Transaction Processing Overview

Transaction processing studied in depth in CS W4112-Database System Implementation
Transactions

• A user program may carry out many operations on the data retrieved from the database, but the DBMS is only concerned about what data is read from, and written to, the database.

• A transaction is the DBMS’s abstract view of a user program, as simply a sequence of reads and writes to the database.
Transactions

A transaction is a series of actions (**Reads** and **Writes**) on a database that form a “logical unit”

Example: all database actions required to transfer money from one bank account to another
Transactions

A transaction is a series of actions (Reads and Writes) on a database that form a “logical unit”

Example: all database actions required to transfer money from one bank account to another

**ACID properties** of transactions:

- **Atomicity** (enforced via log + recovery protocol)
- **Consistency** (enforced by DBMS)
- **Isolation** (enforced by concurrency control protocol)
- **Durability** (enforced via log + recovery protocol)

Transactions either **commit** (when they complete successfully) or **abort** (when they don’t)
Atomicity of Transactions

- A transaction might commit after completing all its actions, or it could abort (or be aborted by the DBMS) after executing some actions
Atomicity of Transactions

• A transaction might commit after completing all its actions, or it could abort (or be aborted by the DBMS) after executing some actions

• A very important property guaranteed by the DBMS for all transactions is that they are atomic, so we can think of a transaction as always either:
  • Executing all its actions in one step, or
  • Not executing any actions at all
Atomicity of Transactions

- A transaction might commit after completing all its actions, or it could abort (or be aborted by the DBMS) after executing some actions.
- A very important property guaranteed by the DBMS for all transactions is that they are atomic, so we can think of a transaction as always either:
  - Executing all its actions in one step, or
  - Not executing any actions at all
- The DBMS logs all actions so that it can undo the actions of aborted transactions.
Consistency of Transactions

- The DBMS doesn’t understand the semantics of the data beyond the constraints that are defined as part of the database schema
- The DBMS makes sure all such constraints hold
Consistency of Transactions

- The DBMS doesn’t understand the semantics of the data beyond the constraints that are defined as part of the database schema.
- The DBMS makes sure all such constraints hold.
- A transaction can cause temporary violations of database constraints while it is still in progress, but everything should be back to normal when the transaction commits.
- Specifically, if the database is in a consistent state (i.e., all constraints hold) when a transaction starts executing, then the database is back in a consistent state after the transaction commits.
Isolation of Transactions

• Concurrent execution of transactions is essential for good DBMS performance
• Concurrency is achieved by the DBMS, which interleaves actions (reads and writes of database objects) of various transactions
• Users submit transactions, and can think of each transaction as executing by itself, logically speaking
Isolation of Transactions

- Concurrent execution of transactions is essential for good DBMS performance
- Concurrency is achieved by the DBMS, which interleaves actions (reads and writes of database objects) of various transactions
- Users submit transactions, and can think of each transaction as executing by itself, logically speaking
- The DBMS supports this “illusion” via a concurrency control protocol such as **Strict 2-Phase Locking**
Durability of Transactions

• If a transaction commits (i.e., it completes successfully), then its effects on the database have to persist forever and cannot be forgotten
• Any changes to the database have to survive system crashes and even natural disasters
Durability of Transactions

• If a transaction commits (i.e., it completes successfully), then its effects on the database have to persist forever and cannot be forgotten.

• Any changes to the database have to survive system crashes and even natural disasters.

