Unit 9

Crash Recovery

- ✓ A computer system, like any other device, is subject to failure from a variety of causes: disk crash, power outage, software error, fire etc. In any failure, information may be lost.
- ✓ Therefore, the database system must take actions in advance to ensure that the atomicity and durability properties of transactions are preserved.
- ✓ 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.

Failure classification

There are various types of failure that may occur in a system, each of which needs to be dealt with in a different manner. Some major types of failure are as follows:

- ❖ **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.
- ❖ **System crash**
  - ➢ A power failure or other hardware or software failure causes the system to crash.
  - ➢ **Fail-stop assumption**: non-volatile storage contents are assumed to not be corrupted by system crash.
  - ➢ Database systems have numerous integrity checks to prevent corruption of disk data.
- ❖ **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 tertiary media, such as DVD or tapes, are used to recover from the failure.

Storage structure

Various data items in the database may be stored and accessed in a number of different storage media. We identified three categories of storage.

- Volatile storage
- Nonvolatile storage
- Stable storage

Volatile storage

- ✓ Data residing in volatile storage does not survive system crashes
- ✓ Examples: main memory, cache memory

Nonvolatile storage

- ✓ Data residing in non-volatile storage survives system crashes
- ✓ Examples: disk, tape, flash memory, non-volatile RAM
- ✓ But may still fail ,losing data

Stable storage

- ✓ A mythical form of storage that survives all failures
- ✓ Approximated by maintaining multiple copies on distinct nonvolatile media.

Log and log records

- ✓ The log is a sequence of log records, recording all the updated activities in the database. In stable storage, logs for each transaction are maintained.
- ✓ Any operation which is performed on the database is recorded on the log.
- ✓ When transaction T_i starts, it registers itself by writing a <T_i **start**> log record
- ✓ Prior to performing any modification to the database, an updated log record is created to reflect that modification. An update log record represented as: <Ti, X, V1, V2> has these fields:

- **Transaction identifier (**T_i): Unique Identifier of the transaction that performed the write operation.
- **Data item (X):** Unique identifier of the data item written.
- **Old value (V1)**: Value of data item prior to write.
- **New value(V2):** Value of data item after write operation.

- ✓ When T_i finishes it last statement, the log record <T_i **commi**t> is written.
- ✓ We assume for now that log records are directly to stable storage (that is, they are not buffered)

Database modification

The database can be modified using two approaches:

1. Deferred database modification
2. Immediate database modification

1. Deferred database modification

- ✓ The deferred database modification scheme records all modifications to the log but defers all the writes to after partial commit.
- ✓ If the system crashes before the transaction completes its execution, or if the transaction abort, then the information on the log is simply ignored.
- ✓ Transactions starts by writing <T_i start> record to the log.
- ✓ A write(X) operations results in a log record < T_i,,X,V> being written, where V is the new value for X.
- ✓ The write is not performed on X at this time, but is deferred.
- ✓ When T_i Partially commits, < T_i, commit> is written to the log.
- ✓ Finally ,the log records are read and used to actually execute the previously deferred writes.

During recovery after crash, a transaction needs to be redone if and only if both <T_i start> and

<T_i commit> are there is the log.

Redoing a transaction T_i,(redo T_i) sets the value of all data items updated by the transaction to the new values.

(a)	(b)	(c)
<T_0 start>	<T_0 start>	<T_0 start>
<T_0, A, 950>	<T_0, A, 950>	<T_0, A, 950>
<T_0, B, 2050>	<T_0, B, 2050>	<T_0, B, 2050>
	<T_0 commit>	<T_0 commit>
	<T_1 start>	<T_1 start>
	<T_1, C, 600>	<T_1, C, 600>
		<T_1 commit>

If log on stable storage at time of crash is as in case:

a) No redo actions need to be taken

b) redo (T_0) must be performed since < T_0 **commit**> is present

c) redo (T_0) must be performed followed by redo (T_1) since < T_0 **commit**> < T_1 **commit**> are present.

If the transaction fails before reaching its commit point, it will have made no changes to database in any way so no UNDO operation is necessary. It may be necessary to REDO effect of the operations of commit transactions from the log because their effect may not have been recorded in the database. Therefore it is also known as **no undo** algorithm**.**

