TODAY’S AGENDA

Background
Transaction Models
Concurrency Control Protocols
Isolation Levels
This course is on database systems for modern transaction processing and analytical workloads.

The first three weeks are focused on how to ingest new data quickly. We will then discuss how to analyze that data and ask complex questions about it.
DATABASE WORKLOADS

On-Line Transaction Processing (OLTP)
→ Fast operations that only read/update a small amount of data each time.

On-Line Analytical Processing (OLAP)
→ Complex queries that read a lot of data to compute aggregates.

Hybrid Transaction + Analytical Processing
→ OLTP + OLAP together on the same database instance
BIFURCATED ENVIRONMENT

Transactions

OLTP Data Silos

OLAP Data Warehouse
BIFURCATED ENVIRONMENT

Transactions

OLTP Data Silos

Extract
Transform
Load

OLAP Data Warehouse
BIFURCATED ENVIRONMENT

Transactions

OLTP Data Silos

Extract Transform Load

Analytical Queries

OLAP Data Warehouse
BIFURCATED ENVIRONMENT

Transactions

Analytical Queries

HTAP Database

Extract
Transform
Load

OLAP Data Warehouse
WORKLOAD CHARACTERIZATION

Operation Complexity

Simple

Complex

Writes

Reads

Workload Focus

OLTP

OLAP

HTAP

Source: Michael Stonebraker
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 is the DBMS fails before that txn finishes.

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

Example #1: Multi-Stage Planning
→ An application needs to make multiple reservations.
→ All the reservations need to occur or none of them.

Example #2: 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 savepoints:

→ **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

*New Savepoint*
**TRANSACTION SAVEPOINTS**

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

**New Savepoint**
- A

New Savepoint
TRANSACTION SAVEPOINTS

Txn #1

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

Savepoint#1

A

New Savepoint
TRANSACTION SAVEPOINTS

Transaction #1:

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

Savepoint #1:

A

New Savepoint:

B
TRANSACTION SAVEPOINTS

Txn #1

<table>
<thead>
<tr>
<th>BEGIN</th>
<th>SAVEPOINT 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>WRITE(A)</td>
<td>WRITE(B)</td>
</tr>
<tr>
<td>WRITE(C)</td>
<td>ROLLBACK TO 1</td>
</tr>
<tr>
<td></td>
<td>COMMIT</td>
</tr>
</tbody>
</table>

Savepoint #1

A

New Savepoint

X

New Savepoint
**Transaction Savepoints**

**Txn #1**

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

**Savepoint #1**

- A

**New Savepoint**

- C

**New Savepoint**

- X
**TRANSACTION SAVEPOINTS**

*Txn #1*

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

**Savepoint#1**

- A

**New Savepoint**

**New Savepoint**

- 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`

New Savepoint

- `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

**Savepoint#1**

A

**New Savepoint**
** 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

- New Savepoint
**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 #1

**Savepoint #2**

Savepoint #2

**New Savepoint**

New Savepoint
**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**: C
- **New Savepoint**
**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

**New Savepoint**

- B
- 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**

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

**New Savepoint**

- **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**
- **D**

**New Savepoint**

- **???”
**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

**New Savepoint**

D
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

$\text{Txn } \#1$

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

**Sub-Txn #1.1**

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

**Txn #1**

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

**Sub-Txn #1.1**

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

**Txn #1**

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

**Txn #1**

- `BEGIN`
- `WRITE(A)`
- `BEGIN`
- `COMMIT`

**Sub-Txn #1.1**

- `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

Sub-Txn #1.1

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
COMMIT
WRITE(B)
ROLLBACK
WRITE(D)

Sub-Txn #1.1
BEGIN
WRITE(B)
BEGIN
WRITE(C)
COMMIT

Sub-Txn #1.1.1
BEGIN
WRITE(D)
ROLLBACK
NESTED TRANSACTIONS

Txn #1

- BEGIN
- WRITE(A)
- BEGIN

Sub-Txn #1.1

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

Sub-Txn #1.1.1

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

**Txn #1**

BEGIN
WRITE(A) ✓
BEGIN

**Sub-Txn #1.1**

BEGIN
WRITE(B) ✗
BEGIN
WRITE(D) ✗
ROLLBACK

**Sub-Txn #1.1.1**

BEGIN
WRITE(C) ✗
COMMIT
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 firsttxn 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
- READ(A)
- WRITE(B)

