Today’s Papers

   Bernstein & Goodman

   Samaras, Britton, Citron, Mohan
Executive Summary (Bernstein et al.)

1. **Problem**: When multiple users issue concurrent transactions to a distributed database, they experience interference.

2. **Solution**: Concurrency Control
   - Resolves interference to preserve correctness
   - Provides the illusion that each transaction is running on a dedicated database system
Executive Summary (Bernstein et al.)

3. **Goal**: Survey the state-of-the-art in concurrency control algorithms using a standardized set of terminology and assumptions
   - Algorithm #1: Two-Phase Locking
   - Algorithm #2: Timestamped Ordering
Outline

• Distributed Database Systems

• Correctness: Serializability

• Two-Phase Locking

• Timestamp Ordering
Centralized Databases

- **Transaction Manager (TM)**
  - supervises interactions between users

- **Data Manager (DM)**
  - supervises the actual database
Parallelism is a double-edged sword

- Pro: higher performance
- Con: interference between users
What can go wrong?

Alice

READ $0
ADD $10

Bob

READ $0
ADD $20

Lost Update Anomaly
What can go wrong? (cont’d)

Checking

Alice

$10

READ $10
SUB $10

Savings

Bob

$0

READ $0

$0

READ $0
ADD $10

$10

READ $10

Inconsistent Retrieval Anomaly
Four Desirable Properties

- **Atomicity**: A transaction either commits in its entirety or does not commit at all.

- **Consistency**: A transaction does not leave the database in an illegal state.

- **Isolation**: Transactions do not interfere with each other.

- **Durability**: A committed transaction stays committed.

Referred to as **ACID** properties.
Outline

• Distributed Database Systems

• Correctness: Serializability

• Two-Phase Locking

• Timestamp Ordering
Serial Execution of Transactions

**Serial Execution**: When each transaction is executed to completion before the next one starts.
A Closer Look at Serial Execution

T1
- READ(X)
- WRITE(Y)

x, y

T2
- READ(Y)
- WRITE(Z)

y, z

T3
- READ(Z)
- WRITE(X)

z
A Closer Look at Serial Execution

T1
- READ (X)
- WRITE (Y)

T2
- READ (Y)
- WRITE (Z)

T3
- READ (Z)
- WRITE (X)

R1 (X) x, y

y, z

Z
A Closer Look at Serial Execution

T1

READ (X)
WRITE (Y)

R1 (X) x, y
W1 (Y) y, z

T2

READ (Y)
WRITE (Z)

W1 (Y) y, z

T3

READ (Z)
WRITE (X)

Z
A Closer Look at Serial Execution

T1
READ (X)
WRITE (Y)

T2
READ (Y)
WRITE (Z)

T3
READ (Z)
WRITE (X)

R1 (X)
W1 (Y)
R2 (Y)
W1 (Y)
A Closer Look at Serial Execution

T1
- READ(X)
- WRITE(Y)
- R1(X)
- W1(Y)
- x, y

T2
- READ(Y)
- WRITE(Z)
- W1(Y)
- W2(Z)
- y, z

T3
- READ(Z)
- WRITE(X)
- W2(Z)
- z
A Closer Look at Serial Execution

T1
- READ(X)
- WRITE(Y)

T2
- READ(Y)
- WRITE(Z)

T3
- READ(Z)
- WRITE(X)

R1(X)
W1(Y)
R2(Y)

W1(Y)
W2(Z)

W2(Z)
R3(Z)
A Closer Look at Serial Execution

T1
- READ (X)
- WRITE (Y)

T2
- READ (Y)
- WRITE (Z)

T3
- READ (Z)
- WRITE (X)

R1 (X) w1 (Y) x, y
R2 (Y) w3 (X)
W1 (Y) w2 (Z) y, z
W2 (Z) Z
R3 (Z)
A Closer Look at Serial Execution

T1: READ(X) WRITE(Y)
T2: READ(Y) WRITE(Z)
T3: READ(Z) WRITE(X)

R1(X) W1(Y) R2(Y) W3(X)

T1: READ(X) WRITE(Y)
T2: READ(Y) WRITE(Z)
T3: READ(Z) WRITE(X)

x, y
y, z
z
Serial or Not?