2. Immediate database modification

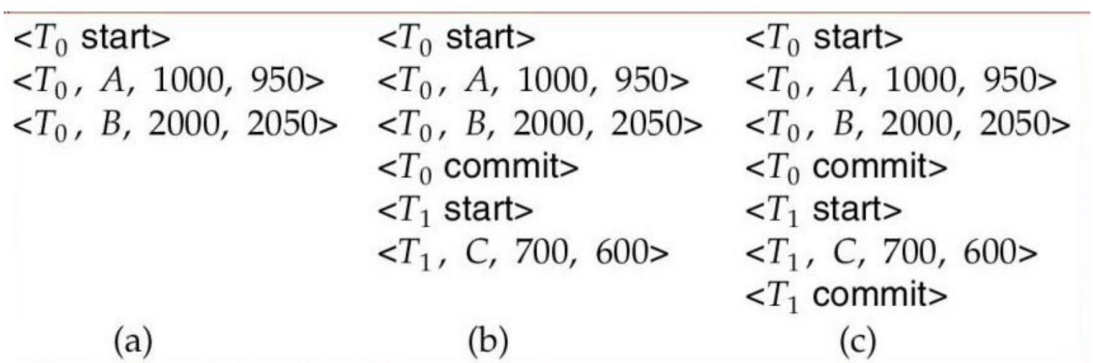
- ✓ Allows database modification while the transaction is still active.
- ✓ Which means all the modifications that is performed before the transaction reaches to commit state are updated to database.
- ✓ Database modifications written by active transactions are called uncommitted modifications.
- ✓ Update log must be written before database items is written.
- ✓ Transaction starts by writing <T_i start> record to log.
- ✓ A write(X) operations result in the log record <T_i X,V1,V2> where V1 is the old value and V2 is the new value . Since undoing may be needed, update logs must have both old value and new value.
- ✓ The write operation on X is recorded in log on disk and is output directly to stable storage without concerning transaction commits or not.
- ✓ In case of failure recovery procedure has two operations instead of one:
    1. undo (T_i) restores the value of all data items updated by T_i to their old values, going backwards from the last record for T_i.
    2. redo(T_i) sets the value of all data items updated by T_i to the new values, going forward from the first log record for T_i.

When recovering after failure:

- ✓ Transaction T_i needs to be undone if the log contains the record **<T_i start>** ,but does not contain the record **<T_i commit>**
- ✓ Transaction T_i needs to be redone if the log contains both the record **<T_i start>** and the record **<T_i commit>**
- ✓ Undo operations are performed first, then redo operations.

Example:

Below we show the log as it appears at three instances of time



Recovery actions in each case above are:

a) Undo (T_0):B is restored to 2000 and A to 1000

b) Undo (T_1) and redo (T_0) : C is restored to 700,and A and B are set to 950 and 2050 respectively.

c) Redo (T_0) and Redo (T_1): A and B is set to 950 and 2050 respectively. Then C is set to 600.

Undo and Redo transaction using log

Because all database modifications must be preceded by the creation of a log record, the system has available both the old value prior to the modification of the data item and new value that is to be written for data item. This allows system to perform redo and undo operations as appropriate:

Undo: using a log record sets the data item specified in log record to old value. (used for immediate database modification only)

Redo: using a log record sets the data item specified in log record to new value.

Checkpoint

The Checkpoint is used to declare a point before which the DBMS was in a consistent state, and all transactions were committed.

Use of Checkpoints

When a system crash occurs, user must consult the log. In principle, that need to search the entire log to determine this information. There are two major difficulties with this approach:

- ✓ The search process is time-consuming.
- ✓ 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, user introduce checkpoints.

A log record of the form <checkpoint L> is used to represent a checkpoint in log where L is a list of transactions active at the time of the checkpoint.

When a checkpoint log record is added to log all the transactions that have committed before this checkpoint have <Ti commit> log record before the checkpoint record.

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

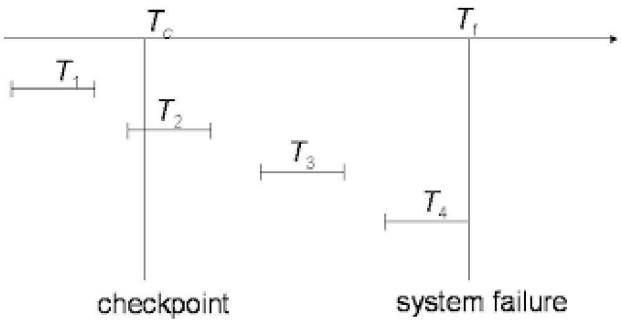
During recovery we need to consider only the most recent transaction T_i that started before the checkpoint, and transactions that started after T_i.

- ✓ Scan backwards from end of log to find the most recent <**checkpoint** *L*> record
- ✓ continue scanning backwards till a record **<T_i start>** is found.
- ✓ Need to consider the part of log following above **<T_i start>** record. Earlier part of log can be ignored during recovery .

After the transaction T_i identified, the redo and undo operations to be applied to the T_i and all T_j that started execution after transaction T_i .

For all transactions (starting from T_i or later)

- with no **<T_i commit>,** execute **undo** (T_i) *(Done only in case of immediate database modification)*
- with **<T_i commit>,** execute **redo** (T_i)



- ✓ T_1 can be ignored (updates already output to disk due to checkpoint)
- ✓ T_2 and T_3 redone
- ✓ T_4 undone

Concurrency control and recovery

Concurrency control means that multiple transactions can be executed at the same time and then the interleaved logs occur. But there may be changes in transaction results so maintain the order of execution of those transactions.

