Lecture #03 – Concurrency Control
Part I

TODAY’S AGENDA

Transaction Models
Concurrency Control Overview
Many-Core Evaluation
TRANSACTION DEFINITION
A txn is a sequence of actions that are executed on a shared database to perform some higher-level function.

Txns are the basic unit of change in the DBMS. No partial txns are allowed.
ACTION CLASSIFICATION

Unprotected Actions
→ These lack all of the ACID properties except for consistency. Their effects cannot be depended upon.

Protected Actions
→ These do not externalize their results before they are completely done. Fully ACID.

Real Actions
→ These affect the physical world in a way that is hard or impossible to reverse.
TRANSACTION MODELS

Flat Txns
Flat Txns + Savepoints
Chained Txns
Nested Txns
Saga Txns
Compensating Txns
FLAT TRANSACTIONS

Standard txn model that starts with **BEGIN**, followed by one or more actions, and then completed with either **COMMIT** or **ROLLBACK**.

*Txn #1*

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

*Txn #2*

- BEGIN
- READ(A)
- WRITE(B)
- ROLLBACK
LIMITATIONS OF FLAT TRANSACTIONS

The application can only rollback the entire txn (i.e., no partial rollbacks).

All of a txn’s work is lost if the DBMS fails before that txn finishes.

Each txn takes place at a single point in time
LIMITATIONS OF FLAT TRANSACTIONS

Multi-Stage Planning
→ An application needs to make multiple reservations.
→ All the reservations need to occur or none of them.

Bulk Updates
→ An application needs to update one billion records.
→ This txn could take hours to complete and therefore the DBMS is exposed to losing all of its work for any failure or conflict.
TRANSACTION SAVEPOINTS

Save the current state of processing for the txn and provide a handle for the application to refer to that savepoint.

The application can control the state of the txn through these checkpoints:

→ **ROLLBACK** – Revert all changes back to the state of the DB at the savepoint.

→ **RELEASE** – Destroys a savepoint previously defined in the txn.
**TRANSACTION SAVEPOINTS**

*Txn #1*

```
BEGIN
WRITE(A)
SAVEPOINT 1
WRITE(B)
ROLLBACK TO 1
WRITE(C)
COMMIT
```
**TRANSACTION SAVEPOINTS**

*Txn #1*

```
BEGIN
WRITE(A)
SAVEPOINT 1
WRITE(B)
ROLLBACK TO 1
WRITE(C)
COMMIT
```
**TRANSACTION SAVEPOINTS**

Txn #1

```
BEGIN
WRITE(A)
SAVEPOINT 1
WRITE(B)
ROLLBACK TO 1
WRITE(C)
COMMIT
```
**TRANSACTION SAVEPOINTS**

**Txn #1**

- BEGIN
- WRITE(A)
- SAVEPOINT 1
- WRITE(B)
- ROLLBACK TO 1
- WRITE(C)
- COMMIT

**Savepoint#1**
**Transaction Savepoints**

**Txn #1**

- `BEGIN`
- `WRITE(A)`
- `SAVEPOINT 1`
- `WRITE(B)`
- `ROLLBACK TO 1`
- `WRITE(C)`
- `COMMIT`

Savepoint #1:

- `A`
TRANSACTION SAVEPOINTS

Txn #1

BEGIN
WRITE(A)
SAVEPOINT 1
WRITE(B)
ROLLBACK TO 1
WRITE(C)
COMMIT

Savepoint#1

A
**TRANSACTION SAVEPOINTS**

**Txn #1**

- BEGIN
- WRITE(A)
- SAVEPOINT 1
- WRITE(B)
- ROLLBACK TO 1
- WRITE(C)
- COMMIT

**Savepoint#1**

A

**Savepoint#2**
**TRANSACTION SAVEPOINTS**

**Txn #1**

BEGIN  
WRITE(A)  
SAVEPOINT 1  
WRITE(B)  
ROLLBACK TO 1  
WRITE(C)  
COMMIT

**Savepoint#1**

A

**Savepoint#2**
TRANSACTION SAVEPOINTS

Txn #1

BEGIN
WRITE(A)
SAVEPOINT 1
WRITE(B)
ROLLBACK TO 1
WRITE(C)
COMMIT

Savepoint#1

A

Savepoint#2

B
**TRANSACTION SAVEPOINTS**

**Txn #1**

- BEGIN
- WRITE(A)
- SAVEPOINT 1
- WRITE(B)
- ROLLBACK TO 1
- WRITE(C)
- COMMIT

- **Savepoint #1**: A
- **Savepoint #2**: B
 TRANSACTION SAVEPOINTS

**Txn #1**

BEGIN
WRITE(A)
SAVEPOINT 1
WRITE(B)
ROLLBACK TO 1
WRITE(C)
COMMIT

Savepoint#1

A

Savepoint#2

X
TRANSACTION SAVEPOINTS

**Txn #1**

- BEGIN
- WRITE(A)
- **SAVEPOINT 1**
- WRITE(B)
- **ROLLBACK TO 1**
- WRITE(C)
- COMMIT

**Savepoint#1**

**Savepoint#2**

**Savepoint#3**
TRANSACTION SAVEPOINTS

**Txn #1**
- BEGIN
- WRITE(A)
- SAVEPOINT 1
- WRITE(B)
- ROLLBACK TO 1
- WRITE(C)
- COMMIT

**Savepoint#1**
- A

**Savepoint#2**
- X

**Savepoint#3**
**Transaction Savepoints**

*Txn #1*

- **BEGIN**
- **WRITE(A)**
- **SAVEPOINT 1**
- **WRITE(B)**
- **ROLLBACK TO 1**
- **WRITE(C)**
- **COMMIT**

**Savepoint#1**

- **A**

**Savepoint#2**

- **X**

**Savepoint#3**

- **C**
TRANSACTION SAVEPOINTS

Txn #1

BEGIN
WRITE(A)
SAVEPOINT 1
WRITE(B)
ROLLBACK TO 1
WRITE(C)
COMMIT

Savepoint#1

A

Savepoint#2

X

Savepoint#3

C
**Transaction Savepoints**

_Txn #1_

```
BEGIN
WRITE(A)
SAVEPOINT 1
WRITE(B)
SAVEPOINT 2
WRITE(C)
SAVEPOINT 3
RELEASE 2
WRITE(D)
ROLLBACK TO 3
```
**Transaction Savepoints**