R2(X) W2(Y) R3(Y) W1(X) W2(Y) W3(Z) W3(Z) R1(Z)

R1(X) W2(Y) R3(Y) W1(X) W2(Y) W3(Z) W3(Z) R1(Z)

R1(X) W1(Y) R2(Y) W3(X) W2(Y) W1(Z) W3(Z) R1(Z)
Conflicting Operations

• Two operations *conflict* with each other...
  1. if they access the same data item
  2. AND one of them is a write

• Examples
  - \( R(X) \rightarrow W(X) \)
  - \( W(X) \rightarrow R(X) \)
  - \( W(X) \rightarrow W(X) \)
Conflicting Operations (cont’d)

**Total ordering of all operations**

- $R_1(X)$
- $W_1(Y)$
- $R_2(Y)$
- $W_3(X)$

**Partial ordering of conflicting operations**

- $R_1(X) \rightarrow W_3(X)$
- $W_1(Y) \rightarrow R_2(Y)$
- $W_2(Z) \rightarrow R_3(Z)$
Computational Equivalence

\[ R_1(X) \quad W_1(Y) \quad R_2(Y) \quad W_3(X) \]

\[ R_1(X) \quad W_3(X) \quad W_1(Y) \quad R_2(Y) \]

\[ R_1(X) \quad W_1(Y) \quad W_3(X) \quad R_2(Y) \]
Serializability

R1(X)  W1(Y)  R2(Y)  W3(X)

x, y

R1(X)  W3(X)  W1(Y)  R2(Y)

x, y

R1(X)  W1(Y)  W3(X)  R2(Y)

x, y
Serializability (cont’d)

If computationally equivalent... -> Serializable

T1 T2 T3

T1 T2 T3

time
Serializable or Not?

- R1(X) W1(X) R3(X)
- W2(X)
- W1(Z) R2(Z)

- R1(X) W1(X) R3(X)
- W2(X)
- W2(Z) R1(Z)

- R1(X) W1(X) R3(X)
- W2(X)
- W1(Z) R2(Z)

Green check marks indicate serializable operations, while red x's indicate non-serializable operations.
Outline

• Distributed Database Systems
• Correctness: Serializability
  • Two-Phase Locking
• Timestamp Ordering
Locks: Exclusive Access to Data

T1
READ(X)
WRITE(Y)

T2
WRITE(Y)
WRITE(X)

Serial Execution
T1 T2 time
Locks (cont’d)

T1
READ (X)
WRITE (Y)
...

T2
WRITE (Y) WRITE (X)

Serial Execution

T2 → T1 → time
Locks (cont’d)

T1
READ(X)
WRITE(Y)

T2
WRITE(Y)
WRITE(X)

Serial Execution  Deadlock  time

Deadlock
A Closer Look at Locks

• Two types of locks
  – ReadLocks: $\mathbb{R}(x)$
    ➡️ Required before a read access
  – WriteLocks: $\mathbb{W}(x)$
    ➡️ Required before a write access

• Two locks *conflict* with each other...
  1. if they are on the same data item
  2. AND one of them is a write
A Closer Look at Locks (cont’d)

• Lock Ownership Rules

1. Different transactions cannot have conflicting locks
   ➔ Exclusive access

2. Once a transaction relinquishes a lock, it cannot obtain additional locks
   ➔ Obtain, obtain, relinquish, relinquish
Two Phases

number of locks

time

locked point

growing phase shrinking phase

T1
Two Phases (cont’d)

number of locks

locked point

T1

T2

T3

time
Two Phases (cont’d)

Ordering of locked points determine the serialization order
Basic Lock Scheduler

Scheduler

Waiting Queue

Lock-Owner Table

<table>
<thead>
<tr>
<th>Item</th>
<th>RLock</th>
<th>WLock</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>–</td>
<td>T2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Variant: Primary Copy