• The DBMS logs all actions so that it can redo the actions of committed transactions if needed, to guarantee transaction durability.
Concurrency Control, for *Isolation*

Consider two transactions T1 and T2:

- **T1**: BEGIN $A=A+100$, $B=B-100$ END
- **T2**: BEGIN $A=1.02^*A$, $B=1.02^*B$ END

- T1 transfers $100$ from account B to account A
- T2 gives each account 2% interest
Concurrent Control, for Isolation

Consider two transactions T1 and T2:

**T1:** BEGIN  A=A+100, B=B-100  END
**T2:** BEGIN  A=1.02*A, B=1.02*B  END

- T1 transfers $100 from account B to account A
- T2 gives each account 2% interest

If T1 and T2 are submitted to the database at about the same time, there is no guarantee that T1 will execute before T2 or vice versa
Concurrency Control, for **Isolation**

Consider two transactions T1 and T2:

\[
\text{T1: BEGIN } A=A+100, \ B=B-100 \ \text{END} \\
\text{T2: BEGIN } A=1.02^*A, \ B=1.02^*B \ \text{END}
\]

- T1 transfers $100 from account B to account A
- T2 gives each account 2% interest

If T1 and T2 are submitted to the database at about the same time, there is no guarantee that T1 will execute before T2 or vice versa.

However, the net effect must be equivalent to T1 and T2 running **serially** in some order (i.e., T1 followed by T2, or T2 followed by T1)
One possible interleaving (or “schedule”) of the actions of T1 and T2

\[
\text{T1: } A = A + 100, \quad B = B - 100 \\
\text{T2: } \quad A = 1.02 \times A, \quad B = 1.02 \times B
\]

Is this a “good” schedule?
One possible interleaving (or “schedule”) of the actions of T1 and T2

<table>
<thead>
<tr>
<th>T1:</th>
<th>A=A+100,</th>
<th>B=B-100</th>
</tr>
</thead>
<tbody>
<tr>
<td>T2:</td>
<td>A=1.02*A,</td>
<td>B=1.02*B</td>
</tr>
</tbody>
</table>

The schedule is OK, because it’s logically equivalent—in terms of its effects on the database contents—to executing T1 fully, followed by executing T2 fully (i.e., it is equivalent to serial schedule T1; T2):

<table>
<thead>
<tr>
<th>T1:</th>
<th>A=A+100,</th>
<th>B=B-100</th>
</tr>
</thead>
<tbody>
<tr>
<td>T2:</td>
<td>A=1.02*A,</td>
<td>B=1.02*B</td>
</tr>
</tbody>
</table>

The two schedules above are equivalent because we can get from one to the other by swapping the order of B=B-100 and A=1.02*A, which are nonconflicting actions affecting different objects in the database (i.e., B and A, respectively)
Another possible interleaving (or “schedule”) of the actions of T1 and T2

| T1: A=A+100,                                  | B=B-100          |
| T2: A=1.02*A, B=1.02*B                          |

Is this a “good” schedule?
Another possible interleaving (or "schedule") of the actions of T1 and T2

<table>
<thead>
<tr>
<th>T1:</th>
<th>A=A+100,</th>
<th>B=B-100</th>
</tr>
</thead>
<tbody>
<tr>
<td>T2:</td>
<td>A=1.02<em>A, B=1.02</em>B</td>
<td></td>
</tr>
</tbody>
</table>

The schedule is not OK, because it’s not logically equivalent—in terms of its effects on the database contents—to either T1; T2 or T2; T1
Concurrent Control: Some Definitions

- **Serial schedule** is a schedule that does not interleave the actions of its transactions (i.e., each transaction in the schedule is fully executed before the next transaction is fully executed, and so on)
Concurrency Control: Some Definitions

• **Serial schedule** is a schedule that does not interleave the actions of its transactions (i.e., each transaction in the schedule is fully executed before the next transaction is fully executed, and so on)

• Two schedules are **equivalent** if, for any database state, the effect on the database of executing the first schedule is identical to the effect of executing the second schedule.
Concurrent Control: Some Definitions

- **Serial schedule** is a schedule that does not interleave the actions of its transactions (i.e., each transaction in the schedule is fully executed before the next transaction is fully executed, and so on).
- Two schedules are **equivalent** if, for any database state, the effect on the database of executing the first schedule is identical to the effect of executing the second schedule.
- **Serializable schedule** is a schedule that, if we only consider its **committed** transactions, the schedule is equivalent to a serial schedule of the committed transactions.
Transaction Schedules as Series of Reads and Writes

From now on, we will view transactions and schedules as sequences of reads and writes, without worrying about the actual values that are read or written.
Transaction Schedules as Series of Reads and Writes

From now on, we will view transactions and schedules as sequences of reads and writes, without worrying about the actual values that are read or written.