**Txn #1**

- BEGIN
- WRITE(A)
- SAVEPOINT 1
- WRITE(B)
- SAVEPOINT 2
- WRITE(C)
- SAVEPOINT 3
- RELEASE 2
- WRITE(D)
- ROLLBACK TO 3

Savepoint #1

ROLLBACK TO 3
**Transaction Savepoints**

*txn #1*

- **BEGIN**
- **WRITE(A)**
- **SAVEPOINT 1**
- **WRITE(B)**
- **SAVEPOINT 2**
- **WRITE(C)**
- **SAVEPOINT 3**
- **RELEASE 2**
- **WRITE(D)**
- **ROLLBACK TO 3**

**Savepoint #1**

- **A**
**Transaction Savepoints**

*Txn #1*

```
BEGIN
  WRITE(A)
  SAVEPOINT 1
  WRITE(B)
  SAVEPOINT 2
  WRITE(C)
  SAVEPOINT 3
  RELEASE 2
  WRITE(D)
  ROLLBACK TO 3
```

**Savepoints**

- **Savepoint #1**: A
- **Savepoint #2**: No change

**Actions**

- **BEGIN**: Starts the transaction
- **WRITE(A)**, **WRITE(B)**, **WRITE(C)**: Update records
- **SAVEPOINT 1**, **SAVEPOINT 2**, **SAVEPOINT 3**: Save the transaction state
- **RELEASE 2**: Rollback to the savepoint
- **WRITE(D)**, **ROLLBACK TO 3**: Final update and rollback to the specified savepoint
**Transaction Savepoints**

*Txn #1*

<table>
<thead>
<tr>
<th>BEGIN</th>
<th>WRITE(A)</th>
<th>SAVEPOINT 1</th>
<th>WRITE(B)</th>
<th>SAVEPOINT 2</th>
<th>WRITE(C)</th>
<th>SAVEPOINT 3</th>
<th>RELEASE 2</th>
<th>WRITE(D)</th>
<th>ROLLBACK TO 3</th>
</tr>
</thead>
</table>

**Savepoint#1**

- A

**Savepoint#2**

- B

Operations:

- BEGIN
- WRITE(A)
- SAVEPOINT 1
- WRITE(B)
- SAVEPOINT 2
- WRITE(C)
- SAVEPOINT 3
- RELEASE 2
- WRITE(D)
- ROLLBACK TO 3
**Transaction Savepoints**

*Txn #1*

- `BEGIN`
- `WRITE(A)`
- `SAVEPOINT 1`
- `WRITE(B)`
- `SAVEPOINT 2`
- `WRITE(C)`
- `SAVEPOINT 3`
- `RELEASE 2`
- `WRITE(D)`
- `ROLLBACK TO 3`
**TRANSACTION SAVEPOINTS**

Transaction #1:

- **BEGIN**
- **WRITE(A)**
- **SAVEPOINT 1**
- **WRITE(B)**
- **SAVEPOINT 2**
- **WRITE(C)**
- **SAVEPOINT 3**
- **RELEASE 2**
- **WRITE(D)**
- **ROLLBACK TO 3**

**Savepoints:**
- **Savepoint#1**: A
- **Savepoint#2**: B
- **Savepoint#3**:
**TRANSACTION SAVEPOINTS**

**Txn #1**

**BEGIN**
- WRITE(A)
- SAVEPOINT 1
- WRITE(B)
- SAVEPOINT 2
- WRITE(C)
- SAVEPOINT 3
- RELEASE 2
- WRITE(D)
- ROLLBACK TO 3

**Savepoint#1**
- A

**Savepoint#2**
- B

**Savepoint#3**
- C
**TRANSACTION SAVEPOINTS**

**Txn #1**

BEGIN  
WRITE(A)  
SAVEPOINT 1  
WRITE(B)  
SAVEPOINT 2  
WRITE(C)  
SAVEPOINT 3  
RELEASE 2  
WRITE(D)  
ROLLBACK TO 3  

SAVEPOINT #1  

SAVEPOINT #2  

SAVEPOINT #3  

SAVEPOINT #4
**TRANSACTION SAVEPOINTS**

**Txn #1**

- **BEGIN**
- **WRITE(A)**
- **SAVEPOINT 1**
- **WRITE(B)**
- **SAVEPOINT 2**
- **WRITE(C)**
- **SAVEPOINT 3**
- **RELEASE 2**
- **WRITE(D)**
- **ROLLBACK TO 3**

**Savepoints**

1. **Savepoint#1**
   - A

2. **Savepoint#2**
   - B

3. **Savepoint#3**
   - C

4. **Savepoint#4**
**TRANSACTION SAVEPOINTS**

**Txn #1**

- **BEGIN**
- **WRITE(A)**
- **SAVEPOINT 1**
- **WRITE(B)**
- **SAVEPOINT 2**
- **WRITE(C)**
- **SAVEPOINT 3**
- **RELEASE 2**
- **WRITE(D)**
- **ROLLBACK TO 3**
**TRANSACTION SAVEPOINTS**

