
<th>Write Timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>123</td>
<td>10000</td>
</tr>
<tr>
<td>B</td>
<td>456</td>
<td>10000</td>
</tr>
</tbody>
</table>
Txn #1

BEGIN
READ(A)
WRITE(A)
WRITE(B)
COMMIT

Workspace

<table>
<thead>
<tr>
<th>Record</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Write Timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Record</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Write Timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Record</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>456</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Write Timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000</td>
</tr>
</tbody>
</table>
OPTIMISTIC CONCURRENCY CONTROL

**Txn #1**

BEGIN
READ(A)
WRITE(A)
WRITE(B)
COMMIT

**Workspace**

<table>
<thead>
<tr>
<th>Record</th>
<th>Value</th>
<th>Write Timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>123</td>
<td>10000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Record</th>
<th>Value</th>
<th>Write Timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>123</td>
<td>10000</td>
</tr>
<tr>
<td>B</td>
<td>456</td>
<td>10000</td>
</tr>
</tbody>
</table>
**OPTIMISTIC CONCURRENCY CONTROL**

**Txn #1**

```
BEGIN
  READ(A)
  WRITE(A)
  WRITE(B)
COMMIT
```

**Workspace**

<table>
<thead>
<tr>
<th>Record</th>
<th>Value</th>
<th>Write Timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>888</td>
<td>∞</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Record</th>
<th>Value</th>
<th>Write Timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>123</td>
<td>10000</td>
</tr>
<tr>
<td>B</td>
<td>456</td>
<td>10000</td>
</tr>
</tbody>
</table>
OPTIMISTIC CONCURRENCY CONTROL

Txn #1

BEGIN
READ(A)
WRITE(A)
WRITE(B)
COMMIT

Workspace

<table>
<thead>
<tr>
<th>Record</th>
<th>Value</th>
<th>Write Timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>888</td>
<td>∞</td>
</tr>
<tr>
<td>B</td>
<td>456</td>
<td>10000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Record</th>
<th>Value</th>
<th>Write Timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>123</td>
<td>10000</td>
</tr>
<tr>
<td>B</td>
<td>456</td>
<td>10000</td>
</tr>
</tbody>
</table>
**OPTIMISTIC CONCURRENCY CONTROL**

**Txn #1**

BEGIN

- READ(A)
- WRITE(A)
- WRITE(B)
- COMMIT

**Workspace**

<table>
<thead>
<tr>
<th>Record</th>
<th>Value</th>
<th>Write Timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>888</td>
<td>∞</td>
</tr>
<tr>
<td>B</td>
<td>999</td>
<td>∞</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Record</th>
<th>Value</th>
<th>Write Timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>123</td>
<td>10000</td>
</tr>
<tr>
<td>B</td>
<td>456</td>
<td>10000</td>
</tr>
</tbody>
</table>

CMU 15-721 (Spring 2018)
**Optimistic Concurrency Control**

**Txn #1**

1. **BEGIN**
   - Read A

2. **WRITE(A)**

3. **WRITE(B)**

4. **VALIDATE PHASE**

5. **WRITE PHASE**

6. **COMMIT**

**Workspace**

<table>
<thead>
<tr>
<th>Record</th>
<th>Value</th>
<th>Write Timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>888</td>
<td>∞</td>
</tr>
<tr>
<td>B</td>
<td>999</td>
<td>∞</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Record</th>
<th>Value</th>
<th>Write Timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>123</td>
<td>10000</td>
</tr>
<tr>
<td>B</td>
<td>456</td>
<td>10000</td>
</tr>
</tbody>
</table>
OPTIMISTIC CONCURRENCY CONTROL

** Txn #1 **

<table>
<thead>
<tr>
<th>BEGIN</th>
<th>READ(A)</th>
<th>WRITE(A)</th>
<th>WRITE(B)</th>
<th>VALIDATE PHASE</th>
<th>WRITE PHASE</th>
<th>COMMIT</th>
</tr>
</thead>
</table>