**Txn #3**
- BEGIN
- WRITE(C)
- ROLLBACK
TRANSACTION CHAINS

txn #1
BEGIN
WRITE(A)
COMMIT

txn #2
BEGIN
READ(A)
WRITE(B)
COMMIT

txn #3
BEGIN
WRITE(C)
ROLLBACK
TRANSACTION CHAINS

Txn #1
BEGIN
WRITE(A)
COMMIT

Txn #2
BEGIN
READ(A)
WRITE(B)
COMMIT

Txn #3
BEGIN
WRITE(C)
ROLLBACK

A
TRANSACTION CHAINS

Txn #1
BEGIN
WRITE(A)
COMMIT

Txn #2
BEGIN
READ(A)
WRITE(B)
COMMIT

Txn #3
BEGIN
WRITE(C)
ROLLBACK

Database:
A  B
TRANSACTION CHAINS

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

Txn #2
- BEGIN
- READ(A)
- WRITE(B)
- COMMIT

Txn #3
- BEGIN
- WRITE(C)
- ROLLBACK

A  B
TRANSACTION CHAINS

Txn #1
BEGIN
WRITE(A)
COMMIT

Txn #2
BEGIN
READ(A)
WRITE(B)
COMMIT

Txn #3
BEGIN
WRITE(C)
ROLLBACK
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 its 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.
SAGA TRANSACTIONS

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 $T_1 \ldots T_j, C_j \ldots C_1$ (where $j < n$)

This allows the DBMS to support long-running, multi-step txns without application-managed logic.
**SAGA TRANSACTIONS**

*Txn #1*
- **BEGIN**
- WRITE(A+1)
- **COMMIT**

*Txn #2*
- **BEGIN**
- WRITE(B+1)
- **COMMIT**

*Txn #3*
- **BEGIN**
- WRITE(C+1)
SAGA TRANSACTIONS

*Txn #1*
- BEGIN
- WRITE(A+1)
- COMMIT

*Txn #2*
- BEGIN
- WRITE(B+1)
- COMMIT

*Txn #3*
- BEGIN
- WRITE(C+1)
SAGA TRANSACTIONS

Txn #1
BEGIN
WRITE(A+1)
COMMIT

Txn #2
BEGIN
WRITE(B+1)
COMMIT

Txn #3
BEGIN
WRITE(C+1)

Comp Txn #2
BEGIN
WRITE(B-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
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

Status
→ The current execution state of the txn.

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

- **Shrinking Phase**
  - 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**

<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**

| 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)
- WRITE(A)
- UNLOCK(A)
- UNLOCK(B)
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)
  - 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.
TIMESTAMP ORDERING

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.
BASIC T/O

#1

BEGIN

READ(A)  WRITE(B)  WRITE(A)  ...  COMMIT
BASIC T/O

BEGIN
READ(A)
WRITE(B)

.......

WRITE(A)

COMMIT

10001

#1
BASIC T/O

BEGIN
READ(A)
WRITE(B)

WRITE(A)
COMMIT

Record | Read Timestamp | Write Timestamp
---|---|---
A | 10000 | 10000
B | 10000 | 10000
Record | Read Timestamp | Write Timestamp
--- | --- | ---
A   | 10000 | 10000
B   | 10000 | 10000
BASIC T/O

<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>
BASIC T/O

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

BEGIN
READ(A)
WRITE(B)

COMMIT
BASIC T/O

#1

BEGIN
READ(A)
WRITE(B)

COMMIT
WRITE(A)

Record | Read Timestamp | Write Timestamp
--- | --- | ---
A | 10001 | 10000
B | 10000 | 10001
BASIC T/O

10001

#1

BEGIN
READ(A)
WRITE(B)

... ...

WRITE(A)

COMMIT

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

CMU 15-721 (Spring 2019)
### BASIC T/O

**Record** | **Read Timestamp** | **Write Timestamp**
---|---|---
A | 10001 | 10005
B | 10000 | 10001
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

- **Record** | **Value** | **Write Timestamp**
  - **A** | 123 | 10000
  - **B** | 456 | 10000
Txn #1

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

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
**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**

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>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)**

**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>
**OPTIMISTIC CONCURRENCY CONTROL**

**Txn #1**

**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**

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>

<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)**

VALIDATE PHASE

WRITE PHASE

**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**

**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>888</td>
<td>10001</td>
</tr>
<tr>
<td>B</td>
<td>999</td>
<td>10001</td>
</tr>
</tbody>
</table>
OPTIMISTIC CONCURRENCY CONTROL

Txn #1

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

VALIDATE PHASE

WRITE PHASE

COMMIT

<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>
OBSERVATION

When there is low contention, optimistic protocols perform better because the DBMS spends less time checking for conflicts.