**Txn #1**

BEGIN
WRITE(A)
SAVEPOINT 1
WRITE(B)
SAVEPOINT 2
WRITE(C)
SAVEPOINT 3
RELEASE 2
WRITE(D)
ROLLBACK TO 3

**Savepoint#1**

A

**Savepoint#2**

B

**Savepoint#3**

C

**Savepoint#4**

D
**Transaction Savepoints**

Transaction #1

<table>
<thead>
<tr>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN</td>
</tr>
<tr>
<td>WRITE(A)</td>
</tr>
<tr>
<td>SAVEPOINT 1</td>
</tr>
<tr>
<td>WRITE(B)</td>
</tr>
<tr>
<td>SAVEPOINT 2</td>
</tr>
<tr>
<td>WRITE(C)</td>
</tr>
<tr>
<td>SAVEPOINT 3</td>
</tr>
<tr>
<td>RELEASE 2</td>
</tr>
<tr>
<td>WRITE(D)</td>
</tr>
<tr>
<td>ROLLBACK TO 3</td>
</tr>
</tbody>
</table>

Savepoint #1

- A

Savepoint #2

- B

Savepoint #3

- C

Savepoint #4

- D

ROLLBACK TO 3
**Transaction Savepoints**

**Txn #1**

```
BEGIN
WRITE(A)
SAVEPOINT 1
WRITE(B)
SAVEPOINT 2
WRITE(C)
SAVEPOINT 3
RELEASE 2
WRITE(D)
ROLLBACK TO 3
```
TRANSACTION SAVEPOINTS

Txn #1

BEGIN
WRITE(A)
SAVEPOINT 1
WRITE(B)
SAVEPOINT 2
WRITE(C)
SAVEPOINT 3
RELEASE 2
WRITE(D)
ROLLBACK TO 3

Savepoint#1

A

B

Savepoint#3

C

Savepoint#4

D

ROLLBACK TO 3
TRANSACTION SAVEPOINTS

**Txn #1**

1. BEGIN
2. WRITE(A)
3. SAVEPOINT 1
4. WRITE(B)
5. SAVEPOINT 2
6. WRITE(C)
7. SAVEPOINT 3
8. RELEASE 2
9. WRITE(D)
10. ROLLBACK TO 3

**Savepoint#1**
- A

**Savepoint#2**
- B

**Savepoint#3**
- C

**Savepoint#4**
- D

???
**TRANSACTION SAVEPOINTS**

*Txn #1*

- **BEGIN**
- **WRITE(A)**
- **SAVEPOINT 1**
- **WRITE(B)**
- **SAVEPOINT 2**
- **WRITE(C)**
- **SAVEPOINT 3**
- **RELEASE 2**
- **WRITE(D)**
- **ROLLBACK TO 3**

**Savepoint #1**

- **A**
- **B**
- **C**

**Savepoint #4**

- **D**

**Note:**
- The transaction is rolled back to savepoint 3, as indicated by `ROLLBACK TO 3`. After this point, the transaction state is undefined (denoted by `??`).
TRANSACTION CHAINS

Multiple txns executed one after another.
Combined **COMMIT / BEGIN** operation is atomic.
→ No other txn can change the state of the database as seen by the second txn from the time that the first txn commits and the second txn begins.

Differences with savepoints:
• **COMMIT** allows the DBMS to free locks.
• Cannot rollback previous txns in chain.
TRANSACTION CHAINS

Txn #1
BEGIN
WRITE(A)
COMMIT

Txn #2
BEGIN
WRITE(B)
COMMIT

Txn #3
BEGIN
WRITE(C)
ROLLBACK
TRANSACTION CHAINS

Txn #1
BEGIN
WRITE(A)
COMMIT

Txn #2
BEGIN
WRITE(B)
COMMIT

Txn #3
BEGIN
WRITE(C)
ROLLBACK
 TRANSACTION CHAINS

Txn #1

BEGIN
WRITE(A)
COMMIT

Txn #2

BEGIN
WRITE(B)
COMMIT

Txn #3

BEGIN
WRITE(C)
ROLLBACK
TRANSACTION CHAINS

Txn #1
BEGIN
WRITE(A)
COMMIT

Txn #2
BEGIN
WRITE(B)
COMMIT

Txn #3
BEGIN
WRITE(C)
ROLLBACK
NESTED TRANSACTIONS

Savepoints organize a transaction as a sequence of actions that can be rolled back individually.

Nested txns form a hierarchy of work.
→ The outcome of a child txn depends on the outcome of its parent txn.
NESTED TRANSACTIONS

**Txn #1**

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

  WRITE(D)
  ROLLBACK

COMMIT
```
NESTED TRANSACTIONS

**Txn #1**

BEGIN
WRITE(A)
BEGIN
BEGIN
WRITE(C)
COMMIT
WRITE(B)
BEGIN
WRITE(B)
COMMIT
WRITE(C)
COMMIT
WRITE(D)
ROLLBACK
COMMIT
NESTED TRANSACTIONS

Txn #1

```
BEGIN
WRITE(A)
BEGIN
BEGIN
WRITE(C)
COMMIT
COMMIT
WRITE(B)
BEGIN
WRITE(B)
COMMIT
BEGIN
WRITE(C)
COMMIT
WRITE(D)
ROLLBACK
COMMIT
```
NESTED TRANSACTIONS

**Txn #1**

```
BEGIN
WRITE(A)
BEGIN
BEGIN
WRITE(C)
COMMIT
COMMIT
WRITE(B)
BEGIN
WRITE(B)
BEGIN
WRITE(C)
COMMIT
COMMIT
WRITE(D)
ROLLBACK
COMMIT
```
NESTED TRANSACTIONS

Txn #1

```
BEGIN
  WRITE(A)
BEGIN
  BEGIN
    WRITE(C)
    COMMIT
  COMMIT
BEGIN
  WRITE(B)
BEGIN
  BEGIN
    WRITE(C)
    COMMIT
  COMMIT
BEGIN
  WRITE(D)
ROLLBACK
COMMIT
```
NESTED TRANSACTIONS

