Transactions and Concurrency

Grant Weddell

Cheriton School of Computer Science
University of Waterloo

CS 348
Introduction to Database Management
Winter 2017
1. Why We Need Transactions
   - Failures
   - Concurrency

2. Serializability
   - Serializable Schedules
   - Serialization Graphs

3. Transactions in SQL
   - Abort and Commit
   - Isolation Levels

4. Implementing Transactions
   - Concurrency Control
   - Recovery Management
Problems Caused by Failures

- Update all account balances at a bank branch.

\[\text{Accounts(Anum, CID, BranchId, Balance)}\]

\[\text{update Accounts set Balance = Balance * 1.05}\]
\[\text{where BranchId = 12345}\]

Problem
If the system crashes while processing this update, some, but not all, tuples with BranchId = 12345 may have been updated.
Problems Caused by Failures

- Update all account balances at a bank branch.

\[ \text{update Accounts}
  \]
\[ \text{set Balance} = \text{Balance} \times 1.05 
  \]
\[ \text{where BranchId} = 12345 \]

**Problem**
If the system crashes while processing this update, some, but not all, tuples with \( \text{BranchId} = 12345 \) may have been updated.
Another Failure-Related Problem

- transfer money between accounts:
  
  \[
  \text{update Accounts} \\
  \text{set Balance = Balance - 100} \\
  \text{where Anum = 8888}
  \]

  \[
  \text{update Accounts} \\
  \text{set Balance = Balance + 100} \\
  \text{where Anum = 9999}
  \]
Another Failure-Related Problem

- transfer money between accounts:
  
  ```sql
  update Accounts
  set Balance = Balance - 100
  where Anum = 8888
  
  update Accounts
  set Balance = Balance + 100
  where Anum = 9999
  ```

Problem

If the system fails between these updates, money may be withdrawn but not redeposited.
Problems Caused by Concurrency

- Application 1:

```sql
update Accounts
set Balance = Balance - 100
where Anum = 8888

update Accounts
set Balance = Balance + 100
where Anum = 9999
```

Problem

If the applications run concurrently, the total balance returned to application 2 may be inaccurate.
Problems Caused by Concurrency

• Application 1:

```sql
update Accounts
set Balance = Balance - 100
where Anum = 8888

update Accounts
set Balance = Balance + 100
where Anum = 9999
```

• Application 2:

```sql
select Sum(Balance)
from Accounts
```
Problems Caused by Concurrency

- **Application 1:**
  
  ```sql
  update Accounts
  set Balance = Balance - 100
  where Anum = 8888

  update Accounts
  set Balance = Balance + 100
  where Anum = 9999
  ```

- **Application 2:**
  
  ```sql
  select Sum(Balance)
  from Accounts
  ```

**Problem**

If the applications run concurrently, the total balance returned to application 2 may be inaccurate.
Another Concurrency Problem

- **Application 1:**
  ```sql
  select balance into :balance
  from Accounts
  where Anum = 8888
  
  compute :newbalance using :balance
  
  update Accounts
  set Balance = :newbalance
  where Anum = 8888
  ```

- **Application 2:** same as Application 1

Problem

If the applications run concurrently, one of the updates may be "lost."
Another Concurrency Problem