At high contention, the both classes of protocols degenerate to essentially the same serial execution.
CONCURRENCY CONTROL EVALUATION

Compare in-memory concurrency control protocols at high levels of parallelism.
→ Single test-bed system.
→ Evaluate protocols using core counts beyond what is available on today's CPUs.

Running in extreme environments exposes what are the main bottlenecks in the DBMS.
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.
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.
## 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>
# 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>
</tbody>
</table>

**Other Methods**

- TIMESTAMP
- MVCC
- OCC
  - Basic T/O Algorithm
  - Multi-Version T/O
  - Optimistic Concurrency Control

- DB2, SQL Server, SQLite, MySQL, Sybase, CUBRID
## 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>
READ-ONLY WORKLOAD

Graph showing throughput (Million txn/s) vs. number of cores for different strategies.

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

The diagram shows the throughput (in million transactions per second) for different systems under varying numbers of cores. The systems include DL_DETECT, TIMESTAMP, NO_WAIT, MVCC, WAIT_DIE, and OCC. The throughput increases as the number of cores increases for all systems, with DL_DETECT and NO_WAIT showing the highest throughput at 1000 cores.
WRITE-INTENSIVE / HIGH-CONTENTION

![Graph showing throughput as a function of number of cores for different contention strategies.](image-url)
WRITE-INTENSIVE / HIGH-CONTENTION

![Graph showing throughput vs. number of cores for different algorithms: DL_DETECT, TIMESTAMP, NO_WAIT, MVCC, WAIT_DIE, OCC. The graph highlights the performance drop at high contention.](image)
WRITE-INTENSIVE / HIGH-CONTENTION

Diagram showing the breakdown of work, aborts, and wait times for different concurrency control mechanisms. The mechanisms compared are DL_DETECT, NO_WAIT, WAIT_DIE, TIMESTAMP, MVCC, and OCC. The legend indicates the following:
- Useful Work
- Abort
- Ts Alloc.
- Index
- Wait
- Manager
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

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

- theta=0
- theta=0.6
- theta=0.8

Graph showing throughput in million transactions per second against the number of cores, with different lines indicating different values of theta.
Lock Thrashing

Lock thrashing is a situation where the number of active transactions exceeds the number of locks available. It occurs when the number of active transactions reaches a critical point, causing many transactions to be blocked due to the lack of available locks.

In the context of database management, lock thrashing can lead to a decrease in system performance. When too many transactions are waiting for locks, the system's overall throughput is reduced, as the time spent waiting for locks increases.

To prevent lock thrashing, database systems use various strategies to manage locks efficiently. These strategies include using lock granularity, controlling the number of active transactions, and implementing lock escalation mechanisms. By managing locks effectively, systems can maintain high performance even under high transaction loads.
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

![Graph showing throughput (in million ts/s) versus number of cores for different methods: Clock, Hardware, Atomic batch=16, Atomic batch=8, Atomic, Mutex.](graph.png)
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.
OBSERVATION

Serializability is useful because it allows programmers to ignore concurrency issues but enforcing it may allow too little parallelism and limit performance.

We may want to use a weaker level of consistency to improve scalability.
ISOLATION LEVELS

Controls the extent that a txn is exposed to the actions of other concurrent txns.
Provides for greater concurrency at the cost of exposing txns to uncommitted changes:
→ Dirty Read Anomaly
→ Unrepeatable Reads Anomaly
→ Phantom Reads Anomaly
ANSI ISOLATION LEVELS

**SERIALIZABLE**
→ No phantoms, all reads repeatable, no dirty reads.

**REPEATABLE READS**
→ Phantoms may happen.

**READ COMMITTED**
→ Phantoms and unrepeatable reads may happen.

**READ UNCOMMITTED**
→ All of them may happen.
ISOLATION LEVEL HIERARCHY

- READ UNCOMMITTED
- READ COMMITTED
- REPEATABLE READS
- SERIALIZABLE
# Real-World Isolation Levels