So we will write a schedule such as:

\[
\begin{array}{ll}
T1: & A = A + 100, \\
    & B = B - 100 \\
T2: & A = 1.02 \times A, \\
    & B = 1.02 \times B
\end{array}
\]

simply as reads \( R \) and writes \( W \), where the number next to \( R \) or \( W \) identifies the corresponding transaction (e.g., \( R1(A) \) is a read of object \( A \) by transaction \( T1 \)).

\[
\begin{array}{ll}
T1: & R1(A) \ W1(A) \\
    & R1(B) \ W1(B) \\
T2: & R2(A) \ W2(A) \\
    & R2(B) \ W2(B)
\end{array}
\]

or sometimes, equivalently, in one line as:

\[
R1(A) \ W1(A) \ R2(A) \ W2(A) \ R1(B) \ W1(B) \ R2(B) \ W2(B)
\]
Anomalies with Interleaved Executions of Transactions

- Reading Uncommitted Data ("dirty reads"):

<table>
<thead>
<tr>
<th>T1: R1(A) W1(A)</th>
<th>R1(B) W1(B) Abort1</th>
</tr>
</thead>
<tbody>
<tr>
<td>T2: R2(A) W2(A)</td>
<td>Commit2</td>
</tr>
</tbody>
</table>
Anomalies with Interleaved Executions of Transactions

- **Reading Uncommitted Data ("dirty reads"):**

<table>
<thead>
<tr>
<th>T1: R1(A) W1(A)</th>
<th>R1(B) W1(B) Abort1</th>
</tr>
</thead>
<tbody>
<tr>
<td>T2:</td>
<td>R2(A) W2(A) Commit2</td>
</tr>
</tbody>
</table>

- **Unrepeatable Reads:**

<table>
<thead>
<tr>
<th>T1: R1(A)</th>
<th>R1(A) W1(A) Commit1</th>
</tr>
</thead>
<tbody>
<tr>
<td>T2: R2(A) W2(A) Commit2</td>
<td></td>
</tr>
</tbody>
</table>
Anomalies (Continued)

- Overwriting Uncommitted Data:

```
T1: W1(A) W1(B) Commit1
T2: W2(A) W2(B) Commit2
```

All of these schedules are problematic (because they are not serializable or have other anomalies)

How can we avoid them?
Strict 2-Phase Locking (Strict 2PL) Protocol

Strict 2PL allows only serializable schedules

- **Before reading** an object A, a transaction must have requested and hold a **shared lock** on A (i.e., S(A))
- Multiple transactions can hold a shared lock on an object simultaneously
Strict 2-Phase Locking (Strict 2PL) Protocol

Strict 2PL allows only serializable schedules

• **Before reading** an object $A$, a transaction must have requested and hold a **shared lock** on $A$ (i.e., $S(A)$)

• Multiple transactions can hold a shared lock on an object simultaneously

• **Before writing** an object $A$, a transaction must have requested and hold an **exclusive lock** on $A$ (i.e., $X(A)$)

• At most one transaction can hold an exclusive lock on an object and no transactions can hold a shared lock on the object
Strict 2-Phase Locking (Strict 2PL) Protocol

Strict 2PL allows only serializable schedules

- **Before reading** an object A, a transaction must have requested and hold a **shared lock** on A (i.e., S(A))
- Multiple transactions can hold a shared lock on an object simultaneously
- **Before writing** an object A, a transaction must have requested and hold an **exclusive lock** on A (i.e., X(A))
- At most one transaction can hold an exclusive lock on an object and no transactions can hold a shared lock on the object
- Transactions only **release their locks (U) when they complete** (i.e., as part of COMMIT or ABORT)
What are the 2 phases in Strict 2PL?