- Application 1:
  ```sql
  select balance into :balance
  from Accounts
  where Anum = 8888
  
  compute :newbalance using :balance

  update Accounts
  set Balance = :newbalance
  where Anum = 8888
  
  Application 2: same as Application 1
Another Concurrency Problem

• Application 1:

    select balance into :balance
    from Accounts
    where Anum = 8888

    compute :newbalance using :balance

    update Accounts
    set Balance = :newbalance
    where Anum = 8888

• Application 2: same as Application 1

Problem

If the applications run concurrently, one of the updates may be “lost”.
Definition (Transaction)

An application-specified *atomic* and *durable* unit of work.

Properties of transactions ensured by the DBMS:

- **Atomic:** a transaction occurs entirely, or not at all
- **Consistency:** each transaction preserves the consistency of the database
- **Isolated:** concurrent transactions do not interfere with each other
- **Durable:** once completed, a transaction’s changes are permanent
Serializability (informal)

- Concurrent transactions must appear to have been executed sequentially, i.e., one at a time, in some order. If $T_i$ and $T_j$ are concurrent transactions, then either:
  
  1. $T_i$ will appear to precede $T_j$, meaning that $T_j$ will “see” any updates made by $T_i$, and $T_i$ will not see any updates made by $T_j$, or
  
  2. $T_i$ will appear to follow $T_j$, meaning that $T_i$ will see $T_j$’s updates and $T_j$ will not see $T_i$’s.
Serializability: An Example

- An interleaved execution of two transactions, $T_1$ and $T_2$:
  \[ H_a = w_1[x] r_2[x] w_1[y] r_2[y] \]

$H_a$ is serializable because it is equivalent to $H_b$, a serial schedule.

$H_c$ is not serializable.
Serializability: An Example

- An interleaved execution of two transactions, $T_1$ and $T_2$:
  \[ H_a = w_1[x] r_2[x] w_1[y] r_2[y] \]

- An equivalent serial execution of $T_1$ and $T_2$:
  \[ H_b = w_1[x] w_1[y] r_2[x] r_2[y] \]
Serializability: An Example

- An interleaved execution of two transactions, $T_1$ and $T_2$:
  \[ H_a = w_1[x] r_2[x] w_1[y] r_2[y] \]

- An equivalent serial execution of $T_1$ and $T_2$:
  \[ H_b = w_1[x] w_1[y] r_2[x] r_2[y] \]

- An interleaved execution of $T_1$ and $T_2$ with no equivalent serial execution:
  \[ H_c = w_1[x] r_2[x] r_2[y] w_1[y] \]
Serializability: An Example

- An interleaved execution of two transactions, $T_1$ and $T_2$:
  \[ H_a = w_1[x] \ r_2[x] \ w_1[y] \ r_2[y] \]

- An equivalent serial execution of $T_1$ and $T_2$:
  \[ H_b = w_1[x] \ w_1[y] \ r_2[x] \ r_2[y] \]

- An interleaved execution of $T_1$ and $T_2$ with no equivalent serial execution:
  \[ H_c = w_1[x] \ r_2[x] \ r_2[y] \ w_1[y] \]

$H_a$ is serializable because it is equivalent to $H_b$, a serial schedule.
$H_c$ is not serializable.
Transactions and Histories

- Two operations conflict if:
  1. they belong to different transactions,
  2. they operate on the same object, and
  3. at least one of the operations is a write
Transactions and Histories

- Two operations conflict if:
  - they belong to different transactions,
  - they operate on the same object, and
  - at least one of the operations is a write

- A transaction is a sequence of read and write operations.
Transactions and Histories

- Two operations conflict if:
  1. they belong to different transactions,
  2. they operate on the same object, and
  3. at least one of the operations is a write

- A transaction is a sequence of read and write operations.

- An execution history over a set of transactions $T_1 \ldots T_n$ is an interleaving of the operations of $T_1 \ldots T_n$ in which the operation ordering imposed by each transaction is preserved.
Transactions and Histories

• Two operations conflict if:
  1. they belong to different transactions,
  2. they operate on the same object, and
  3. at least one of the operations is a write

• A transaction is a sequence of read and write operations.

• An execution history over a set of transactions $T_1 \ldots T_n$ is an interleaving of the operations of $T_1 \ldots T_n$ in which the operation ordering imposed by each transaction is preserved.

• Two important assumptions:
  1. Transactions interact with each other only via reads and writes of objects
  2. A database is a fixed set of independent objects
Definition (Conflict) Equivalence

Two histories are \textit{(conflict) equivalent} if

- they are over the same set of transactions, and
- the ordering of each pair of conflicting operations is the same in each history
Definition ((Conflict) Equivalence)

Two histories are (conflict) equivalent if

- they are over the same set of transactions, and
- the ordering of each pair of conflicting operations is the same in each history

