Lecture 11

Database Recovery

Dr. Vandana Kushwaha

Department of Computer Science

Institute of Science, BHU, Varanasi

Why Recovery Is Needed

- Whenever a **transaction** is submitted to a **DBMS** for execution, the system is responsible for making sure that :
  - **Either all** the operations in the transaction are **completed successfully** and their effect is recorded permanently in the database(**Transaction committed**).

    OR
  - That the transaction does not have any effect on the database or any other transactions. (**Transaction aborted**)
- If a transaction **fails** after executing some of its operations but before executing all of them, the **operations already executed** must be **undone** to **ensure the Atomicity.**

Types of Failures

- There are **several possible reasons** for a **transaction to fail** in the middle of execution:
- **A Computer failure (system crash)**
  - A **hardware, software**, or **network error** occurs in the computer system during transaction execution.
- **A Transaction error**
  - Some operation in the transaction may cause it to fail, such as **integer overflow** or **division by zero.**
  - **Transaction failure** may also occur because of **erroneous parameter values** or because of a **logical programming error.**
  - Additionally, the **user may interrupt** the **transaction** during its execution.

Types of Failures

- **Local errors or exception conditions detected by the transaction**
  - During transaction execution, **certain conditions** may occur that necessitate cancellation of the transaction.
  - For example**, data** for the transaction may **not be found.**
  - An **exception condition**, such as **insufficient account balance** in a **banking database**, may cause a transaction, such as a fund withdrawal, to be canceled.
  - This **exception** could be programmed in the transaction itself, and in such a case would not be considered as a transaction failure.

Types of Failures

- **Failure due to Concurrency control enforcement**
  - The **concurrency control method** may decide to **abort a transaction** because it violates **serializability**, or it may abort one or more transactions to resolve a state of **deadlock** among several **transactions**.
  - **Transactions aborted** because of **serializability violations** or **deadlocks** are typically **restarted automatically** at a later time.
- **Disk failure**
  - Some **disk blocks** may **lose their data** because of a **read or write malfunction** or because of a **disk read/write head crash.**
  - This may happen during a **read or a write operation** of the transaction.

Types of Failures

- **Physical problems and catastrophes**
- This refers to an endless list of problems that includes:
  - Power or air-conditioning failure,
  - Fire,
  - Theft,
  - Sabotage,
  - Overwriting disks or tapes by mistake etc.

Recovery and Atomicity

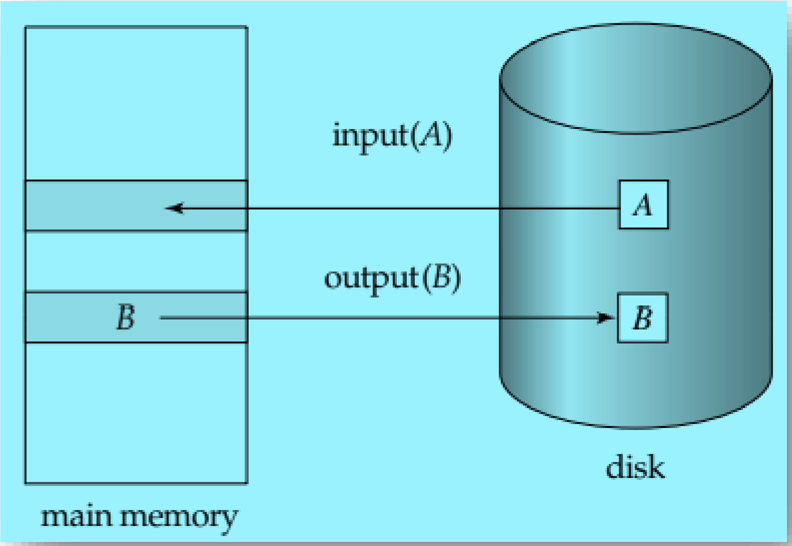
- Consider **transaction Ti** that transfers **$50** from **account A** to **account B.**
- **Atomicity goal** is either to perform all database modifications made by **Ti** or none at all.
- Multiple **output(write) operations** may be required for **Ti** (to output A and B).
- A **failure** may occur after one of these modifications have been made but before all of them are made.
- To **ensure Atomicity despite failures**, we first **output information** describing the modifications **to stable storage without modifying the database itself.**
- There are **two approaches** of **Database recovery: –**
  - **Log-based Recovery**
  - **Shadow-paging**
- We assume (initially) that transactions run serially, that is, one after the other.