During recovery, it would be very difficult for the recovery system to backtrack all the logs and then start recovering.

Recovery with concurrent transactions can be done in the following four ways.

1. Interaction with concurrency control
2. Transaction rollback
3. Checkpoints
4. Restart recovery

1. Interaction with concurrency control:

The recovery scheme depends greatly on the concurrency-control scheme that is used. To roll back a failed transaction, we must undo the updates performed by the transaction. Suppose that a transaction T_0 has to be rolled back, and a data item Q that was updated by T_0 has to be restored to its old value. Using the log-based schemes for recovery, we restore the value by using the undo information in a log record.

Suppose now that a second transaction T_1 has performed yet another update on Q before T_0 is rolled back. Then, the update performed by T_1 will be lost if T_0 is rolled back.

Therefore, we require that, if a transaction T has updated a data item Q, no other transaction may update the same data item until T has committed or been rolled back. We can ensure this requirement easily by using strict two-phase locking—that is, two-phase locking with exclusive locks held until the end of the transaction.

2. Transaction rollback:

We roll back a failed transaction, Ti, by using the log. The system scans the log backward; for every log record of the form <Ti, Xj, V1, V2> found in the log, the system restores the data item Xj to its old value V1. Scanning of the log terminates when the log record <Ti, start> is found.

Scanning the log backward is important, since a transaction may have updated a data item more than once. As an illustration, consider the pair of log records

<T_1 start>
<T_1, A, 10, 20>
<T_1, A, 20, 30>

The log records represent a modification of data item A by T_1, followed by another modification of A by T_1. Scanning the log backward sets A correctly to 10.

3. Checkpoint

The checkpoint is used to declare the point before which the DBMS was in the consistent state, and all the transactions were committed. This reduces the amount of work during recovery.

By creating checkpoints, recovery operations don't need to start from the very beginning. Instead, they can begin from the most recent checkpoint, thereby considerably speeding up the recovery process.

Since we assumed no concurrency, it was necessary to consider only the following transactions during recovery:

- ✓ Those transactions that started after the most recent checkpoint
- ✓ The one transaction, if any, that was active at the time of the most recent checkpoint

The situation is more complex when transactions can execute concurrently, since several transactions may have been active at the time of the most recent checkpoint.

In a concurrent transaction-processing system, we require that the checkpoint log record be of the form <checkpoint L>, where L is a list of transactions active at the time of the checkpoint.

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.

4. Restart Recovery

When the system recovers from a crash, it constructs two lists: The undo-list consists of transactions to be undone, and the redo-list consists of transactions to be redone.

The system constructs the two lists as follows:

1. Initially, they are both empty.
2. The system scans the log backward, examining each record, until it finds the first <checkpoint> record, then
   i. For each record found of the form $<T_i$ commit>, it adds T_i to redo-list.
   ii. For each record found of the form $<T_i$ start>, if T_i is not in redo-list, then it adds T_i to undo-list.

COMMIT Operations Performance Optimization

During COMMITs the changes of the transaction are persisted to disk and made visible for other transactions.

Factors can influence the time of COMMIT operations such as Disk I/O write performance to logs, Synchronous system replication mode, Synchronous table replication, High amount of active versions, Integrated liveCache and SAP HANA bugs.