```
Txn #1

BEGIN
WRITE(A)
BEGIN
BEGIN
WRITE(C)
COMMIT
COMMIT
WRITE(B)
BEGIN
WRITE(C)
COMMIT
WRITE(D)
ROLLBACK
COMMIT
```
NESTED TRANSACTIONS

Txn #1

BEGIN
WRITE(A)
BEGIN
WRITE(B)
BEGIN
WRITE(C)
COMMIT
WRITE(D)
ROLLBACK
COMMIT

Sub-Txn #1.1
NESTED TRANSACTIONS

Txn #1

BEGIN
WRITE(A)
BEGIN
BEGIN
WRITE(C)
COMMIT
WRITE(B)
BEGIN
WRITE(C)
COMMIT
WRITE(D)
ROLLBACK

Sub-Txn #1.1
NESTED TRANSACTIONS

Txn #1

BEGIN
WRITE(A)
BEGIN
BEGIN
WRITE(C)
COMMIT
WRITE(B)
BEGIN
WRITE(D)
ROLLBACK
COMMIT

Sub-Txn #1.1
NESTED TRANSACTIONS

Sub-Txn #1.1

BEGIN
WRITE(B)
BEGIN
WRITE(C)
COMMIT
WRITE(D)
ROLLBACK

Txn #1

BEGIN
WRITE(A)
BEGIN
BEGIN
WRITE(C)
COMMIT
WRITE(B)
BEGIN
NESTED TRANSACTIONS

Txn #1

BEGIN
WRITE(A)
BEGIN
BEGIN
WRITE(C)
COMMIT
BEGIN
WRITE(B)
BEGIN
WRITE(C)
COMMIT
WRITE(D)
ROLLBACK

COMMIT

Sub-Txn #1.1

Sub-Txn #1.1.1
NESTED TRANSACTIONS

Txn #1

BEGIN
WRITE(A)
BEGIN
BEGIN
WRITE(C)
COMMIT
BEGIN
WRITE(B)
ROLLBACK
WRITE(D)
BEGIN
COMMIT

Sub-Txn #1.1

BEGIN
WRITE(B)
BEGIN
WRITE(D)
ROLLBACK

Sub-Txn #1.1.1

BEGIN
WRITE(C)
COMMIT
NESTED TRANSACTIONS

Txn #1

BEGIN
WRITE(A)
BEGIN
BEGIN
WRITE(C)
COMMIT
BEGIN
WRITE(B)
BEGIN
WRITE(D)
ROLLBACK
COMMIT

Sub-Txn #1.1

Sub-Txn #1.1.1

BEGIN
WRITE(C)
COMMIT
NESTED TRANSACTIONS

Txn #1

Sub-Txn #1.1

BEGIN
WRITE(A)
BEGIN
BEGIN
WRITE(C)
COMMIT
BEGIN
WRITE(B)
BEGIN
WRITE(D)
ROLLBACK
COMMIT

Sub-Txn #1.1.1

BEGIN
WRITE(C)
COMMIT
NESTED TRANSACTIONS

Txn #1

BEGIN
WRITE(A)
BEGIN
BEGIN
WRITE(C)
COMMIT
BEGIN
WRITE(B)
ROLLBACK
WRITE(D)
BEGIN
COMMIT

Sub-Txn #1.1

BEGIN
WRITE(B)
BEGIN
WRITE(D)
COMMIT

Sub-Txn #1.1.1

BEGIN
WRITE(C)
COMMIT
NESTED TRANSACTIONS

Txn #1

BEGIN
WRITE(A)
BEGIN
BEGIN
WRITE(C)
COMMIT
COMMIT
WRITE(B)
ROLLBACK
WRITE(D)

Sub-Txn #1.1

BEGIN
WRITE(B)
BEGIN
WRITE(D)
ROLLBACK

Sub-Txn #1.1.1

BEGIN
WRITE(C)
COMMIT
NESTED TRANSACTIONS

Sub-Txn #1.1

Txn #1

BEGIN
WRITE(A)
BEGIN
BEGIN
WRITE(C)
COMMIT
COMMIT
WRITE(B)
ROLLBACK
WRITE(D)
ROLLBACK

Sub-Txn #1.1.1

BEGIN
WRITE(C)
COMMIT
WRITE(D)
COMMIT
X
X
X
X
NESTED TRANSACTIONS

**Txn #1**

- BEGIN
- WRITE(A) ✓
- BEGIN

**Sub-Txn #1.1**

- BEGIN
- WRITE(B) X
- BEGIN
- WRITE(D) X
- ROLLBACK

**Sub-Txn #1.1.1**

- BEGIN
- WRITE(C) X
- COMMIT
BULK UPDATE PROBLEM

These other txn models are nice, but they still do not solve our bulk update problem.

Chained txns seems like the right idea but they require the application to handle failures and maintain it’s own state.
→ Has to be able to reverse changes when things fail.
COMPENSATING TRANSACTIONS

A special type of txn that is designed to semantically reverse the effects of another already committed txn.

Reversal has to be **logical** instead of physical.

→ Example: Decrement a counter by one instead of reverting to the original value.
A sequence of chained txns $T_1, ..., T_n$ and compensating txns $C_1, ..., C_{n-1}$ where one of the following is guaranteed:

→ The txns will commit in the order $T_1, ..., T_n$

→ The txns will commit in the order $T_1, ..., T_j, C_j, ..., C_1$ (where $j < n$)
SAGA TRANSACTIONS

Txn #1
BEGIN
WRITE(A+1)
COMMIT

Txn #2
BEGIN
WRITE(B+1)
COMMIT

Txn #3
BEGIN
WRITE(C+1)
COMMIT

Comp Txn #1
BEGIN
WRITE(A-1)
COMMIT

Comp Txn #2
BEGIN
WRITE(B-1)
COMMIT

Comp Txn #3
BEGIN
WRITE(C-1)
COMMIT
SAGA TRANSACTIONS

Txn #1
BEGIN
WRITE(A+1)
COMMIT

Txn #2
BEGIN
WRITE(B+1)
COMMIT

Txn #3
BEGIN
WRITE(C+1)
COMMIT

Comp Txn #1
BEGIN
WRITE(A-1)
COMMIT

Comp Txn #2
BEGIN
WRITE(B-1)
COMMIT

Comp Txn #3
BEGIN
WRITE(C-1)
COMMIT
SAGA TRANSACTIONS

Txn #1
BEGIN
WRITE(A+1)
COMMIT

Txn #2
BEGIN
WRITE(B+1)
COMMIT

Txn #3
BEGIN
WRITE(C+1)
COMMIT

Comp Txn #1
BEGIN
WRITE(A-1)
COMMIT

Comp Txn #2
BEGIN
WRITE(B-1)
COMMIT

Comp Txn #3
BEGIN
WRITE(C-1)
COMMIT
SAGA TRANSACTIONS

Txn #1
BEGIN
WRITE(A+1)
COMMIT

Txn #2
BEGIN
WRITE(B+1)
COMMIT

Txn #3
BEGIN
WRITE(C+1)
COMMIT

Comp Txn #1
BEGIN
WRITE(A-1)
COMMIT

Comp Txn #2
BEGIN
WRITE(B-1)
COMMIT

Comp Txn #3
BEGIN
WRITE(C-1)
COMMIT
SAGA TRANSACTIONS

Txn #1
BEGIN
WRITE(A+1)
COMMIT

Txn #2
BEGIN
WRITE(B+1)
COMMIT

Txn #3
BEGIN
WRITE(C+1)

Comp Txn #1
BEGIN
WRITE(A-1)
COMMIT

Comp Txn #2
BEGIN
WRITE(B-1)
COMMIT

Comp Txn #3
BEGIN
WRITE(C-1)
COMMIT

SAQA TRANSACTIONS

Txn #1
BEGIN
WRITE(A+1)
COMMIT

Txn #2
BEGIN
WRITE(B+1)
COMMIT

Txn #3
BEGIN
WRITE(C+1)
COMMIT

Comp Txn #1
BEGIN
WRITE(A-1)
COMMIT

Comp Txn #2
BEGIN
WRITE(B-1)
COMMIT

Comp Txn #3
BEGIN
WRITE(C-1)
COMMIT
SAGA TRANSACTIONS

Txn #1
BEGIN
WRITE(A+1)
COMMIT

Txn #2
BEGIN
WRITE(B+1)
COMMIT