**Workspace**

<table>
<thead>
<tr>
<th>Record</th>
<th>Value</th>
<th>Write Timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>888</td>
<td>∞</td>
</tr>
<tr>
<td>B</td>
<td>999</td>
<td>∞</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Record</th>
<th>Value</th>
<th>Write Timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>123</td>
<td>10000</td>
</tr>
<tr>
<td>B</td>
<td>456</td>
<td>10000</td>
</tr>
</tbody>
</table>
### Optimistic Concurrency Control

**Txn #1**

<table>
<thead>
<tr>
<th>Record</th>
<th>Value</th>
<th>Write Timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>888</td>
<td>∞</td>
</tr>
<tr>
<td>B</td>
<td>999</td>
<td>∞</td>
</tr>
</tbody>
</table>

Workspace

<table>
<thead>
<tr>
<th>Record</th>
<th>Value</th>
<th>Write Timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>123</td>
<td>10000</td>
</tr>
<tr>
<td>B</td>
<td>456</td>
<td>10000</td>
</tr>
</tbody>
</table>

Commit
**OPTIMISTIC CONCURRENCY CONTROL**

**Txn #1**

BEGIN

- READ(A)
- WRITE(A)
- WRITE(B)
- VALIDATE PHASE
- WRITE PHASE

**Workspace**

<table>
<thead>
<tr>
<th>Record</th>
<th>Value</th>
<th>Write Timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>888</td>
<td>∞</td>
</tr>
<tr>
<td>B</td>
<td>999</td>
<td>∞</td>
</tr>
</tbody>
</table>

**Commit**

10001

<table>
<thead>
<tr>
<th>Record</th>
<th>Value</th>
<th>Write Timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>888</td>
<td>10001</td>
</tr>
<tr>
<td>B</td>
<td>999</td>
<td>10001</td>
</tr>
</tbody>
</table>
**Optimistic Concurrency Control**

**Txn #1**

<table>
<thead>
<tr>
<th>Record</th>
<th>Value</th>
<th>Write Timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>888</td>
<td>10001</td>
</tr>
<tr>
<td>B</td>
<td>999</td>
<td>10001</td>
</tr>
</tbody>
</table>
READ PHASE

Track the read/write sets of txns and store their writes in a private workspace.
The DBMS copies every tuple that the txn accesses from the shared database to its workspace ensure repeatable reads.
When the txn invokes COMMIT, the DBMS checks if it conflicts with other txns.

Two methods for this phase:
→ Backward Validation
→ Forward Validation
BACKWARD VALIDATION

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

TIME

Txn #1

Txn #2

Txn #3
Check whether the committing txn intersects its read/write sets with those of any txns that have already committed.
Check whether the committing txn intersects its read/write sets with those of any txns that have already committed.

Validation Scope

TIME

Txn #1

Txn #2

Txn #3
FORWARD VALIDATION

Check whether the committingtxn intersects its read/write sets with any active txns that have **not** yet committed.

---

**Diagram:**

- **Txn #1**
- **Txn #2**
- **Txn #3**
FORWARD VALIDATION

Check whether the committing txn intersects its read/write sets with any active txns that have **not** yet committed.

![Diagram showing the timeline of transactions](image)
Check whether the committing txn intersects its read/write sets with any active txns that have **not** yet committed.
VALIDATION PHASE

Original OCC uses serial validation. Parallel validation means that each txn must check the read/write sets of other txns that are trying to validate at the same time.

→ Each txn has to acquire locks for its write set records in some **global order**.

→ The txn does not need locks for read set records.
The DBMS propagates the changes in the txn’s write set to the database and makes them visible to other txns.

As each record is updated, the txn releases the lock acquired during the Validation Phase.
Mutex
→ Worst option. Mutexes are the "Hitler of Concurrency".

Atomic Addition
→ Requires cache invalidation on write.

Batched Atomic Addition
→ Needs a back-off mechanism to prevent fast burn.

Hardware Clock
→ Not sure if it will exist in future CPUs.