<table>
<thead>
<tr>
<th>Database</th>
<th>Default</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actian Ingres</td>
<td>SERIALIZABLE</td>
<td>SERIALIZABLE</td>
</tr>
<tr>
<td>Greenplum</td>
<td>READ COMMITTED</td>
<td>SERIALIZABLE</td>
</tr>
<tr>
<td>IBM DB2</td>
<td>CURSOR STABILITY</td>
<td>SERIALIZABLE</td>
</tr>
<tr>
<td>MySQL</td>
<td>REPEATABLE READS</td>
<td>SERIALIZABLE</td>
</tr>
<tr>
<td>MemSQL</td>
<td>READ COMMITTED</td>
<td>READ COMMITTED</td>
</tr>
<tr>
<td>MS SQL Server</td>
<td>READ COMMITTED</td>
<td>SERIALIZABLE</td>
</tr>
<tr>
<td>Oracle</td>
<td>READ COMMITTED</td>
<td>SNAPSHOT ISOLATION</td>
</tr>
<tr>
<td>Postgres</td>
<td>READ COMMITTED</td>
<td>SERIALIZABLE</td>
</tr>
<tr>
<td>SAP HANA</td>
<td>READ COMMITTED</td>
<td>SERIALIZABLE</td>
</tr>
<tr>
<td>VoltDB</td>
<td>SERIALIZABLE</td>
<td>SERIALIZABLE</td>
</tr>
</tbody>
</table>

Source: Peter Bailis
CRITICISM OF ISOLATION LEVELS

The isolation levels defined as part of SQL-92 standard only focused on anomalies that can occur in a 2PL-based DBMS.

Two additional isolation levels:
→ CURSOR STABILITY
→ SNAPSHOT ISOLATION
CURSOR STABILITY (CS)

The DBMS’s internal cursor maintains a lock on a item in the database until it moves on to the next item.

CS is a stronger isolation level in between REPEATABLE READS and READ COMMITTED that can (sometimes) prevent the Lost Update Anomaly.

Source: Jepsen
LOST UPDATE ANOMALY

**Txn #1**

BEGIN

READ(A)

... ...

WRITE(A)

COMMIT

**Txn #2**

BEGIN

WRITE(A)

... ...

COMMIT
LOST UPDATE ANOMALY

**Txn #1**
BEGIN
READ(A)  
...  
WRITE(A)  
COMMIT

**Txn #2**
BEGIN
...  
WRITE(A)  
...  
COMMIT
LOST UPDATE ANOMALY

Txn #1

BEGIN
READ(A)

Txn #2

BEGIN
WRITE(A)

COMMIT

COMMIT
LOST UPDATE ANOMALY

**Txn #1**

BEGIN

READ(A) \[\ldots\] WRITE(A)

COMMIT

**Txn #2**

BEGIN

\[\ldots\] WRITE(A) \[\ldots\] COMMIT
**LOST UPDATE ANOMALY**

Txn #1

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

Txn #2

```
BEGIN
  ...
  WRITE(A)
  ...
  COMMIT
```

Txn #2’s write to A will be lost even though it commits after Txn #1.

A **cursor lock** on A would prevent this problem.
SNAPSHOT ISOLATION (SI)

Guarantees that all reads made in a txn see a consistent snapshot of the database that existed at the time the txn started.

→ A txn will commit under SI only if its writes do not conflict with any concurrent updates made since that snapshot.

SI is susceptible to the Write Skew Anomaly
WRITE SKEW ANOMALY

**Txn #1**
Change white marbles to black.

**Txn #2**
Change black marbles to white.
WRITE SKEW ANOMALY

**Txn #1**
Change white marbles to black.

**Txn #2**
Change black marbles to white.
**WRITE SKEW ANOMALY**

*Txn #1*
Change white marbles to black.

*Txn #2*
Change black marbles to white.
WRITE SKEW ANOMALY

**Txn #1**
*Change white marbles to black.*

![Diagram of Txn #1]

**Txn #2**
*Change black marbles to white.*

![Diagram of Txn #2]
**WRITE SKEW ANOMALY**

*Txn #1*
Change white marbles to black.

*Txn #2*
Change black marbles to white.
**WRITE SKEW ANOMALY**

*Txn #1*
Change white marbles to black.

*Txn #2*
Change black marbles to white.
ISOLATION LEVEL HIERARCHY

- Serializable
  - Repeatable Reads
  - Snapshot Isolation
  - Cursor Stability
  - Read Committed
  - Read Uncommitted
Figure 4-1: A partial order to relate various isolation levels.

Source: Atul Adya
Transactions are hard.
Transactions are awesome.

Things get even more wild when we add more internal components to the DBMS:
→ Indexes
→ Triggers
→ Catalogs
→ Sequences
→ Materialized Views
NEXT CLASS

Multi-Version Concurrency Control