Data Access

- The **database system** resides **permanently** on **nonvolatile storage** (usually **disks**), and is partitioned into fixed-length storage units called **blocks.**
- **Blocks** are the **units of data transfer** to and from **disk**, and may contain several data items.
- **Transactions** **input** information **from** the **disk to main memory**, and then **output** the **information back onto the disk.**
- The **input** and **output operations** are done in **block units.**
- The **blocks** residing on the **disk** are **referred** to as **Physical blocks**;
- The **blocks residing temporarily** in **main memory** are referred to as **Buffer blocks**.

Data Access

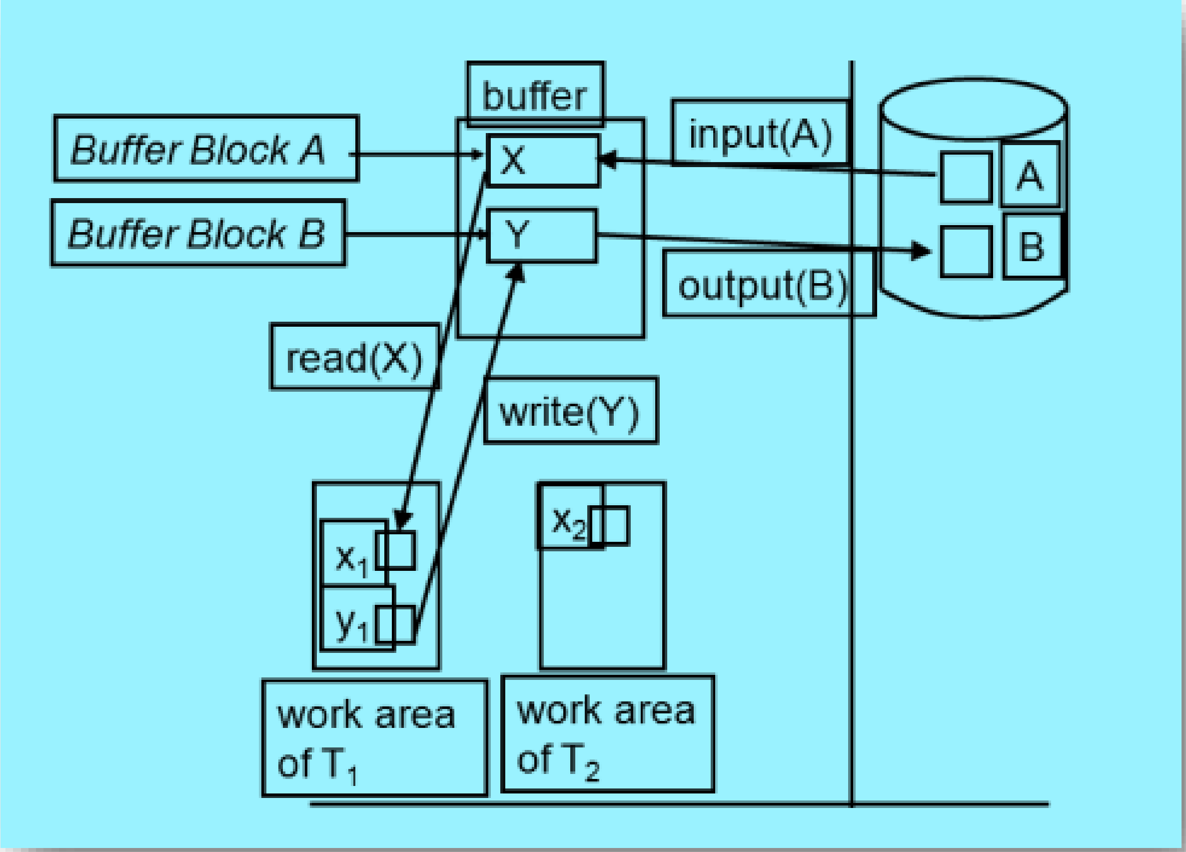
- **Block movements** between **disk and main memory** are initiated through the following **two operations:**
- **1. input(A)** transfers the **physical block A** to **main memory.**
- 2. **output(B)** transfers the **buffer block B** to the **disk,** and **replaces** the appropriate **physical block** there.