Hardware Counter
→ Not implemented in existing CPUs.
TIMESTAMP ALLOCATION

Throughput (Million ts/s)

Number of Cores

- Clock
- Hardware
- Atomic batch=16
- Atomic batch=8
- Atomic
- Mutex

STARING INTO THE ABYSS: AN EVALUATION OF CONCURRENCY CONTROL WITH ONE THOUSAND CORES
VLDB 2014
MODERN OCC

Harvard/MIT Silo
MIT/CMU TicToc
Single-node, in-memory OLTP DBMS.
→ Serializable OCC with parallel backward validation.
→ Stored procedure-only API.

No writes to shared-memory for read txns.
Batched timestamp allocation using **epochs**.

Pure awesomeness from **Eddie Kohler**.
Single-node, in-memory OLTP DBMS.

→ Serializable OCC with parallel backward validation.

→ Stored procedure-only API.

No writes to shared memory for read txns.

Batched timestamp allocation using epochs.

Pure awesomeness from Eddie Kohler.
SILO: EPOCHS

Time is sliced into fixed-length epochs (40ms).
All txns that start in the same epoch will be committed together at the end of the epoch.
→ Txns that span an epoch have to refresh themselves to be carried over into the next epoch.

Worker threads only need to synchronize at the beginning of each epoch.
Each worker thread generates a unique txn id based on the current epoch number and the next value in its assigned batch.
Each worker thread generates a unique txn id based on the current epoch number and the next value in its assigned batch.
Each worker thread generates a unique txn id based on the current epoch number and the next value in its assigned batch.
### SILO: COMMIT PROTOCOL

<table>
<thead>
<tr>
<th>TID Word</th>
<th>ATTR1</th>
<th>ATTR2</th>
</tr>
</thead>
<tbody>
<tr>
<td>#-####-#</td>
<td>John</td>
<td>$100</td>
</tr>
<tr>
<td>#-#####-#</td>
<td>Tupac</td>
<td>$999</td>
</tr>
<tr>
<td>#-#####-#</td>
<td>Wiz</td>
<td>$67</td>
</tr>
<tr>
<td>#-#####-#</td>
<td>O.D.B.</td>
<td>$13</td>
</tr>
</tbody>
</table>

- **Write Lock Bit**
- **Latest Version Bit**
- **Absent Bit**
**Workspace**

<table>
<thead>
<tr>
<th>Read Set</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>#-#####-#</td>
<td>O.D. B.</td>
<td>$13</td>
<td></td>
</tr>
<tr>
<td>#-#####-#</td>
<td>Tupac</td>
<td>$999</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Write Set</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Tupac</td>
<td>$777</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TID Word</th>
<th>ATTR1</th>
<th>ATTR2</th>
</tr>
</thead>
<tbody>
<tr>
<td>#-#####-#</td>
<td>John</td>
<td>$100</td>
</tr>
<tr>
<td>#-#####-#</td>
<td>Tupac</td>
<td>$999</td>
</tr>
<tr>
<td>#-#####-#</td>
<td>Wiz</td>
<td>$67</td>
</tr>
<tr>
<td>#-#####-#</td>
<td>O.D. B.</td>
<td>$13</td>
</tr>
</tbody>
</table>
**SILO: COMMIT PROTOCOL**

**Workspace**

<table>
<thead>
<tr>
<th>Read Set</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>#-####-#</td>
<td>O.D.B.</td>
<td>$13</td>
</tr>
<tr>
<td>#-####-#</td>
<td>Tupac</td>
<td>$999</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Write Set</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Tupac</td>
<td>$777</td>
</tr>
</tbody>
</table>

**Step #1: Lock Write Set**

<table>
<thead>
<tr>
<th>TID Word</th>
<th>ATTR1</th>
<th>ATTR2</th>
</tr>
</thead>
<tbody>
<tr>
<td>#-####-#</td>
<td>John</td>
<td>$100</td>
</tr>
<tr>
<td>#-####-#</td>
<td>Tupac</td>
<td>$999</td>
</tr>
<tr>
<td>#-####-#</td>
<td>Wiz</td>
<td>$67</td>
</tr>
<tr>
<td>#-####-#</td>
<td>O.D.B.</td>
<td>$13</td>
</tr>
</tbody>
</table>
### SILO: COMMIT PROTOCOL

