Recovery System

Recovery System

- An integral part of a database system is a **recovery scheme** that can restore the database to the consistent state that existed before the failure.

- The recovery scheme must also provide **high availability**; that is, it must ensure that within minimum time the database should be available after failure.

Failure Classification

A Failure is an event at which the system does not perform according to specifications.

1. **Transaction failure**. There are two types of errors that may cause a transaction to fail:

   - **Logical error**. The transaction can no longer continue with its normal execution because of some internal condition, such as bad input, data not found, overflow, or resource limit exceeded.

   - **System error**. The system has entered an undesirable state (for example, deadlock), as a result of which a transaction cannot continue with its normal execution. The transaction, however, can be re -executed at a later time.

Failure Classification

2. **System crash**.
   - There is a hardware malfunction, or a bug in the database software or the operating system, that causes the loss of the content of volatile storage, and brings transaction processing to a halt.
   - The content of nonvolatile storage remains intact, and is not corrupted. (Fail Stop Assumption)

3. **Disk failure**. A disk block loses its content as a result of either a head crash or failure during a data-transfer operation. Copies of the data on other disks, or archival backups on tertiary media, such as DVD or tapes, are used to recover from the failure.

Determination of Recovery

- The determination of how the system should recover from failures, depends on failure modes of those devices used for storing data.
- Based on such failures, recovery manager employs algorithm which takes care of the following things:

  - Actions taken during normal transaction processing to ensure that enough information exists to allow recovery from failures.
  - Actions taken after a failure to recover the database contents to a state that ensures database consistency, transaction atomicity, and durability.

Recovery & Atomicity

1. Log Records:

- 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. An **update log record** describes a single database write. It has these fields:
  - **Transaction identifier**: the unique identifier of the transaction that performed the write operation.

Recovery & Atomicity

- **Data-item identifier**: the unique identifier of the data item written. (the location on disk of the data item, consisting of the block identifier of the block on which the data item resides, and an offset within the block)
- **Old value**, which is the value of the data item prior to the write.
- **New value**, which is the value that the data item will have after the write.

Recovery & Atomicity

- We represent an update log record as <*Ti* , *Xj* , *V*1, *V*2>, indicating that transaction *Ti* has performed a write on data item *Xj* . *Xj* had value *V*1 before the write, and has value *V*2 after the write.
- Other special log records exist to record significant events during transaction processing, such as the start of a transaction and the commit or abort of a transaction.

Among the types of log records are:

- ✓ <*Ti* start> - Transaction *Ti* has started.
- ✓ <*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 and added to the log, 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.

Log Records and Database Modification

- The log records allow the system to undo changes made by a transaction in the event that the transaction must be aborted;
- They allow the system also to redo changes made by a transaction if the transaction has committed but the system crashed before those changes could be stored in the database on disk.
- We need to consider the steps a transaction takes in modifying a data item:

1. The transaction performs some computations in its own private part of main memory.

2. The transaction modifies the data block in the disk buffer in main memory holding the data item.

3. The database system executes the output operation that writes the data block to disk.

Log Records and Database Modification

- Consider our simplified banking system.

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

$T0$: read(A);
$A := A - 50$;
write(A);
read(B);
$B := B + 50$;
write(B).

Let $T1$ be a transaction that withdraws \$100 from account C:

$T1$: read(C);
$C := C - 100$;
write(C).

Log Records and Database Modification

- The following figure shows the log records of the combined effect of both the transactions:

<T0 start>
<T0 , A, 1000, 950>
<T0 , B, 2000, 2050>
<T0 commit>
<T1 start>
<T1 , C, 700, 600>
<T1 commit>