Data Access

- Each **transaction Ti** has a **private work area**(in main memory) in which **copies** of **all the data items** accessed and updated by **Ti** are **kept.**
- The **system creates** this **work area** when the **transaction is initiated**; the **system removes it** when the **transaction either commits or aborts.**
- Each **data item X** kept in the **work area** of **transaction Ti** is denoted by **Xi.**
- **Transaction Ti** interacts with the **database system** by **transferring data** to and from its **work area** to the **system buffer(main memory)** using **read** and **write** operations.

Data Access



buffer
Buffer Block A
Buffer Block B
X
Y
input(A)
output(B)
A
B
read(X)
write(Y)
x_1
y_1
x_2
work area of T_1
work area of T_2

Log-Based Recovery

- The most widely used structure for **recording database modifications** is the **log.**
- The **log** is a **sequence of log records**, recording all the **update activities** in the **database.**
- There are **several types** of **log records:**
  - **<Ti start>** Transaction Ti has **started.**
  - **<Ti, Xj, V1, V2>** Transaction **Ti** has performed a **write** on data item **Xj** . Xj had value **V1** before the write, and will have value **V2** after the write.
  - **<Ti commit>** Transaction **Ti** has **committed.**
  - **<Ti abort>** Transaction **Ti** has **aborted.**
- Whenever a **transaction performs a write,** it is **essential** that the **log record for that write** be **created before** the **database** is **modified.**
- For **log records** to be useful for **recovery** from system and disk failures, the **log must reside in stable storage.**

Deferred Database Modification

- The **Deferred-modification technique** ensures **transaction atomicity** by **recording all database modifications** in the **log**, but **deferring** the **execution** of **all write operations** of a transaction **until** the **transaction partially commits**.
- A **transaction** is said to be **partially committed** once the **final action** of the **transaction** has been **executed.**
- If the **system crashes before** the **transaction completes** its **execution,** or if the **transaction aborts**, then **the information on the log** is **simply ignored.**
- Before **Ti starts** its execution, a record **<Ti start>** is written to the **log.**
- A **write(X)** operation by **Ti** results in the writing of a new record **<Ti, Xj, V1, V2>** to the **log.**
- Finally, when **Ti partially commits**, a record **<Ti commit>** is written to the **log.**
- When transaction **Ti partially commits**, the records associated with it in the **log** are **used** in **executing** the **deferred writes.**

Deferred Database Modification

- Since a **failure may occur** while this updating is taking place, **we must ensure** that, **before** the **start of these updates**, **all the log records are written out to stable storage.**
- Once **they have been written**, the **actual updating takes place**, and **the transaction enters the committed state.**
- Using the **log,** the **system can handle any failure** that results in the **loss of information on volatile storage.**
- The **recovery scheme** uses the **following recovery procedure**:
- **redo(Ti)** - **sets** the **value** of **all data items updated by transaction Ti** to the **new values.**
- The **set of data items updated** by **Ti** and their **respective new values** can be **found in the log.**

Deferred Database Modification

- After a failure, the **recovery subsystem** consults the **log** to **determine** which **transactions need to be redone.**
- **Transaction Ti needs** to be **redone if and only if** the **log contains both the record <Ti start>** and the record **<Ti commit>.**
- Thus, if the **system crashes** after the **transaction completes its execution**, the **recovery scheme uses the information** in the **log to restore the system** to a **previous consistent state** after the transaction had completed.

Deferred Database Modification

- Let **T0** be a transaction that transfers \$50 from account A to account B:

T_0: read(A);
$A := A - 50$;
write(A);
read(B);
$B := B + 50$;
write(B).

T_1: read(C);
$C := C - 100$;
write(C).