Txn #3
BEGIN
WRITE(C+1)

Comp Txn #1
BEGIN
WRITE(A-1)
COMMIT

Comp Txn #2
BEGIN
WRITE(B-1)
COMMIT

Comp Txn #3
BEGIN
WRITE(C-1)
COMMIT
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
TXN INTERNAL STATE

**Undo Log Entries**
→ Stored in an in-memory data structure.
→ Dropped on commit.

**Redo Log Entries**
→ Append to the in-memory tail of WAL.
→ Flushed to disk on commit.

**Read/Write Set**
→ Depends on the concurrency control scheme.
CONCURRENCY CONTROL SCHEMES

**Two-Phase Locking (2PL)**
→ Assume txns will conflict so they must acquire locks on elements before they are allowed to access them.

**Timestamp Ordering (T/O)**
→ Assume that conflicts are rare so txns do not need to acquire locks and instead check for conflicts at commit time.
TWO-PHASE LOCKING

**Txn #1**

<table>
<thead>
<tr>
<th>BEGIN</th>
<th>LOCK(A)</th>
<th>READ(A)</th>
<th>LOCK(B)</th>
<th>WRITE(B)</th>
<th>UNLOCK(A)</th>
<th>UNLOCK(B)</th>
<th>COMMIT</th>
</tr>
</thead>
</table>
TWO-PHASE LOCKING

Txn #1

BEGIN

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

**Txn #1**

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

Txn #1

Growing Phase
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
## TWO-PHASE LOCKING

### Txn #1

<table>
<thead>
<tr>
<th>BEGIN</th>
<th>LOCK(A)</th>
<th>READ(A)</th>
<th>LOCK(B)</th>
<th>WRITE(B)</th>
<th>UNLOCK(A)</th>
<th>UNLOCK(B)</th>
<th>COMMIT</th>
</tr>
</thead>
</table>

### Txn #2

<table>
<thead>
<tr>
<th>BEGIN</th>
<th>LOCK(B)</th>
<th>WRITE(B)</th>
<th>LOCK(A)</th>
<th>WRITE(A)</th>
<th>UNLOCK(A)</th>
<th>UNLOCK(B)</th>
<th>COMMIT</th>
</tr>
</thead>
</table>
TWO-PHASE LOCKING

**Txn #1**

<table>
<thead>
<tr>
<th>BEGIN</th>
<th>LOCK(A)</th>
<th>READ(A)</th>
<th>LOCK(B)</th>
<th>WRITE(B)</th>
<th>UNLOCK(A)</th>
<th>UNLOCK(B)</th>
<th>COMMIT</th>
</tr>
</thead>
</table>

**Txn #2**

<table>
<thead>
<tr>
<th>BEGIN</th>
<th>LOCK(B)</th>
<th>WRITE(B)</th>
<th>LOCK(A)</th>
<th>WRITE(A)</th>
<th>UNLOCK(A)</th>
<th>UNLOCK(B)</th>
<th>COMMIT</th>
</tr>
</thead>
</table>
TWO-PHASE LOCKING

**Txn #1**

<table>
<thead>
<tr>
<th>BEGIN</th>
<th>LOCK(A)</th>
<th>READ(A)</th>
<th>LOCK(B)</th>
<th>WRITE(B)</th>
<th>UNLOCK(A)</th>
<th>UNLOCK(B)</th>
<th>COMMIT</th>
</tr>
</thead>
</table>

**Txn #2**

<table>
<thead>
<tr>
<th>BEGIN</th>
<th>LOCK(B)</th>
<th>WRITE(B)</th>
<th>LOCK(A)</th>
<th>WRITE(A)</th>
<th>UNLOCK(A)</th>
<th>UNLOCK(B)</th>
<th>COMMIT</th>
</tr>
</thead>
</table>
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**

<table>
<thead>
<tr>
<th>Begin</th>
<th>Lock(A)</th>
<th>Read(A)</th>
<th>Lock(B)</th>
<th>Write(B)</th>
<th>Unlock(A)</th>
<th>Unlock(B)</th>
<th>Commit</th>
</tr>
</thead>
</table>

**Txn #2**

<table>
<thead>
<tr>
<th>Begin</th>
<th>Lock(B)</th>
<th>Write(B)</th>
<th>Lock(A)</th>
<th>Write(A)</th>
<th>Unlock(A)</th>
<th>Unlock(B)</th>
<th>Commit</th>
</tr>
</thead>
</table>
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)
- 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 othertxn.
TIMESTAMP ORDERING

Txn #1

BEGIN

READ(A)

WRITE(B)

• • • •  • • •

WRITE(A)

COMMIT
TIMESTAMP ORDERING

#1

BEGIN
READ(A)
WRITE(B)

• • • •  • • •

WRITE(A)

COMMIT
TIMESTAMP ORDERING

#1

BEGIN
READ(A)

WRITE(B)

... ... ...

WRITE(A)

COMMIT
**TIMESTAMP ORDERING**

<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

<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

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>10000</td>
</tr>
</tbody>
</table>

txn #1

10001
TIMESTAMP ORDERING

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

<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>

#1
TIMESTAMP ORDERING

Record | Read Timestamp | Write Timestamp
---|---|---
A | 10001 | 10000
B | 10000 | 10001
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>10001</td>
<td>10000</td>
</tr>
<tr>
<td>B</td>
<td>10000</td>
<td>10001</td>
</tr>
</tbody>
</table>
### TIMESTAMP ORDERING

<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>

**Example Transactions:**
- **BEGIN**: Transaction #1
  - **READ(A)**
  - **WRITE(B)**
- **COMMIT**
  - **READ(A)**
  - **WRITE(A)**

**Ordering Example:**
- **Record A**: Read at 10001, Write at 10000
- **Record B**: Read at 10000, Write at 10001
TIMESTAMP ORDERING

<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>10005</td>
</tr>
<tr>
<td>B</td>
<td>10000</td>
<td>10001</td>
</tr>
</tbody>
</table>

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