Definition ((Conflict) Serializability)

A history $H$ is said to be (conflict) serializable if there exists some serial history $H'$ that is (conflict) equivalent to $H$
Testing for Serializability

\[ r_1[x] \quad r_3[x] \quad w_4[y] \quad r_2[u] \quad w_4[z] \quad r_1[y] \quad r_3[u] \quad r_2[z] \quad w_2[z] \quad r_3[z] \quad r_1[z] \quad w_3[y] \]

Is this history serializable?
Testing for Serializability

Is this history serializable?

\[ r_1[x] \ r_3[x] \ w_4[y] \ r_2[u] \ w_4[z] \ r_1[y] \ r_3[u] \ r_2[z] \ w_2[z] \ r_3[z] \ r_1[z] \ w_3[y] \]

Theorem

A history is serializable iff its serialization graph is acyclic.
Serialization Graphs

\[ r_1[x] \ r_3[x] \ w_4[y] \ r_2[u] \ w_4[z] \ r_1[y] \ r_3[u] \ r_2[z] \ w_2[z] \ r_3[z] \ r_1[z] \ w_3[y] \]
The history above is equivalent to

\[ w_4[y] w_4[z] r_2[u] r_2[z] w_2[z] r_1[x] r_1[y] r_1[z] r_3[x] r_3[u] r_3[z] w_3[y] \]

That is, it is equivalent to executing \( T_4 \) followed by \( T_2 \) followed by \( T_1 \) followed by \( T_3 \).
• A transaction may terminate in one of two ways:
Abort and Commit

• A transaction may terminate in one of two ways:
  • When a transaction *commits*, any updates it made become durable, and they become visible to other transactions. A commit is the “all” in “all-or-nothing” execution.
  • When a transaction *aborts*, any updates it may have made are undone (erased), as if the transaction never ran at all. An abort is the “nothing” in “all-or-nothing” execution.

• A transaction that has started but has not yet aborted or committed is said to be *active*. 
Abort and Commit

- A transaction may terminate in one of two ways:
  - When a transaction \textit{commits}, any updates it made become durable, and they become visible to other transactions. A commit is the “all” in “all-or-nothing” execution.
  - When a transaction \textit{aborts}, any updates it may have made are undone (erased), as if the transaction never ran at all. An abort is the “nothing” in “all-or-nothing” execution.
Abort and Commit

- A transaction may terminate in one of two ways:
  - When a transaction *commits*, any updates it made become durable, and they become visible to other transactions. A commit is the “all” in “all-or-nothing” execution.
  - When a transaction *aborts*, any updates it may have made are undone (erased), as if the transaction never ran at all. An abort is the “nothing” in “all-or-nothing” execution.
- A transaction that has started but has not yet aborted or committed is said to be *active*. 
Transactions in SQL

- A new transaction is begun when an application first executes an SQL command.
A new transaction is begun when an application first executes an SQL command.

Two SQL commands are available to terminate a transaction:

- `commit work`: commits the transaction
- `rollback work`: abort the transaction
Transactions in SQL

- A new transaction is begun when an application first executes an SQL command.
- Two SQL commands are available to terminate a transaction:
  - `commit work`: commits the transaction
  - `rollback work`: abort the transaction
- A new transaction begins with the application’s next SQL command after `commit work` or `rollback work`. 
SQL allows the serializability guarantee to be relaxed, if necessary.
SQL Isolation Levels

- SQL allows the serializability guarantee to be relaxed, if necessary.
- For each transaction, it is possible to specify an isolation level.

Level 0 (Read Uncommitted): transaction may see uncommitted updates
Level 1 (Read Committed): transaction sees only committed changes, but non-repeatable reads are possible
Level 2 (Repeatable Read): reads are repeatable, but “phantoms” are possible
Level 3 (Serializability)
SQL allows the serializability guarantee to be relaxed, if necessary.

For each transaction, it is possible to specify an isolation level.

Four isolation levels are supported, with the highest being serializability:

Level 0 (Read Uncommitted): transaction may see uncommitted updates

Level 1 (Read Committed): transaction sees only committed changes, but non-repeatable reads are possible

Level 2 (Repeatable Read): reads are repeatable, but “phantoms” are possible

Level 3 (Serializability)
Non-Repeatable Reads

- Application 1:
  ```sql
  update Employee
  set Salary = Salary + 1000
  where WorkDept = 'D11'
  ```

Problem

If there are employees in D11 with surnames that begin with "A", Application 2's queries may see them with different salaries.
Non-Repeatable Reads

- Application 1:
  
  ```
  update Employee
  set Salary = Salary + 1000
  where WorkDept = 'D11'
  ```

- Application 2:
  
  ```
  select * from Employee
  where WorkDept = 'D11'
  ```