**Workspace**

<table>
<thead>
<tr>
<th>Read Set</th>
<th>TID Word</th>
<th>ATTR1</th>
<th>ATTR2</th>
</tr>
</thead>
<tbody>
<tr>
<td>#-####-#</td>
<td>O.D.B.</td>
<td>$13</td>
<td></td>
</tr>
<tr>
<td>#-####-#</td>
<td>Tupac</td>
<td>$999</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Write Set</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Tupac</td>
<td></td>
<td></td>
<td>$777</td>
</tr>
</tbody>
</table>

**Step #1:** Lock Write Set

**Step #2:** Examine Read Set
SILO: COMMIT PROTOCOL

Workspace

Read Set

| #---###-# | O.D.B. | $13 |
| #---###-# | Tupac  | $999 |

Write Set

| Tupac     | $777 |

TID Word

<table>
<thead>
<tr>
<th>TID Word</th>
<th>ATTR1</th>
<th>ATTR2</th>
</tr>
</thead>
<tbody>
<tr>
<td>#---###-#</td>
<td>John</td>
<td>$100</td>
</tr>
<tr>
<td>#---###-#</td>
<td>Tupac</td>
<td>$999</td>
</tr>
<tr>
<td>#---###-#</td>
<td>Wiz</td>
<td>$67</td>
</tr>
<tr>
<td>#---###-#</td>
<td>O.D.B.</td>
<td>$13</td>
</tr>
</tbody>
</table>

Step #1: Lock Write Set

Step #2: Examine Read Set
### SILO: COMMIT PROTOCOL

**Workspace**

<table>
<thead>
<tr>
<th>Read Set</th>
<th>Write Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>#-####-#</td>
<td>Tupac</td>
</tr>
<tr>
<td>O.D.B.</td>
<td>$777</td>
</tr>
<tr>
<td>Tupac</td>
<td>$999</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TID Word</th>
<th>ATTR1</th>
<th>ATTR2</th>
</tr>
</thead>
<tbody>
<tr>
<td>#-####-#</td>
<td>John</td>
<td>$100</td>
</tr>
<tr>
<td>Tupac</td>
<td></td>
<td>$999</td>
</tr>
<tr>
<td>Wiz</td>
<td></td>
<td>$67</td>
</tr>
<tr>
<td>O.D.B.</td>
<td></td>
<td>$13</td>
</tr>
</tbody>
</table>

**Step #1:** Lock Write Set

**Step #2:** Examine Read Set
SILO: COMMIT PROTOCOL

Workspace

<table>
<thead>
<tr>
<th>Read Set</th>
<th>Write Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>#-#####-#</td>
<td>O.D.B.</td>
</tr>
<tr>
<td>#-#####-#</td>
<td>Tupac</td>
</tr>
<tr>
<td></td>
<td>Tupac</td>
</tr>
</tbody>
</table>

Step #1: Lock Write Set

Step #2: Examine Read Set
**SILO: COMMIT PROTOCOL**

**Workspace**

<table>
<thead>
<tr>
<th>Read Set</th>
<th>Write Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>O.D.B. $13</td>
<td>Tupac $777</td>
</tr>
<tr>
<td>Tupac $999</td>
<td></td>
</tr>
</tbody>
</table>

**Step #1:** Lock Write Set

**Step #2:** Examine Read Set

**Step #3:** Install Write Set
Cooperative threads GC.
Each worker thread marks a deleted object with a **reclamation epoch**.
→ This is the epoch after which no thread could access the object again, and thus can be safely removed.
→ Object references are maintained in thread-local storage to avoid unnecessary data movement.
DBMS handles phantoms by tracking the txn’s scan set (node set) on indexes.
→ Re-execute scans in the validation phase to see whether the index has changed.
→ Have to include “virtual” entries for keys that do not exist in the index.