- Phase 1: execution of the transaction, where locks are acquired but never released
- Phase 2: COMMIT or ABORT of the transaction, where all locks are finally released

Examples of schedules possible and not possible under Strict 2PL? On board …
Aborting a Transaction

Consider a transaction T1:

\[ T1: R1(A) \ W1(A) \ W1(B) \ R1(C) \ Abort1 \]

What should we do to remove all effects of T1 from the database as we terminate the transaction?
Abort a Transaction

Consider a transaction T1:

\[ \text{T1: R1(A) W1(A) W1(B) R1(C) Abort1} \]

What should we do to remove all \textit{effects of T1} from the database as we terminate the transaction?

\[ \text{T1: R1(A) W1(A) W1(B) R1(C) W1(B) W1(A) Abort1} \]
Aborting a Transaction

Consider a transaction T1:

T1: R1(A) W1(A) W1(B) R1(C) Abort1

What should we do to remove all effects of T1 from the database as we terminate the transaction?

T1: R1(A) W1(A) W1(B) R1(C) W1(B) W1(A) Abort1

• As part of the abort operation, we write back/restore the old values of B and A that T1 received, which T1 can do because it is still holding exclusive locks on B and A (strict 2PL…).
Abort a Transaction

Consider a transaction T1:
T1: R1(A) W1(A) W1(B) R1(C) Abort1

What should we do to remove all effects of T1 from the database as we terminate the transaction?
T1: R1(A) W1(A) W1(B) R1(C) W1(B) W1(A) Abort1

- As part of the abort operation, we write back/restore the old values of B and A that T1 received, which T1 can do because it is still holding exclusive locks on B and A (strict 2PL…)
- The old values are in the database log, in which the old and new values of every write to the database are recorded
- Finally T1 releases all locks that T1 acquired
Cascading Aborts

- If a transaction Ti is aborted, all its actions have to be undone
- Furthermore, if Tj read an object last written by Ti, Tj must be aborted as well
Cascading Aborts

• If a transaction $T_i$ is aborted, all its actions have to be undone
• Furthermore, if $T_j$ read an object last written by $T_i$, $T_j$ must be aborted as well.
• **Strict 2PL** avoids such cascading aborts because transactions only release their locks at the very end (i.e., at commit or abort time). So if $T_i$ writes an object, it holds on to its exclusive lock on the object until the end; $T_j$ cannot read the object until after $T_i$ commits or aborts.
Unrecoverable Schedules

Consider this schedule for transactions T1 and T2, with the “reading uncommitted data” problem we saw:

T1: R1(A) \textbf{W1(A)} \quad \text{Abort1}

T2: \quad \textbf{R2(A)} \quad \text{R2(B)} \quad \text{W2(B)} \quad \text{Commit2}

Is the schedule serializable?
Unrecoverable Schedules

Consider this schedule for transactions T1 and T2, with the “reading uncommitted data” problem we saw:

T1: R1(A) **W1(A)** Abort1
T2: **R2(A)** R2(B) W2(B) Commit2

Is the schedule serializable? Yes
Is there a problem?
Unrecoverable Schedules

Consider this schedule for transactions T1 and T2, with the “reading uncommitted data” problem we saw:

T1:  R1(A)  \textbf{W1(A)} \hspace{2cm} \textbf{Abort1}

T2:  \textbf{R2(A)}  \textbf{R2(B)}  \textbf{W2(B)} \hspace{0.5cm} \textbf{Commit2}

Is the schedule serializable? Yes

\textbf{Is there a problem? Yes!} T2 reads a value of A written by T1, T2 commits (so because of durability, T2 persists forever), and then T1 aborts (so because of atomicity, all of T1’s actions should be erased)
Recoverable Schedules

A schedule is **recoverable** if its transactions commit only after—and if—all transactions whose changes they read have committed.