- Let **T1** be a transaction that withdraws \$100 from account C.
- Suppose that these transactions are **executed serially**, in the order T0 followed by T1.
- Let the values of accounts A, B, and C before the execution took place were \$1000, \$2000, and \$700, respectively.
- The portion of the **log** containing the relevant information on these two transactions.

<T_0 start>
<T_0, A, 950>
<T_0, B, 2050>
<T_0 commit>
<T_1 start>
<T_1, C, 600>
<T_1 commit>

Deferred Database Modification

- State of the log and database corresponding to T0 and T1.

Log	Database
$<T_0$ start$>$	
$<T_0, A, 950>$	
$<T_0, B, 2050>$	
$<T_0$ commit$>$	
	$A = 950$
	$B = 2050$
$<T_1$ start$>$	
$<T_1, C, 600>$	
$<T_1$ commit$>$	
	$C = 600$

- **CASE 1:** Assume that the crash occurs just after the log record for the step **write(B)** of transaction **T0** has been written to stable storage.
- The log at the time of the crash appears in Figure :

$<T_0$ start$>$
$<T_0, A, 950>$
$<T_0, B, 2050>$

- When the system comes back up, **no redo** actions need to be taken, since **no commit** record appears in the log.
- The values of accounts A and B remain \$1000 and \$2000, respectively.
- The log records of the incomplete **transaction T0 can be deleted** from the **log.**

Deferred Database Modification

- **CASE 2:**
- Now, let us assume the crash comes just after the log record for the step **write(C)** of transaction **T1** has been written to stable storage.
- In this case, the log at the time of the crash is as in Figure.

<T_0 start>
<T_0, A, 950>
<T_0, B, 2050>
<T_0 commit>
<T_1 start>
<T_1, C, 600>

- When the system comes back up, the **operation redo(T0) is performed**, since the record appears in the log on the disk.
- After this operation is executed, the values of accounts A and B are $950 and $2050, respectively.
- The value of account **C** remains $700.
- As before, the log records of the **incomplete transaction T1** can be deleted from the log.

Deferred Database Modification

- **CASE 3:**
- Finally, assume that a crash occurs just after the log record is written to **stable storage.**
- The log at the time of this crash is as in Figure.

<T_0 start>
<T_0, A, 950>
<T_0, B, 2050>
<T_0 commit>
<T_1 start>
<T_1, C, 600>
<T_1 commit>

- When the system comes back up, **two commit records are in the log**: one for T0 and one for T1.
- Therefore, the system must perform operations **redo(T0) and redo(T1),** in the order in which their commit records appear in the log.
- After the system executes these operations, the values of accounts A, B, and C are \$950, \$2050, and \$600, respectively.

Immediate Database Modification

- The **immediate-modification technique** allows **database modifications** to be **output to the database** while the **transaction is still** in the **active state.**
- **Data modifications** written by **active transactions** are called **uncommitted modifications.**
- In the event of a **crash** or a **transaction failure**, the **system** must **use the old-value field of the log records** to **restore the modified data items** to the value they had prior to the start of the transaction.
- The **Undo operation** accomplishes this **restoration.**
- Before a **transaction Ti starts** its execution, the system writes the record **<Ti start>** to the log.
- During its execution, any **write(X)** operation by Ti is preceded by the writing of the appropriate new update record to the **log.**
- When **Ti partially commits**, the **system writes** the **record<Ti commit>** to the **log.**

Immediate Database Modification

- Before execution of an **output(B) operation**, the **log records corresponding to B** be **written onto stable storage.**
- Consider the transactions T0 and T1 and the portion of the system log corresponding to T0 and T1:

<T_0 start>
<T_0, A, 1000, 950>
<T_0, B, 2000, 2050>
<T_0 commit>
<T_1 start>
<T_1, C, 700, 600>
<T_1 commit>

- The state of system log and database corresponding to T0 and T1 is:

Log	Database
<T_0 start>	
<T_0, A, 1000, 950>	
<T_0, B, 2000, 2050>	
	$A = 950$
	$B = 2050$
<T_0 commit>	
<T_1 start>	
<T_1, C, 700, 600>	
	$C = 600$
<T_1 commit>	