We will discuss key-range and index gap locking next week...
SILO: PERFORMANCE

Database: TPC-C with 28 Warehouses
Processor: 4 sockets, 8 cores per socket

Source: Eddie Kohler
Serializable OCC implemented in DBx1000.
→ Parallel backward validation
→ Stored procedure-only API

No global timestamp allocation. 
Txn timestamps are derived from records.
TICTOC: RECORD TIMESTAMPs

**Write Timestamp (W-TS):**
→ The logical timestamp of the last committed txn that wrote to the record.

**Read Timestamp (R-TS):**
→ The logical timestamp of the last txn that read the record.

A record is considered valid from W-TS to R-TS
TICTOC: VALIDATION PHASE

Txn

WRITE(A)

READ(B)

READ(C)

LOGICAL TIME

1  2  3  4

W-\text{TS}

R-\text{TS}
TICTOC: VALIDATION PHASE

Step #1: Lock Write Set

LOGICAL TIME

WRITE(A)
READ(B)
READ(C)
TICTOC: VALIDATION PHASE

Step #1: Lock Write Set
Step #2: Compute CommitTS
TICTOC: VALIDATION PHASE

**Txn**

- **WRITE(A)**
- **READ(B)**
- **READ(C)**

**CommitTS**

**Step #1**: Lock Write Set

**Step #2**: Compute CommitTS

**LOGICAL TIME**

1 2 3 4
TICTOC: VALIDATION PHASE

**Txn**

1. **LOCK WRITE SET**
   - WRITE(A)
2. **COMPUTE COMMITTS**
   - READ(B)
3. **VALIDATE READ SET**
   - READ(C)

**LOGICAL TIME**

Step #1: Lock Write Set
Step #2: Compute CommitTS
Step #3: Validate Read Set
TICTOC: VALIDATION PHASE

**Txn**

- **Step #1:** Lock Write Set
- **Step #2:** Compute CommitTS
- **Step #3:** Validate Read Set
**TICTOC: VALIDATION PHASE**

Step #1: Lock Write Set
Step #2: Compute CommitTS
Step #3: Validate Read Set

**LOGICAL TIME**

1  2  3  4

WRITE(A)
READ(B)
READ(C)
TICTOC: VALIDATION PHASE

**Step #1:** Lock Write Set

**Step #2:** Compute CommitTS

**Step #3:** Validate Read Set

**Case 1:** Latest Version

LOGICAL TIME
TICTOC: VALIDATION PHASE

Step #1: Lock Write Set
Step #2: Compute CommitTS
Step #3: Validate Read Set
   Case 1: Latest Version
   Case 2: Updated Before CommitTS
**TICTOC: VALIDATION PHASE**

**Txn**

- WRITE(A)
- READ(B)
- READ(C)

**CommitTS**

**LOGICAL TIME**

1 2 3 4

- **Step #1:** Lock Write Set
- **Step #2:** Compute CommitTS
- **Step #3:** Validate Read Set

**Case 1:** Latest Version
**Case 2:** Updated Before CommitTS
**Case 3:** Updated After CommitTS
**TICTOC: PERFORMANCE**

*Database: 10GB YCSB*

*Processor: 4 sockets, 10 cores per socket*

Throughput (txn/sec) vs. Thread Count for different systems:

- **TICTOC**
- **HEKATON**
- **DL_DETECT**
- **NO_WAIT**
- **SILO**

**Medium Contention**
- 90% Reads / 10% Writes

**High Contention**
- 50% Reads / 50% Writes
PARTING THOUGHTS

Trade-off between aborting txns early or later.
→ **Early**: Avoid wasted work for txns that will eventually abort, but has checking overhead.
→ **Later**: No runtime overhead but lots of wasted work under high contention.

Silo is a very influential system.
NEXT CLASS

Multi-Version Concurrency Control
**Multi-Version Concurrency Control**

**Summary Comments**

Dear Authors,

Thank you for your submission to PVLDB Vol 10.

We have now received the reviews for your manuscript as an "Experiments and Analyses Papers" paper from the Review Board. While the reviewers appreciate your research results, they have given a substantial amount of comments for your revision (enclosed).

We encourage you to revise your paper taking into consideration of the reviewer comments, and submit an improved version of the manuscript in due course.

 Regards,

Associate Editor

- Remove "This is the Best Paper Ever" from the title and revise it to be scientific and reflect the experimental nature of the work.

from design issues in the classification to make the taxonomy more general.