Duplicate copies of $x$

Duplicate locks on $x$
Variant: Primary Copy

**Primary Copy:**
Resolves all locks on $x$

**Duplicate copies of $x$:**
Variant: Centralized

Distributed Schedulers
Variant: Centralized

Centralized Scheduler
When Two-Phase Locking Fails

waiting for $T_2$ to release lock on $y$

waiting for $T_1$ to release lock on $x$

waiting for $T_3$ to release lock on $z$

Deadlock
Handling Deadlocks

1. **Deadlock Prevention**
   - “Pessimistic” approach
   - Abort/restart a transaction when a deadlock *might* occur

2. **Deadlock Detection**
   - “Optimistic” approach
   - Abort/restart a transaction only when a deadlock *actually* occurs
1. Deadlock Prevention

Q. Which to abort?
   - Preemptive: $T_1$
   - Non-preemptive: $T_2$

Q. When to abort?
   - Desc. Priority: $T_1 > T_2$
   - Asc. Priority: $T_1 < T_2$

Note: An aborted transaction is transparently restarted by the TM; the user remains oblivious of the fact
1. Deadlock Prevention (cont’d)

Scheme #1: “Wound-Wait”
   – Preemptive
   – Descending Priority

Scheme #2: “Wait-Die” Scheme
   – Non-Preemptive
   – Ascending Priority
2. Deadlock Detection

1. Detect cycles
2. Abort victim
2. Deadlock Detection (cont’d)

• Requires global view of database
  – Local views are insufficient to detect cycles
  – Local views must be merged periodically
  – Constructing a global view is expensive

• Takes a long time to detect deadlocks
  – Depends on how often merges are performed
  – Typical interval: 100s of milliseconds
Outline

• Distributed Database Systems

• Correctness: Serializability

• Two-Phase Locking

• Timestamp Ordering
 Timestamps

- Each transaction assigned a unique *timestamp* – e.g., POSIX time concatenated with TM identifier

- All conflicting operations are ordered with respect to the timestamp of their transaction
Timestamps (cont’d)

\[ T_1 \]  \quad \text{Timestamp:} T

\[ T_2 \]  \quad \text{Timestamp:} T+1

Serial Execution \quad T_1 \quad T_2 \quad \text{time}
Timestamp Scheduler

**Scheduler**

**Timestamp Table**

<table>
<thead>
<tr>
<th>Item</th>
<th>RTS</th>
<th>WTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>T</td>
<td>T+1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

*Stores the largest timestamp operation that accessed an item*
Timestamp Scheduler (cont’d)

• **READ Operation**

  if (TS < WTS(x))
  abort();
  else
  RTS(x) = max(TS, RTS(x));

• If a transaction is aborted, it is assigned a newer/larger timestamp, then restarted
Timestamp Scheduler (cont’d)

• **WRITE Operation**

  ```
  if (TS < RTS(x) || TS < WTS(X))
      abort();
  else
      WTS(x) = max(TS,WTS(x));
  ```

• If a transaction is aborted, it is assigned a newer/larger timestamp, then restarted
Variant: Multiversion

\[
\begin{array}{|c|c|}
\hline
\text{RTS} & \text{versions} \\
\hline
T & \text{WTS} \\
T+2 & T-1 \quad 0 \\
\ldots & T+1 \quad 1 \\
\ldots & \ldots \\
\hline
\end{array}
\]
Variant: Multiversion (cont’d)

• **READ Operation**
  1. Read the *version* of \( x \) with largest timestamp less than TS
  2. Add operation to *set*

• Reads are never aborted
Variant: Multiversion (cont’d)

• **WRITE** Operation
  – Find the *version* of *x* that has the next largest timestamp compared to TS
  – If an entry of *x* in *set* lies between these two timestamps, then abort
  – Else add operation to *version*
Half-Time
Today’s Papers

1. *Concurrency Control in Distributed Database Systems (1981).*
   Bernstein & Goodman

2. *Two-Phase Commit Optimizations and Tradeoffs in the Commercial Environment (1993).*
   Samaras, Britton, Citron, Mohan
Executive Summary (Samaras et al.)