Strict 2PL does not allow unrecoverable schedules, so the schedule in the previous slide is not possible under Strict 2PL.

Only serializable, recoverable schedules that avoid cascading aborts are possible under Strict 2PL.
The Phantom Problem

Consider this schedule for T1 and T2, following Strict 2PL:

T1: locks all pages with sailors with rating=1 and finds oldest such sailor is 71 years old

T2: inserts a new sailor with rating=1 and age=96 deletes oldest sailor with rating=2, who happens to be 80 years old, commits

Problem?
The Phantom Problem

Consider this schedule for T1 and T2, following Strict 2PL:

**T1:** locks all pages with sailors with *rating* = 1 and finds oldest such sailor is 71 years old

**T2:** inserts a new sailor with *rating* = 1 and age = 96, deletes oldest sailor with *rating* = 2, who happens to be 80 years old, commits

**Problem:** Yes! Schedule is not equivalent to either serial schedule:

- **Not T1; T2:** the oldest sailors for rating=1 and rating=2 that T1 observes would have been 71 and 80 (not 63) years old, respectively
- **Not T2; T1:** the oldest sailors for rating=1 and rating=2 that T1 observes would have been 96 (not 71) and 63 years old, respectively
Dynamic Databases and Phantoms

• This schedule is possible under Strict 2PL yet it’s not serializable!
Dynamic Databases and Phantoms

• This schedule is possible under Strict 2PL yet it’s not serializable!

• Problem: implicit assumption that transaction T1 has locked all sailors, current or future, with rating=1, not just existing sailors
Dynamic Databases and Phantoms

• This schedule is possible under Strict 2PL yet it’s not serializable!

• Problem: implicit assumption that transaction T1 has locked all sailors, current or future, with rating=1, not just existing sailors

• One way to handle such dynamic databases with Strict 2PL to guarantee true serializability is to use **predicate locking**: lock all current or future tuples satisfying a predicate (e.g., rating=1 OR rating=2 for T1)
Transactions have an **access mode**: READ WRITE vs. READ ONLY

- “Read Uncommitted” transactions must be READ ONLY and don’t use locks (if concurrency control is based on Strict 2PL)
- For all other isolation levels, exclusive locks are held until the end of the transactions
- For “Read Committed” transactions, shared locks are released immediately after the corresponding reads
- “Repeatable Reads” and “Serializable” transactions follow Strict 2PL
- “Serializable” transactions use predicate locking

See [https://www.postgresql.org/docs/10/transaction-iso.html](https://www.postgresql.org/docs/10/transaction-iso.html) for PostgreSQL support
ACID Properties of Transactions

- **Atomicity**
- **Consistency** ✔
- **Isolation** ✔
- **Durability**

Atomicity and durability are handled by the recovery manager

We have already discussed how to terminate a transaction during normal database operation

Now: what should we do when the system crashes, to guarantee atomicity and durability?
Recovery After a Crash

Assumptions:

• Disk is safe, RAM is not (and goes away when system crashes)
• DBMS uses Strict 2-Phase locking
The Database Log

• The log is a **single, sequential, append-only file** that keeps information on actions by all transactions

• Log record for a write operation:

<table>
<thead>
<tr>
<th>TID</th>
<th>pageID</th>
<th>offset</th>
<th>length</th>
<th>old value</th>
<th>new value</th>
</tr>
</thead>
</table>

TID=transaction ID; pageID, offset, and length refer to the block on disk, the position within the block, and the length that is written
The Database Log

• The log is a **single, sequential, append-only file** that keeps information on actions by all transactions

• Log record for a write operation:

<table>
<thead>
<tr>
<th>TID</th>
<th>pageID</th>
<th>offset</th>
<th>length</th>
<th>old value</th>
<th>new value</th>
</tr>
</thead>
</table>

TID=transaction ID; pageID, offset, and length refer to the block on disk, the position within the block, and the length that is written