  ```
  select * from Employee
  where Lastname like 'A%'
  ```

Problem

If there are employees in D11 with surnames that begin with “A”, Application 2’s queries may see them with different salaries.
Non-Repeatable Reads

- Application 1:
  ```
  update Employee
  set Salary = Salary + 1000
  where WorkDept = 'D11'
  ```

- Application 2:
  ```
  select * from Employee
  where WorkDept = 'D11'
  ```

  ```
  select * from Employee
  where Lastname like 'A%'
  ```

Problem

If there are employees in D11 with surnames that begin with “A”, Application 2’s queries may see them with different salaries.
• Application 1:
  ```sql
  insert into Employee
  values ('000123', 'Sheldon', 'Q', 'Jetstream', 'D11', '05/01/00', 52000.00)
  ```

Problem

Application 2's second query may see Sheldon Jetstream, even though its first query does not.
Phantoms

- Application 1:
  
  ```sql
  insert into Employee
  values ('000123','Sheldon','Q','Jetstream','D11','05/01/00',52000.00)
  ```

- Application 2:
  
  ```sql
  select *
  from Employee
  where WorkDept = 'D11'
  ```

  ```sql
  select *
  from Employee
  where Salary > 50000
  ```

Problem Application 2’s second query may see Sheldon Jetstream, even though its first query does not.
Phantoms

- Application 1:
  ```
  insert into Employee
  values ('000123','Sheldon','Q','Jetstream','D11',
         '05/01/00',52000.00)
  ```

- Application 2:
  ```
  select *
  from Employee
  where WorkDept = 'D11'