Immediate Database Modification

- Using the log, the system can handle any failure that does not result in the loss of information in nonvolatile storage.
- The recovery scheme uses **two recovery procedures:**
  - **undo(Ti)** restores the value of all data items updated by transaction Ti to the old values.
  - **redo(Ti)** sets the value of all data items updated by transaction Ti to the new values.
  - The set of data items updated by Ti and their respective old and new values can be found in the **log.**
- After a failure has occurred, the **recovery scheme** consults the log to determine which **transactions** need to be redone, and which need to be undone:
  - **Transaction Ti** needs to be **undone** if the log contains the record **<Ti start>** , but does not contain the record **<Ti commit>.**
  - **Transaction Ti** needs to be **redone** if the log contains both the record **<Ti start>** and **<Ti commit>** the record .

Immediate Database Modification

- Suppose that the **system crashes** before the completion of the transactions.
- **Case 1:** the crash occurs just after the log record for the step write(B) of transaction T0 has been written to stable storage.

<T_0 start>
<T_0, A, 1000, 950>
<T_0, B, 2000, 2050>

- When the system comes back up, it finds the record <T0 start> in the log, but no corresponding <T0 commit> record.
- Thus, transaction T0 must be undone, so an **undo(T0)** is performed.
- As a result, the values in accounts A and B (on the disk) are restored to $1000 and $2000, respectively.

Immediate Database Modification

- **Case 2:** Next, let us assume that the crash comes just after the log record for the step write(C) of transaction T1 has been written to stable storage.

<T_0 start>
<T_0, A, 1000, 950>
<T_0, B, 2000, 2050>
<T_0 commit>
<T_1 start>
<T_1, C, 700, 600>

- When the system comes back up, two recovery actions need to be taken.
- The operation **undo(T1) must be performed**, since the record appears in the log, but there is no record .
- The operation **redo(T0) must be performed**, since the log contains both the record and the record .
- At the end of the entire recovery procedure, the values of accounts A, B, and C are \$950, \$2050, and \$700, respectively.

Immediate Database Modification

- **Case 3:** let us assume that the crash occurs just after the log record <T1 commit> has been written to stable storage.

<T_0 start>
<T_0, A, 1000, 950>
<T_0, B, 2000, 2050>
<T_0 commit>
<T_1 start>
<T_1, C, 700, 600>
<T_1 commit>

- When the system comes back up, **both T0 and T1 need to be redone.**
- Since the records <T0 start>and <T0 commit> appear in the log, as do the records <T1 start> and <T1 commit> .
- After the system performs the recovery procedures redo(T0) and redo(T1), the values in accounts A, B, and C are $950, $2050, and $600, respectively.

Immediate Database Modification

- Notice that during **Immediate Database Modification** it is **not a requirement** that **every update** be **applied immediately** to disk; it is just possible that some updates are applied to disk before the transaction commits.
- Theoretically, we can distinguish **two main categories** of **immediate update algorithms.**
  - If the **recovery technique** ensures that all updates of a transaction are recorded in the database on disk *before the transaction commits,* there is **never a need** to **REDO** any **operations** of **committed transactions.**
  - This is called the **UNDO/NO-REDO recovery algorithm**.
  - If the **recovery technique** found that **only few updates** of a **transaction** are recorded in the database on disk *before the transaction commits,* there is a **need to REDO** all the **operations** of **committed transactions**.
  - This is called the **UNDO/REDO recovery algorithm**.

Checkpoints

- In **Log-based Recovery techniques** when a **system failure** occurs, we must consult the **log** to determine those **transactions** that **need to be redone** and those that need to be **undone.**
- We need to **search the entire log** to determine this information which is **time consuming.**
- To **reduce** these types of **overhead**, **Checkpoints** were proposed.
- During **Transaction** execution, the system maintains the **log,** using one of the two techniques described earlier.
- **In addition**, the **system periodically performs checkpoints**, which require the following **sequence of actions** to take place:
  - **1. Output** onto **stable storage** all **log records** currently residing in **main memory.**
  - **2. Output** to the **disk all modified buffer blocks.**
  - **3. Output** onto **stable storage** a log record **<checkpoint>.**