Log Records and Database Modification

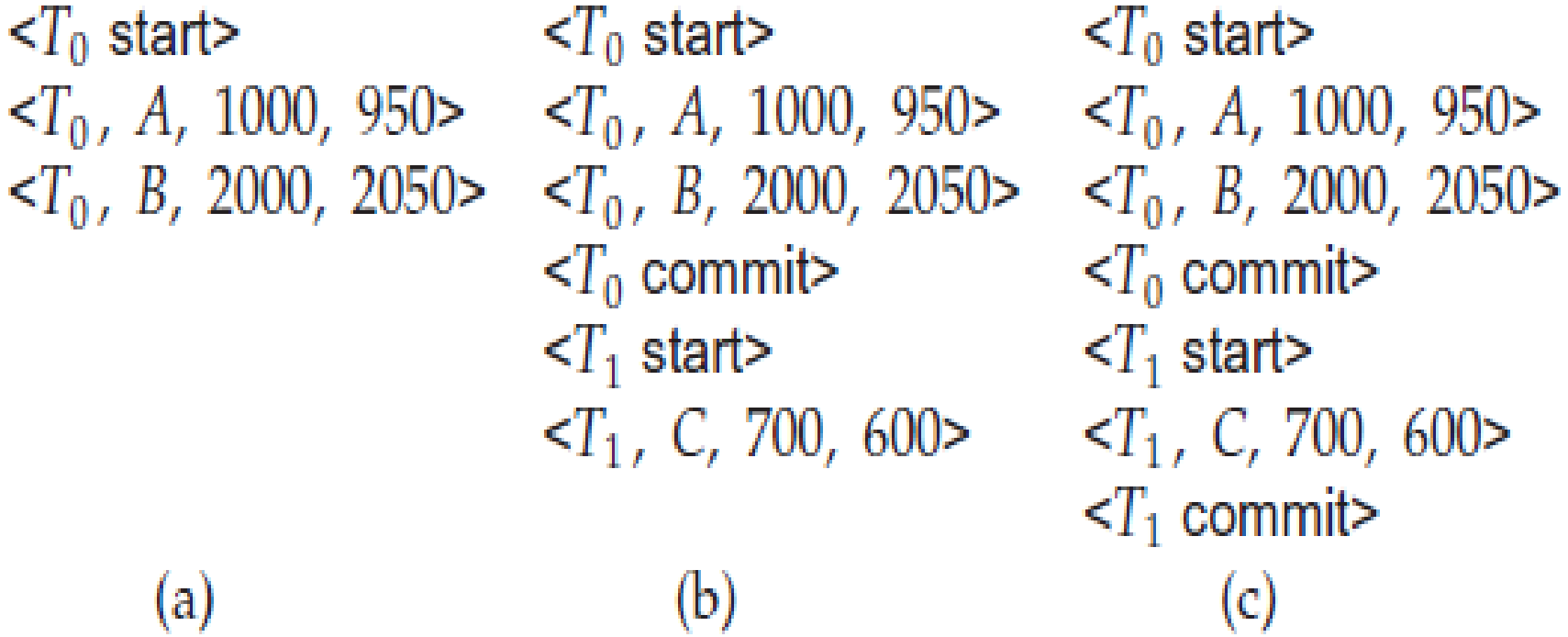
- After a system crash has occurred, the system consults the log to determine which transactions need to be redone, and which need to be undone so as to ensure atomicity.

1. Transaction *Ti* needs to be undone if the log contains the record <*Ti* start>, but does not contain either the record <*Ti* commit> or the record <*Ti* abort>.

2. Transaction *Ti* needs to be redone if the log contains the record<*Ti* start>and either the record <*Ti* commit> or the record <*Ti* abort>. It may seem strange to redo *Ti* if the record <*Ti* abort> is in the log. To see why this works, note that **if <*Ti* abort> is in the log, so are the redo-only records written by the undo operation.** Thus, the end result will be to undo *Ti* 's modifications in this case.

Log Records and Database Modification

- with transaction $T0$ and $T1$ executed one after the other in the order $T0$ followed by $T1$. Suppose that the system crashes before the completion of the transactions.
- initial balance of T0 and T1 are $1000 and $2000 respectively



Checkpoints

- The log based recovery approach has following issues:

1. The search process is time-consuming.
2. Most of the transactions that, according to our algorithm, need to be redone have already written their updates into the database. Although redoing them will cause no harm, it will cause recovery to take longer.