#1

10001
TIMESTAMP ORDERING

Records:

<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>10005</td>
</tr>
<tr>
<td>B</td>
<td>10000</td>
<td>10001</td>
</tr>
</tbody>
</table>
TIMESTAMP ORDERING

<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>10005</td>
</tr>
<tr>
<td>B</td>
<td>10000</td>
<td>10001</td>
</tr>
</tbody>
</table>

10001

#1
TIMESTAMP ORDERING

Basic T/O
→ Check for conflicts on each read/write.
→ Copy tuples on each access to ensure repeatable reads.

Multi-Version Concurrency Control (MVCC)
→ Create a new version of a tuple whenever a txn modifies it. Use timestamps as version id.
→ Check visibility on every read/write.

Optimistic Currency Control (OCC)
→ Store all changes in private workspace.
→ Check for conflicts at commit time and then merge.
# Concurrency Control Schemes

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DL_DETECT</strong></td>
<td>2PL w/ Deadlock Detection</td>
</tr>
<tr>
<td><strong>NO_WAIT</strong></td>
<td>2PL w/ Non-waiting Prevention</td>
</tr>
<tr>
<td><strong>WAIT_DIE</strong></td>
<td>2PL w/ Wait-and-Die Prevention</td>
</tr>
<tr>
<td><strong>TIMESTAMP</strong></td>
<td>Basic T/O Algorithm</td>
</tr>
<tr>
<td><strong>MVCC</strong></td>
<td>Multi-Version T/O</td>
</tr>
<tr>
<td><strong>OCC</strong></td>
<td>Optimistic Concurrency Control</td>
</tr>
</tbody>
</table>
## CONCURRENCY CONTROL SCHEMES

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DL_DETECT</td>
<td>2PL w/ Deadlock Detection</td>
</tr>
<tr>
<td>NO_WAIT</td>
<td>2PL w/ Non-waiting Prevention</td>
</tr>
<tr>
<td>WAIT_DIE</td>
<td>2PL w/ Wait-and-Die Prevention</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>Basic T/O Algorithm</td>
</tr>
<tr>
<td>MVCC</td>
<td>Multi-Version T/O</td>
</tr>
<tr>
<td>OCC</td>
<td>Optimistic Concurrency Control</td>
</tr>
</tbody>
</table>
## Concurrency Control Schemes

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DL_DETECT</td>
<td>2PL w/ Deadlock Detection</td>
</tr>
<tr>
<td>NO_WAIT</td>
<td>2PL w/ Non-waiting Prevention</td>
</tr>
<tr>
<td>WAIT_DIE</td>
<td>2PL w/ Wait-and-Die Prevention</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>Basic T/O Algorithm</td>
</tr>
<tr>
<td>MVCC</td>
<td>Multi-Version T/O</td>
</tr>
<tr>
<td>OCC</td>
<td>Optimistic Concurrency Control</td>
</tr>
</tbody>
</table>

**Database Systems**
- IBM DB2
- Microsoft SQL Server
- MySQL
- Sybase
- SQLite
- Cubrid
## CONCURRENCY CONTROL SCHEMES

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DL_DETECT</td>
<td>2PL w/ Deadlock Detection</td>
</tr>
<tr>
<td>NO_WAIT</td>
<td>2PL w/ Non-waiting Prevention</td>
</tr>
<tr>
<td>WAIT_DIE</td>
<td>2PL w/ Wait-and-Die Prevention</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>Basic T/O Algorithm</td>
</tr>
<tr>
<td>MVCC</td>
<td>Multi-Version T/O</td>
</tr>
<tr>
<td>OCC</td>
<td>Optimistic Concurrency Control</td>
</tr>
<tr>
<td>Scheme</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>DL_DETECT</td>
<td>2PL w/ Deadlock Detection</td>
</tr>
<tr>
<td>NO_WAIT</td>
<td>2PL w/ Non-waiting Prevention</td>
</tr>
<tr>
<td>WAIT_DIE</td>
<td>2PL w/ Wait-and-Die Prevention</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>Basic T/O Algorithm</td>
</tr>
<tr>
<td>MVCC</td>
<td>Multi-Version T/O</td>
</tr>
<tr>
<td>OCC</td>
<td>Optimistic Concurrency Control</td>
</tr>
</tbody>
</table>
# Concurrency Control Schemes

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Algorithm/Prevention</th>
</tr>
</thead>
<tbody>
<tr>
<td>DL_DETECT</td>
<td>2PL w/ Deadlock Detection</td>
</tr>
<tr>
<td>NO_WAIT</td>
<td>2PL w/ Non-waiting Prevention</td>
</tr>
<tr>
<td>WAIT_DIE</td>
<td>2PL w/ Wait-and-Die Prevention</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>Basic T/O Algorithm</td>
</tr>
<tr>
<td>MVCC</td>
<td>Multi-Version T/O</td>
</tr>
<tr>
<td>OCC</td>
<td>Optimistic Concurrency Control</td>
</tr>
</tbody>
</table>
## CONCURRENCY CONTROL SCHEMES

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DL_DETECT</td>
<td>2PL w/ Deadlock Detection</td>
</tr>
<tr>
<td>NO_WAIT</td>
<td>2PL w/ Non-waiting Prevention</td>
</tr>
<tr>
<td>WAIT_DIE</td>
<td>2PL w/ Wait-and-Die Prevention</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>Basic T/O Algorithm</td>
</tr>
<tr>
<td>MVCC</td>
<td>Multi-Version T/O</td>
</tr>
<tr>
<td>OCC</td>
<td>Optimistic Concurrency Control</td>
</tr>
</tbody>
</table>