Checkpoints

- **Transactions** are **not allowed** to perform **any update actions**, such as **writing** to a buffer block or writing a log record, while a **checkpoint is in progress.**
- Consider a **transaction Ti** that **committed prior** to the **checkpoint.**
- For such a transaction, the **<Ti commit>** record appears in the log before the record **<checkpoint>.**
- Any **database modifications** made by **Ti** must have been **written to the database** either **prior to the checkpoint** or as part of the checkpoint itself.
- Thus, at **recovery time**, there is **no need to perform** a **redo operation on Ti.**

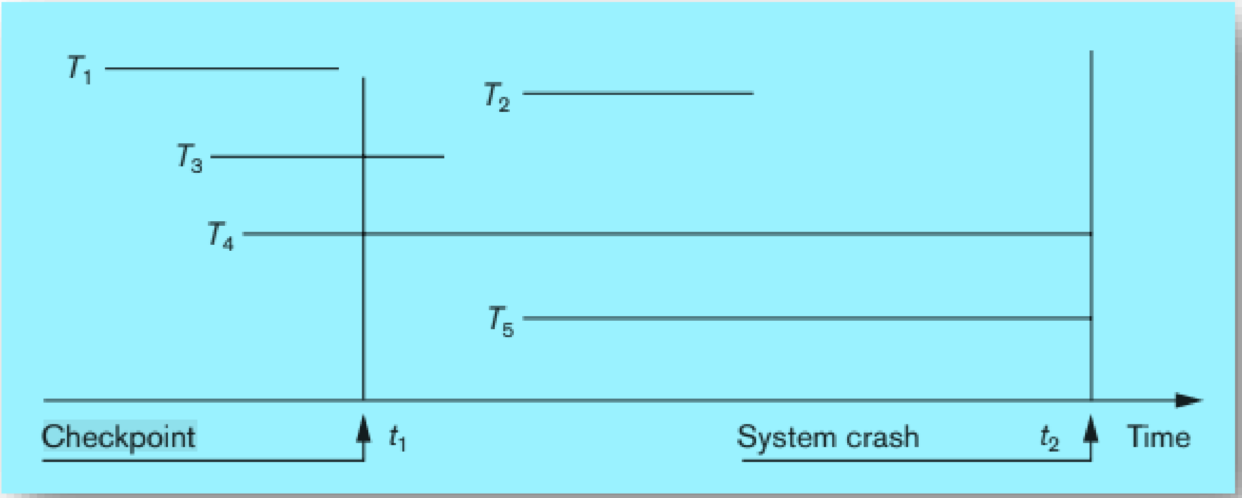
Checkpoints

- After a **failure has occurred**, the **recovery scheme** examines the **log to determine the most recent transaction Ti** that **started executing before the most recent checkpoint took place.**
- It can find such a **transaction** by :
  - **Searching the log backward**, from the end of the **log**, until it finds the **first <checkpoint> record**;
  - Then it continues the search **backward** until it **finds the next <Ti start>** record.
- Once the system has identified **transaction Ti**, the **redo and undo operations** need to be **applied to only transaction Ti** and **all transactions Tj that started executing after transaction Ti.**

Example 1: Checkpoints

- Consider the set **of transactions {T0, T1, . . ., T100}** executed in the order of the subscripts.
- Suppose that the **most recent checkpoint** took place during the execution of **transaction T67.**
- Thus, **only transactions T67, T68, . . ., T100 need to be considered** during the **recovery scheme.**
- Each of them needs to be **redone if it has committed**; otherwise, **it needs to be undone.**

Example 2: Checkpoints



- When the **checkpoint** was taken at **time t1**, **transaction T1** had **committed,** whereas **transactions T3 and T4** had **not.**
- Before the **system crash** at **time t2**, **T3 and T2** were **committed** but **not T4 and T5.**
- According to the **Deferred Database Modification** method, there is **no need to redo transaction T1**—or any **transactions committed before the last checkpoint time t1.**

Checkpoints

- The **transactions T2 and T3 must be redone**, however, because both transactions reached their commit points after the last checkpoint.
- **Transactions T4 and T5 are ignored:** They are effectively **canceled or rolled back** because none of their write operations were recorded in the database on disk under the **deferred update protocol.**