- To reduce these types of overhead, we introduce another type of entry in the log called a **checkpoint**.
- A [checkpoint, list of active transactions] record is written into the log periodically at that point when the system writes out to the database on disk all DBMS buffers that have been modified.
- all transactions that have their [commit, T] entries in the log before a [checkpoint] entry do not need to have their WRITE operations redone in case of a system crash.

CHECKPOINTS

- The recovery manager of a DBMS must decide at what intervals to take a checkpoint.
- The interval may be measured in time—say, every m minutes—or in the number **t** of committed transactions since the last checkpoint, where the values of **m** or t are system parameters.
- Taking a checkpoint consists of the following actions:

1. Suspend execution of transactions temporarily.
2. Force-write all main memory buffers that have been modified to disk.
3. Write a [checkpoint] record <checkpoint L> (where L is a list of transactions active at the time of the checkpoint) to the log, and force-write the log to disk.
4. Resume executing transactions.

Checkpoints

- After a system crash has occurred, the system examines the log to find the last <checkpoint L> record.
- The redo or undo operations need to be applied only to transactions in L, and to all transactions that started execution after the <checkpoint L> record was written to the log.
- Let us denote this set of transactions as T:

1. For all transactions Tk in T that have no <Tk commit> record or <Tk abort> record in the log, execute undo(Tk).
2. For all transactions Tk in T such that either the record <Tk commit> or the record <Tk abort> appears in the log, execute redo(Tk).

Checkpoints

- Example: Consider the set of transactions {T0, T1, . . . , T100}. Suppose that the most recent checkpoint took place during the execution of transaction T67 and T69, while T68 and all transactions with subscripts lower than 67 completed before the checkpoint. Thus, only transactions T67, T69. . ., T100 need to be considered during the recovery scheme. Each of them needs to be redone if it has
- completed (that is, either committed or aborted); otherwise, it was incomplete, and needs to be undone.