1. **Problem**: Two-phase commit (the widely employed commit protocol) incurs high networking/logging overheads

2. **Solution**: “Make the common case fast”
   - 11 performance optimizations
   - Reduces the overheads of two-phase commit
Outline

• Two-Phase Commit (2PC)
  • Baseline 2PC
  • Optimized 2PC
### Four Desirable Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atomicity</td>
<td>A transaction either commits in its entirety or does not commit at all.</td>
</tr>
<tr>
<td>Consistency</td>
<td>A transaction does not leave the database in an illegal state.</td>
</tr>
<tr>
<td>Isolation</td>
<td>Transactions do not interfere with each other.</td>
</tr>
<tr>
<td>Durability</td>
<td>A committed transaction stays committed.</td>
</tr>
</tbody>
</table>

**Commit Protocol**

Referred to as **ACID** properties
Single-Phase Commit (1PC)

TM

WRITE (x, 1)

DM → x = 0

DM → x = 0
Single-Phase Commit (1PC)

WRITE(x, 1)

Consistent
1PC: What can go wrong?

WRITE \( x, 1 \)
1PC: What can go wrong?

WRITE(X, 1)
1PC: What can go wrong?

WRITE(x, 1)

Inconsistent

x = 1

x = 0
Two-Phase Commit (2PC)

TM: WRITE(x, 1)

DM: x=0

DM: x=1

Persistent Log

x=0

x=1
Two-Phase Commit (2PC)

TM \text{ WRITE}(x, 1)\]

Persistent Log

DM \[x = 1\] \[x = 0\]

DM \[x = 0\] \[x = 0\]
Two-Phase Commit (2PC)

- **TM**
- **WRITE**(x, 1)
- **DM**: x=1
- **Persistent Log**
- **Consistent**

1. **DM**: x=1
2. **Consistent Log**
Outline

• Two-Phase Commit (2PC)
  • Baseline 2PC
  • Optimized 2PC
A Closer Look at 2PC

1. PREPARE
   - VOTE YES
   - COMMIT
   - ACK

Log
- Prepared
- Committed
- END

Log
- END

forced
Quantifying the Cost of 2PC

Assume $N$ participants: 1 TM & $N-1$ DMs

- **Network Traffic**
  - Each (TM, DM) pair $\rightarrow$ 4 messages
  - Total Messages: $4(N-1)$

- **Log Traffic**
  - Each TM $\rightarrow$ 1 forced, 1 unforced logs
  - Each DM $\rightarrow$ 2 forced, 1 unforced logs
  - Total Forced Logs: $1 + 2(N-1) = 2N-1$
  - Total Unforced Logs: $1 + (N-1) = N$
Outline

• Two-Phase Commit (2PC)

• Baseline 2PC

• Optimized 2PC
Optimizations to Baseline 2PC

1. Read Only
2. Last Agent
3. Unsolicited Vote
4. Group Commits
5. Commit Acknowledgment
1. Read Only

Log
Committed

Log END

time

PREPARE

VOTE RO

COMMIT

ACK

Log
Prepared

Log Commited

Log END

time

Read Only
2. Last Agent

Log Prepared
Log Committed
Log END

time

VOTE YES

COMMIT

Log Committed
Log END

Last Agent
3. Unsolicited Vote

"Smart"

Log Committed

Log Prepared

Log Committed

Log END

PREPARE

VOTE YES

COMMIT

ACK

Log END

Log END
4. Group Commits

• **Forced logs = Blocking I/O**
  – Commit process must wait until writes have been completely flushed out
  – Large number of writes can queue up in the I/O system

  ➔ *Large latency penalty*

• **Idea:** “Batch” multiple writes into one
  – Reduces number of writes
5. Commit Acknowledgment

When to send back an ACK message?

1. Early Acknowledgment: “Optimistic”
   - I have committed
   - My subordinates (other DMs) may still be committing

2. Late Acknowledgment: “Pessimistic”
   - I have committed
   - My subordinates (other DMs) have committed
End
Today’s Papers

   Bernstein & Goodman

   Samaras, Britton, Citron, Mohan