1000-CORE CPU SIMULATOR

DBx1000 Database System
→ In-memory DBMS with pluggable lock manager.
→ No network access, logging, or concurrent indexes

MIT Graphite CPU Simulator
→ Single-socket, tile-based CPU.
→ Shared L2 cache for groups of cores.
→ Tiles communicate over 2D-mesh network.

STARING INTO THE ABYSS: AN EVALUATION OF CONCURRENCY CONTROL WITH ONE THOUSAND CORES
TARGET WORKLOAD

Yahoo! Cloud Serving Benchmark (YCSB)
→ 20 million tuples
→ Each tuple is 1KB (total database is ~20GB)
Each transactions reads/modifies 16 tuples.
Varying skew in transaction access patterns.
Serializable isolation level.
READ-ONLY WORKLOAD

![Graph showing throughput vs number of cores for different workload types including DL_DETECT, TIMESTAMP, NO_WAIT, MVCC, WAIT_DIE, and OCC.]
READ-ONLY WORKLOAD

Throughput (Million txn/s) vs Number of Cores

- DL_DETECT
- TIMESTAMP
- NO_WAIT
- MVCC
- WAIT_DIE
- OCC
READ-ONLY WORKLOAD

- **DL_DETECT**
- **TIMESTAMP**
- **NO_WAIT**
- **MVCC**
- **WAIT_DIE**
- **OCC**

Throughput (Milliontxn/s) vs Number of Cores graph.
READ-ONLY WORKLOAD

The diagram illustrates the throughput (in million transactions per second) as a function of the number of cores. The throughput is measured for different workloads and concurrency control mechanisms.

- **DL_DETECT** (red line with circle markers)
- **TIMESTAMP** (green line with triangle markers)
- **NO_WAIT** (orange line with diamond markers)
- **MVCC** (blue line with square markers)
- **WAIT_DIE** (yellow line with square markers)
- **OCC** (purple line with triangle markers)

The x-axis represents the number of cores, ranging from 0 to 1000, and the y-axis represents throughput in million transactions per second, ranging from 0 to 14.

- The DL_DETECT workload shows the highest throughput across all core counts.
- WAIT_DIE and OCC have the lowest throughput.

The diagram provides a clear comparison of how different concurrency control mechanisms affect the performance of the read-only workload under varying core counts.
WRITE-INTENSIVE / MEDIUM-CONTENTION

The graph shows the throughput (in million transactions per second) as a function of the number of cores. Different lines represent different protocols or policies, such as DL_DETECT, TIMESTAMP, NO_WAIT, MVCC, WAIT_DIE, and OCC. The performance varies significantly depending on the protocol and the number of cores.
WRITE-INTENSIVE / MEDIUM-CONTENTION

Throughput (Million txn/s)

0.0 0.5 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5

Number of Cores

0 200 400 600 800 1000

DL_DETECT  TIMESTAMP  NO_WAIT  MVCC  WAIT_DIE  OCC
WRITE-INTENSIVE / MEDIUM-CONTENTION

![Graph showing throughput in million transactions per second (txn/s) vs. number of cores. The graph compares different protocols:
- DL_DETECT
- NO_WAIT
- WAIT_DIE
- TIMESTAMP
- MVCC
- OCC

The x-axis represents the number of cores ranging from 0 to 1000, and the y-axis represents throughput ranging from 0.0 to 4.5. Each protocol is plotted with distinct markers and colors, allowing for easy comparison of performance across different core counts.](image-url)
WRITE-INTENSIVE / MEDIUM-CONTENTION

Throughput (Milliontxn/s)

- DL_DETECT
- NO_WAIT
- WAIT_DIE
- TIMESTAMP
- MVCC
- OCC

Number of Cores

0 200 400 600 800 1000
WRITE-INTENSIVE / HIGH-CONTENTION

Throughput (Milliontxn/s)

Number of Cores

DL_DETECT
TIMESTAMP
NO_WAIT
MVCC
WAIT_DIE
OCC
WRITE-INTENSIVE / HIGH-CONTENTION

Throughput (Million txn/s) vs. Number of Cores

- DL_DETECT
- NO_WAIT
- WAIT_DIE
- TIMESTAMP
- MVCC
- OCC

Chart shows the throughput (in millions of transactions per second) for various protocols as the number of cores increases from 0 to 1000.
WRITE-INTENSIVE / HIGH-CONTENTION
WRITE-INTENSIVE / HIGH-CONTENTION

![Graph showing throughput vs number of cores for different database transaction strategies.]

- DL_DETECT
- NO_WAIT
- WAIT_DIE
- TIMESTAMP
- MVCC
- OCC

Throughput (Million txn/s) vs Number of Cores
WRITE-INTENSIVE / HIGH-CONTENTION
WRITE-INTENSIVE / HIGH-CONTENTION

Throughput (Million txn/s) vs Number of Cores

- DL_DETECT
- NO_WAIT
- WAIT_DIE
- TIMESTAMP
- MVCC
- OCC
WRITE-INTENSIVE / HIGH-CONTENTION

[Graph showing throughput vs. number of cores for different transaction strategies: DL_DETECT, NO_WAIT, WAIT_DIE, TIMESTAMP, MVCC, OCC.]

CMU 15-721 (Spring 2016)
WRITE-INTENSIVE / HIGH-CONTENTION