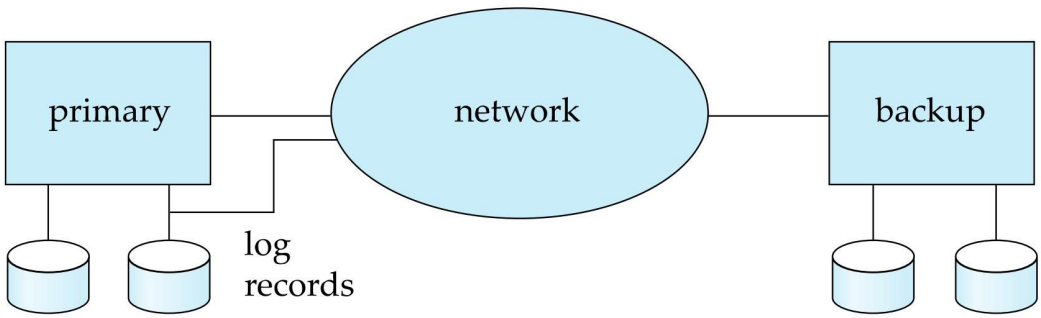
<table>
<tr><th>Reason</th><th>Details</th></tr>
<tr><td>Disk I/O write performance to logs</td><td>During COMMITs the change information is written to the log files. If writing down the data is slow, the COMMIT times can increase.</td></tr>
<tr><td>Synchronous system replication mode</td><td>If a synchronous system replication mode is activated, changes are propagated to the replication side during a COMMIT, and the local session has to wait for an acknowledgement. If you suffer from long COMMIT times in a system with activated synchronous system replication, you should at first check if a switch to a less critical system replication mode (e.g. SYNC -> SYNCMEM or SYNCMEM -> ASYNC) improve the performance. If yes, you can analyze if there are unnecessary replication delays, e.g. due to limited network bandwidth or high latency times.</td></tr>
<tr><td>Synchronous table replication</td><td>If a transaction performs changes on a table with activated optimistic synchronous table replication (OSTR), the replicas need to be refreshed during COMMIT. This can have an adverse impact on the COMMIT performance.</td></tr>
<tr><td>High amount of active versions</td><td>The COMMIT performance can be significantly impacted by a high amount of active versions.</td></tr>
<tr><td>Integrated liveCache</td><td>If a SAP HANA integrated liveCache is used, the SAP HANA COMMIT performance can be impacted by related COMMITs on liveCache side.
The time for the 'Kernel-Commit' method is part of the SAP HANA COMMIT time. Other related times like 'Flush-Cache', 'Commit-Invalidate-Callback' and 'Validate-Callback' have to be considered on top of the SAP HANA COMMIT time.
During a liveCache COMMIT a potentially large amount of data has to be flushed, so in case of high COMMIT times you should always consider the amount of flushed data to judge if it is more an I/O issue or a data volume issue.</td></tr>
<tr><td>SAP HANA bugs</td><td>The following SAP HANA bugs can be responsible for increased COMMIT times:
<table>
<tr><th>Impacted Revision</th><th>Details</th></tr>
<tr><td>122.062.00.000</td><td>Overhead in internal COMMIT processing design</td></tr>
</table>
</td></tr>
</table>

If COMMITs take very long, a termination with error **"snapshot timestamp synchronization failed"** is possible.

High Availability Using Remote Backup System

- ✓ Traditional transaction-processing systems are centralized or client–server systems. Such systems are vulnerable to environmental disasters such as fire, flooding, or earthquakes.
- ✓ So that there is a need for transaction-processing systems that can function in spite of system failures or environmental disasters and such systems must provide high availability.
- ✓ **We can achieve high availability** by performing transaction processing at one site, called the primary site, and having **a remote backup site** where all the data from the primary site are replicated. The remote backup site is sometimes also called the secondary site.
- ✓ The remote site must be kept synchronized with the primary site, as updates are performed at the primary.
- ✓ We achieve synchronization by sending all log records from primary site to the remote backup site. The remote backup site must be physically separated from the primary—for example, we can locate it in a different state—so that a disaster at the primary does not damage the remote backup site.

Figure below shows the architecture of a remote backup system.



- ✓ When the primary site fails, the remote backup site takes over processing. It performs recovery, using its copy of the data from the primary, and the log records received from the primary. In effect, the remote backup site is performing recovery actions that would have been performed at the primary site when the latter recovered.
- ✓ Once recovery has been performed, the remote backup site starts processing transactions.

Several issues must be addressed in designing a remote backup system:

Detection of failure: It is important for the remote backup system to detect when the primary has failed. Failure of communication lines can fool the remote backup into believing that the primary has failed. To avoid this problem, we maintain several communication links with independent modes of failure between the primary and the remote backup.

Transfer of control: When the primary fails, the backup site takes over processing and becomes the new primary. When the original primary site recovers, it can either play the role of remote backup, or take over the role of primary site again. In either case, the old primary must receive a log of updates carried out by the backup site while the old primary was down.

Time to recover: If the log at the remote backup grows large, recovery will take a long time. The remote backup site can periodically process the redo log records that it has received and can perform a checkpoint, so that earlier parts of the log can be deleted.

A hot-spare configuration can make takeover by the backup site almost instantaneous. In this configuration, the remote backup site continually processes redo log records as they arrive, applying the updates locally. As soon as the failure of the primary is detected, the backup site completes recovery by rolling back incomplete transactions; it is then ready to process new transactions.

Time to commit: To ensure that the updates of a committed transaction are durable, a transaction must not be declared committed until its log records have reached the backup site. This delay can result in a longer wait to commit a transaction, and some systems therefore permit lower degrees of durability.

The degrees of durability can be classified as follows:

One-safe: A transaction commits as soon as its commit log record is written to stable storage at the primary site.

Two-very-safe: A transaction commits as soon as its commit log record is written to stable storage at the primary and the backup site.

Two-safe: This scheme is the same as two-very-safe if both primary and backup sites are active. If only the primary is active, the transaction is allowed to commit as soon as its commit log record is written to stable storage at the primary site.