• When a transaction **writes** an object, the log record includes both the **old** and **new** values for the object

• When a transaction **commits** or **aborts**, a log record documents this action
The Database Log

• The log is a **single, sequential, append-only file** that keeps information on actions by all transactions

• Log record for a write operation:

  | TID | pageID | offset | length | old value | new value |

  TID=transaction ID; pageID, offset, and length refer to the block on disk, the position within the block, and the length that is written

• When a transaction **writes** an object, the log record includes both the **old** and **new** values for the object

• When a transaction **commits** or **aborts**, a log record documents this action

• The **most recent records of the log are in main memory**, while the rest of the log is stored stably on disk (and often replicated as well)
Write-Ahead Logging, or WAL

For efficiency, we allow writes of data and log to happen in memory without an immediate disk-write counterpart: dangerous if not done properly! Why?
Write-Ahead Logging, or WAL

For efficiency, we allow writes of data and log to happen in memory without an immediate disk-write counterpart: dangerous if not done properly!

Write-Ahead Logging rules:

• We must force the log record for an update to disk before the corresponding data page for the update makes it to disk
Write-Ahead Logging, or WAL

For efficiency, we allow writes of data and log to happen in memory without an immediate disk-write counterpart: dangerous if not done properly!

Write-Ahead Logging rules:

• We must force the log record for an update to disk before the corresponding data page for the update makes it to disk

• We must write all log records for a transaction—including the commit record—to disk before declaring the transaction committed

Note: The log is a sequential file, so if we push a log record to disk, all earlier log records are also pushed to disk
Recover from a Crash

T1: commit
T2: commit
T3: commit

Recovery?

CRASH!
Recovering from a Crash

During recovery:

- **Redo updates by committed transactions** T1, T2, and T3 (their updates might not have made it to disk by the time of the crash); use log for this
- **Undo updates by in-progress transactions** T4 and T5 (their updates might have made it to disk); use log for this; these transactions get terminated and restarted from scratch
ARIES Recovery Algorithm: 3 Phases

• **Analysis:** Scan the log to identify all transactions that had committed or aborted, or were in progress at crash time
ARIES Recovery Algorithm: 3 Phases

- **Analysis:** Scan the log to identify all transactions that had committed or aborted, or were in progress at crash time.

- **Redo:** Redo all updates—by using the new values in the corresponding log records—to ensure that all logged updates are indeed carried out, in chronological order.
ARIES Recovery Algorithm: 3 Phases

• **Analysis:** Scan the log to identify all transactions that had committed or aborted, or were in progress at crash time

• **Redo:** Redo all updates—by using the new values in the corresponding log records—to ensure that all logged updates are indeed carried out, in chronological order

• **Undo:** Undo all updates—by using the old values in the corresponding log records—of in-progress transactions, in reverse chronological order
Summary of Transaction Processing

• ACID properties of transactions are critically important for sensitive applications
• Transaction processing—with the enforcement of the ACID properties—is done largely transparently by DBMS
• This is a big advantage of using DBMSs over ad-hoc software
• CS W4112-Database System Implementation covers transaction processing in depth
COMS E6111
Advanced Database Systems

Spring 2020

Prerequisites:
COMS W4111 (not W4112);
fluency in Java or Python
Advanced Database Systems: 2 Themes

• **Broader families of “data,”** beyond relational model
  • text, spatial data, time series, …

• **Broader families of “queries,”** beyond relatively simple SQL
  • keyword search, “data mining”/OLAP, …
Sample of Potential Topics Covered

• Information Retrieval
• Web Search
• Data Mining
• Data Warehousing, OLAP, Decision Support
• Spatial Data Management
• Time Series Analysis
• Information Extraction
• …
General Structure of Course (subject to change)

- Regular lectures, but with more discussion
- No homework
- Many readings through research papers
- 3 projects (in Java or Python)
- Midterm and final

Both undergraduate and graduate students can register