![Bar chart showing the breakdown of useful work, aborts, TS alloc., index, wait, and manager for different strategies: DL_DETECT, NO_WAIT, WAIT_DIE, TIMESTAMP, MVCC, and OCC.]}
BOTTLENECKS

**Lock Thrashing**
→ DL_DETECT, WAIT_DIE

**Timestamp Allocation**
→ All T/O algorithms + WAIT_DIE

**Memory Allocations**
→ OCC + MVCC
LOCK THRASHING

Each txn waits longer to acquire locks, causing other txn to wait longer to acquire locks.

Can measure this phenomenon by removing deadlock detection/prevention overhead.
→ Force txns to acquire locks in primary key order.
→ Deadlocks are not possible.
LOCK THRASHING

![Graph showing throughput vs. number of cores with different values of theta (0, 0.6, 0.8)]
LOCK THRASHING
LOCK THRASHING

When the number of active transactions gets too high, many transactions suddenly become blocked and begin to experience serious delays. The result is called lock thrashing, which occurs when the lock manager becomes overwhelmed by the number of active transactions. When the number of active transactions exceeds the capacity of the lock manager, each transaction that tries to acquire a lock encounters a significant delay.

This delay can lead to significant performance degradation, as shown in the graph. The graph demonstrates the relationship between the number of active transactions and the throughput. As the number of active transactions increases, the throughput decreases, indicating that the system is experiencing thrashing.

The graph shows that as the number of active transactions increases, the throughput decreases. This is because the lock manager becomes overwhelmed, leading to increased blocking times for transactions. The point at which the throughput begins to decrease is the threshold at which lock thrashing becomes significant.

To prevent lock thrashing, it is important to monitor the number of active transactions and adjust the system parameters accordingly. This can involve increasing the number of locks, optimizing queries to reduce contention, or employing more sophisticated locking mechanisms.
LOCK THRASHING

In a database system, lock contention can lead to lock conversion deadlock, because at most one transaction can have exclusive lock on each table. When two transactions attempt to acquire the lock at the same time, a deadlock occurs. To avoid this, database systems use an optimistic approach where the transaction is allowed to proceed without acquiring the lock. However, if another transaction acquires the lock, this transaction must then convert to the lock. This process can lead to a situation where many transactions may be blocked due to lock contention.

To avoid lock thrashing, database systems use algorithms to prevent or minimize lock conversions. One such algorithm is Lock Throttling. In Lock Throttling, if the number of active transactions exceeds a certain threshold, the system will not acquire new locks until the number of active transactions decreases below the threshold. This helps prevent lock thrashing and improves system performance.

![Lock Throttling Diagram](image)

**Lock Throttling**: When the number of active transactions gets too high, many transactions are blocked, leading to lock thrashing. This can be mitigated by dynamically adjusting the lock acquisition rate based on the number of active transactions.
TIMESTAMP ALLOCATION

**Mutex**
→ Worst option.

**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) vs Number of Cores

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

CMU 15-721 (Spring 2016)
MEMORY ALLOCATIONS

Copying data on every read/write access slows down the DBMS because of contention on the memory controller.

→ In-place updates and non-copying reads are not affected as much.

Default libc malloc is slow. Never use it.
PARTITION-LEVEL LOCKING

The database is split up into horizontal partitions:
→ Each partition is assigned a single-threaded execution engine that has exclusive access to its data.
→ In-place updates.

Only one txn can execute at a time per partition.
→ Order txns based on when they arrive at the DBMS.
→ A txn acquires the lock for a partition when it has the lowest timestamp.
→ It is not allowed to access any partition that it does not hold the lock for.
MULTI-PARTITION WORKLOADS

![Graph showing throughput vs. number of cores for different partitions]
Concurrency control is hard to get correct and perform well.

Evaluation did not consider HTAP workloads.
NEXT CLASS

Isolation Levels
Modern MVCC