  select *
  from Employee
  where Salary > 50000
  ```

Problem

Application 2’s second query may see Sheldon Jetstream, even though its first query does not.
Implementing Transactions

The implementation of transactions in a DBMS has two parts:

Concurrency Control: guarantees that the execution history has the desired properties (such as serializability)

Recovery Management: guarantees that committed transactions are durable (despite failures), and that aborted transactions have no effect on the database
The implementation of transactions in a DBMS has two parts:

**Concurrency Control:** guarantees that the execution history has the desired properties (such as serializability)
Implementing Transactions

The implementation of transactions in a DBMS has two parts:

Concurreny Control: guarantees that the execution history has the desired properties (such as serializability)

Recovery Management: guarantees that committed transactions are durable (despite failures), and that aborted transactions have no effect on the database
Serializability can be guaranteed by executing transactions serially, but in many environments this leads to poor performance.

Typically, many transactions are in progress concurrently, and a concurrency control protocol is used to ensure that the resulting history is serializable. Many concurrency control protocols have been proposed, based on:

- Locking,
- Timestamps,
- Serialization graph analysis

By far the most commonly implemented protocol is strict two-phase locking. The strict two-phase locking protocol can be relaxed, as necessary, to accommodate isolation levels below serializability.
Concurrent Control

- Serializability can be guaranteed by executing transactions serially, but in many environments this leads to poor performance.
- Typically, many transactions are in progress concurrently, and a concurrency control protocol is used to ensure that the resulting history is serializable.
Concurrent Control

- Serializability can be guaranteed by executing transactions serially, but in many environments this leads to poor performance.
- Typically, many transactions are in progress concurrently, and a concurrency control protocol is used to ensure that the resulting history is serializable.
- Many concurrency control protocols have been proposed, based on:
  - locking, or
  - timestamps, or
  - serialization graph analysis

By far the most commonly implemented protocol is strict two-phase locking. The strict two-phase locking protocol can be relaxed, as necessary, to accommodate isolation levels below serializability.
Concurrency Control

- Serializability can be guaranteed by executing transactions serially, but in many environments this leads to poor performance.
- Typically, many transactions are in progress concurrently, and a concurrency control protocol is used to ensure that the resulting history is serializable.
- Many concurrency control protocols have been proposed, based on:
  - locking, or
  - timestamps, or
  - serialization graph analysis
- By far the most commonly implemented protocol is *strict two-phase locking*. 
Concurrency Control

- Serializability can be guaranteed by executing transactions serially, but in many environments this leads to poor performance.
- Typically, many transactions are in progress concurrently, and a concurrency control protocol is used to ensure that the resulting history is serializable.
- Many concurrency control protocols have been proposed, based on:
  - locking, or
  - timestamps, or
  - serialization graph analysis
- By far the most commonly implemented protocol is strict two-phase locking.
- The strict two-phase locking protocol can be relaxed, as necessary, to accommodate isolation levels below serializability.
Strict Two-Phase Locking

The rules:

1. Before a transaction may read or write an object, it must have a lock on that object.
   - a shared lock is required to read an object
   - an exclusive lock is required to write an object

2. Two or more transactions may not hold locks on the same object unless all hold shared locks.

3. A transaction may not release any locks until it commits (or aborts).

If all transactions use strict two-phase locking, the execution history is guaranteed to be serializable.
Strict Two-Phase Locking

The rules:

1. Before a transaction may read or write an object, it must have a lock on that object.
   - a *shared lock* is required to read an object
   - an *exclusive lock* is required to write an object
Strict Two-Phase Locking

The rules:

1. Before a transaction may read or write an object, it must have a lock on that object.
   - a *shared lock* is required to read an object
   - an *exclusive lock* is required to write an object

2. Two or more transactions may not hold locks on the same object unless all hold shared locks.
Strict Two-Phase Locking

The rules:

1. Before a transaction may read or write an object, it must have a lock on that object.
   - a *shared lock* is required to read an object
   - an *exclusive lock* is required to write an object

2. Two or more transactions may not hold locks on the same object unless all hold shared locks.

3. A transaction may not release any locks until it commits (or aborts).
Strict Two-Phase Locking

The rules:

1. Before a transaction may read or write an object, it must have a lock on that object.
   - a *shared lock* is required to read an object
   - an *exclusive lock* is required to write an object

2. Two or more transactions may not hold locks on the same object unless all hold shared locks.

3. A transaction may not release any locks until it commits (or aborts).

If all transactions use strict two-phase locking, the execution history is guaranteed to be serializable.
Consider the following sequence of events:

- $T_1$ acquires a shared lock on $x$ and reads $x$
- $T_2$ attempts to acquire an exclusive lock on $x$ (so that it can write $x$)
Consider the following sequence of events:

- $T_1$ acquires a shared lock on $x$ and reads $x$
- $T_2$ attempts to acquire an exclusive lock on $x$ (so that it can write $x$)

The two-phase locking rules prevent $T_2$ from acquiring its exclusive lock—this is called a lock conflict.
Consider the following sequence of events:

- $T_1$ acquires a shared lock on $x$ and reads $x$
- $T_2$ attempts to acquire an exclusive lock on $x$ (so that it can write $x$)

The two-phase locking rules prevent $T_2$ from acquiring its exclusive lock—this is called a lock conflict.

Lock conflicts can be resolved in one of two ways:

1. $T_2$ can be blocked - forced to wait until $T_1$ releases its lock
2. $T_1$ can be pre-empted - forced to abort and give up its locks
• Transaction blocking can result in *deadlocks*
Deadlocks

- Transaction blocking can result in *deadlocks*
- For example:
  - $T_1$ reads object $x$
  - $T_2$ reads object $y$
  - $T_2$ attempts to write object $x$ (it is blocked)
  - $T_1$ attempts to write object $y$ (it is blocked)
Deadlocks

- Transaction blocking can result in *deadlocks*
- For example:
  - \(T_1\) reads object \(x\)
  - \(T_2\) reads object \(y\)
  - \(T_2\) attempts to write object \(x\) (it is blocked)
  - \(T_1\) attempts to write object \(y\) (it is blocked)

A deadlock can be resolved only by forcing one of the transactions involved in the deadlock to abort.
Recovery Management

- **Recovery management** means:
  1. implementing voluntary or involuntary rollback of individual transactions
  2. implementing recovery from *system failures*
Recovery Management

- **Recovery management** means:
  1. implementing voluntary or involuntary rollback of individual transactions
  2. implementing recovery from *system failures*

- **System failure** means:
  1. the database server is halted abruptly
  2. processing of in-progress SQL command(s) is halted abruptly
  3. connections to application programs (clients) are broken.
  4. contents of memory buffers are lost
  5. database files are not damaged.
To ensure that transactions are atomic, every transaction that is active when a system failure occurs must either be

- restarted after the failure from the point it which it left off, or
- rolled back after the failure
Failures and Transactions

- To ensure that transactions are atomic, every transaction that is active when a system failure occurs must either be
  - restarted after the failure from the point it which it left off, or
  - rolled back after the failure
- It is difficult to restart applications after a system failure, so the recovery manager does the following:
  - abort transactions that were active at the time of the failure
  - ensure that changes made by transactions that committed before the failure are not lost
Logging

• Recovery management is usually accomplished using a log.

A log is a read/append data structure located in persistent storage (it must survive the failure). When transactions are running, log records are appended to the log. Log records contain:

- UNDO information: old versions of objects that have been modified by a transaction. Used to undo database changes made by a transaction that aborts.
- REDO information: new versions of objects that have been modified by a transaction. Used to redo the work done by a transaction that commits.
- BEGIN/COMMIT/ABORT: records are recorded whenever a transaction begins, commits, or aborts.

Requires Write-Ahead-Logging
Log records must be written before updating the database!
Logging

- Recovery management is usually accomplished using a log.
- A log is a read/append data structure located in persistent storage (it must survive the failure)
Logging

- Recovery management is usually accomplished using a log.
- A log is a read/append data structure located in persistent storage (it must survive the failure)
- When transactions are running, log records are appended to the log. Log records contain:
  - UNDO information: old versions of objects that have been modified by a transaction. Used to undo database changes made by a transaction that aborts.
  - REDO information: new versions of objects that have been modified by a transaction. Used to redo the work done by a transaction that commits.
  - BEGIN/COMMIT/ABORT: records are recorded whenever a transaction begins, commits, or aborts.
Logging

- Recovery management is usually accomplished using a *log*.
- A log is a read/append data structure located in persistent storage (it must survive the failure)
- When transactions are running, *log records* are appended to the log. Log records contain:
  - **UNDO information**: old versions of objects that have been modified by a transaction. Used to undo database changes made by a transaction that aborts.
  - **REDO information**: new versions of objects that have been modified by a transaction. Used to redo the work done by a transaction that commits.
  - **BEGIN/COMMIT/ABORT** records are recorded whenever a transaction begins, commits, or aborts.

Requires Write-Ahead-Logging: Log records must be written before updating the database!
Logging

- Recovery management is usually accomplished using a log.
- A log is a read/append data structure located in persistent storage (it must survive the failure)
- When transactions are running, log records are appended to the log. Log records contain:
  UNDO information: old versions of objects that have been modified by a transaction. Used to undo database changes made by a transaction that aborts.
  REDO information: new versions of objects that have been modified by a transaction. Used to redo the work done by a transaction that commits.
• Recovery management is usually accomplished using a log.
• A log is a read/append data structure located in persistent storage (it must survive the failure)
• When transactions are running, log records are appended to the log. Log records contain:
  UNDO information: old versions of objects that have been modified by a transaction. Used to undo database changes made by a transaction that aborts.
  REDO information: new versions of objects that have been modified by a transaction. Used to redo the work done by a transaction that commits.
  BEGIN/COMMIT/ABORT: records are recorded whenever a transaction begins, commits, or aborts.
Logging

- Recovery management is usually accomplished using a *log*.
- A log is a read/append data structure located in persistent storage (it must survive the failure)
- When transactions are running, *log records* are appended to the log. Log records contain:
  - **UNDO information**: old versions of objects that have been modified by a transaction. Used to undo database changes made by a transaction that aborts.
  - **REDO information**: new versions of objects that have been modified by a transaction. Used to redo the work done by a transaction that commits.
  - **BEGIN/COMMIT/ABORT**: records are recorded whenever a transaction begins, commits, or aborts.

**Requires Write-Ahead-Logging**
Log records must be written *before* updating the database!
• Recovering from a system failure:

1. Scan the log from tail (newest) to head (oldest):
   • Create a list of committed transactions
   • Undo updates of active and aborted transactions

2. Scan the log from head (oldest) to tail (newest):
   • Redo updates of committed transactions.

• Rolling back a single transaction:
  1. Scan the log from the tail to the transaction’s BEGIN record.
  2. Undo the transaction’s updates.
Recovery

- Recovering from a system failure:
  1. Scan the log from tail (newest) to head (oldest):
     - Create a list of committed transactions
     - Undo updates of active and aborted transactions
  2. Scan the log from head (oldest) to tail (newest):
     - Redo updates of committed transactions.
Recovering from a system failure:

1. Scan the log from tail (newest) to head (oldest):
   - Create a list of committed transactions
   - Undo updates of active and aborted transactions

2. Scan the log from head (oldest) to tail (newest):
   - Redo updates of committed transactions.
Recovery

• Recovering from a system failure:
  1. Scan the log from tail (newest) to head (oldest):
     • Create a list of committed transactions
     • Undo updates of active and aborted transactions
  2. Scan the log from head (oldest) to tail (newest):
     • Redo updates of committed transactions.

• Rolling back a single transaction:
Recovering from a system failure:

1. Scan the log from tail (newest) to head (oldest):
   - Create a list of committed transactions
   - Undo updates of active and aborted transactions

2. Scan the log from head (oldest) to tail (newest):
   - Redo updates of committed transactions.

Rolling back a single transaction:

1. Scan the log from the tail to the transaction’s BEGIN record.
   - Undo the transaction’s updates.