Concept of Shadow Paging

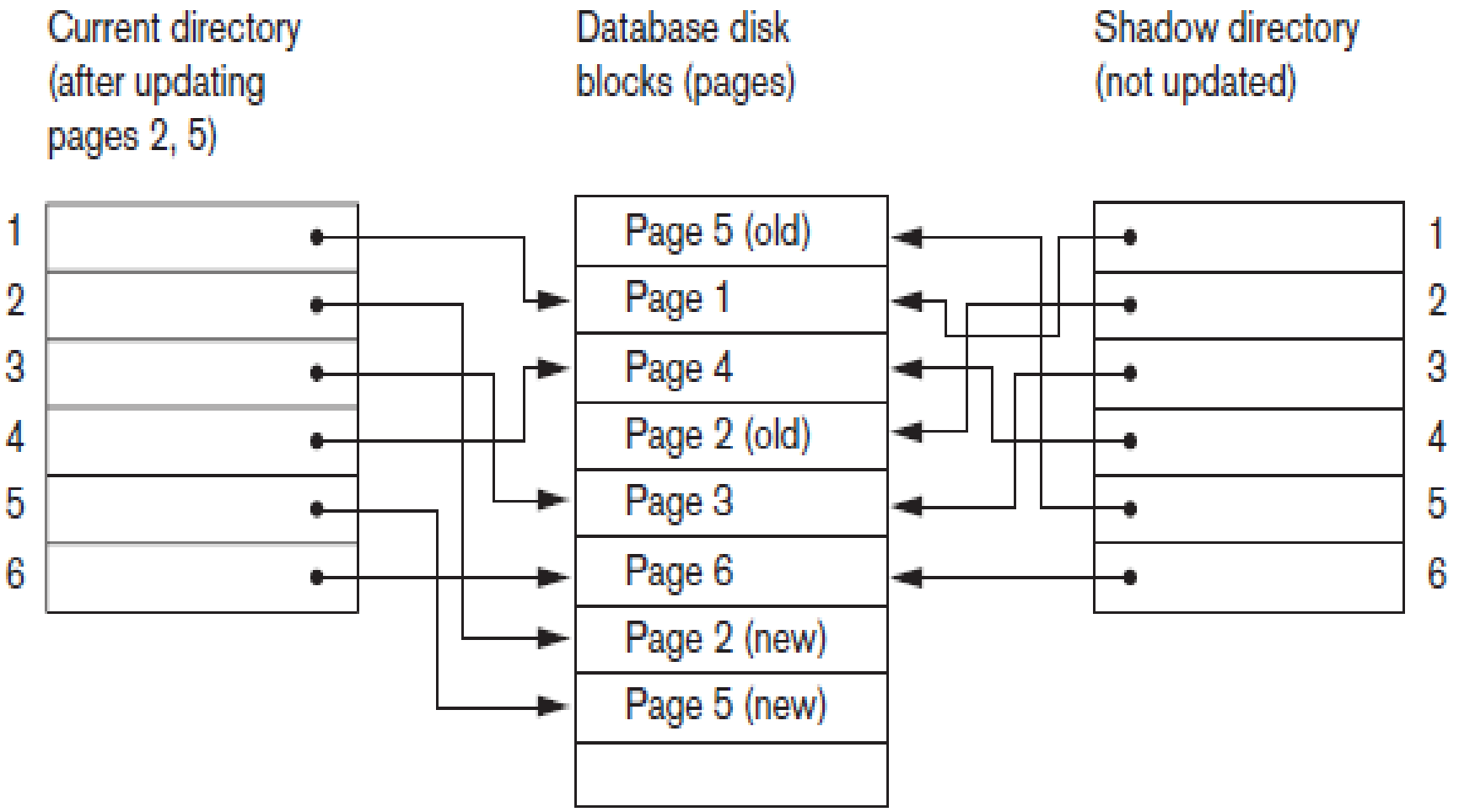
- This recovery scheme does not require the use of a log in a single-user environment. In a multiuser environment, a log may be needed for the concurrency control method.
- ✓ considers the database to be made up of a number of fixed size disk pages (or disk blocks)—say, n—for recovery purposes
- ✓ A directory with n entries is constructed, where the i^{th} entry points to the i^{th} database page on disk. The directory is kept in main memory if it is not too large, and all references—reads or writes—to database pages on disk go through it.
- ✓ When a transaction begins executing, the current directory—whose entries point to the most recent or current database pages on disk—is copied into a shadow directory

Concept of Shadow Paging

- The shadow directory is then saved on disk while the current directory is used by the transaction.
- During transaction execution, the shadow directory is never modified.
- When a write_item operation is performed, a new copy of the modified database page is created, but the old copy of that page is not overwritten. Instead, the new page is written elsewhere—on some previously unused disk block.
- The current directory entry is modified to point to the new disk block, whereas the shadow directory is not modified and continues to point to the old unmodified disk block.

Concept of Shadow Paging

- **The old version is referenced by the shadow directory and the new version by the current directory**



Recovery using Shadow Paging

- Issues in Shadow Paging:
- 1. To recover from a failure (**not committed**) during transaction execution, it is sufficient to free the modified database pages and to discard the current directory. The state of the database before transaction execution is available through the shadow directory.
- **Committing** a transaction corresponds to discarding the previous shadow directory.
- Since recovery involves neither undoing nor redoing data items, this technique can be categorized as a **NOUNDO/ NO-REDO** technique for recovery.

Recovery using Shadow Paging

- To recover from a failure (**not committed**) during transaction execution, it is sufficient to free the modified database pages and to discard the current directory. The state of the database before transaction execution is available through the shadow directory.
- **Committing** a transaction corresponds to discarding the previous shadow directory.
- Since recovery involves neither undoing nor redoing data items, this technique can be categorized as a **NOUNDO/ NO-REDO** technique for recovery.