Recovery System

Failure Classification, Recovery and Atomicity, Recovery Algorithm, Buffer Management, Failure with Loss of Nonvolatile Storage, ARIES, Remote Backup Systems

Recovery System

Database recovery is the process of restoring the database to a consistent state in the event of a failure.

In other words, it is the process of restoring the database to the most recent consistent state that existed shortly before the time of system failure.

The failure may be the result of a system crash due to hardware or software errors, a media failure such as head crash, or a software error in the application such as a logical error in the program that is accessing the database.

Recovery System restores a database from a given state, usually inconsistent, to a previously consistent state.

Failure Classification

There are three types of failure that may occur in a system, each of which needs to be dealt with in a different manner.

1. **Transaction Failure**
2. **System Crash**
3. **Disk Failure**

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.

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.

   The assumption that hardware errors and bugs in the software bring the system to a halt, but do not corrupt the nonvolatile storage contents, is known as the **fail-stop assumption**. Well-designed systems have numerous internal checks, at the hardware and the software level that brings the system to a halt when there is an error. Hence, the fail-stop assumption is a reasonable one.

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.

Recovery and Atomicity

- When a system crashes, it may have several transactions being executed and various files opened for them to modify the data items.
- But according to ACID properties of DBMS, atomicity of transactions as a whole must be maintained, that is, either all the operations are executed or none.
- Database recovery means recovering the data when it gets deleted, hacked or damaged accidentally.

There are **six types** of techniques, which can help a DBMS in recovering as well as maintaining the atomicity of a transaction

1. Log Based Recovery
2. Database Modification
3. Concurrency Control and Recovery
4. Transaction Commit
5. Using the Log to Redo and Undo Transactions
6. Checkpoints

1. 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. An **update log record** describes a single database write. It has these fields:

- **Transaction Identifier**, which is the unique identifier of the transaction that performed the write operation.
- **Data-item Identifier**, which is the unique identifier of the data item written. Typically, it is 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.

We represent an update log record as

<Ti , Xj , V1, V2>,

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.*

2. Database Modification

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.

If a transaction does not modify the database until it has committed, it is said to use the **deferred-modification** technique.

If database modifications occur while the transaction is still active, the transaction is said to use the **immediate-modification** technique.

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 the new value that is to be written for the data item.

This allows the system to perform undo and redo operations as appropriate.

- **Undo** using a log record sets the data item specified in the log record to the old value.
- **Redo** using a log record sets the data item specified in the log record to the new value.

3. Concurrency Control and Recovery

If the concurrency control scheme allows a data item *X* that has been modified by a transaction *T*1 to be further modified by another transaction *T*2 before *T*1 commits, then undoing the effects of *T*1 by restoring the old value of *X* (before *T*1 updated *X*) would also undo the effects of *T*2.

To avoid such situations, recovery algorithms usually require that if a data item has been modified by a transaction, no other transaction can modify the data item until the first transaction commits or aborts.

This requirement can be ensured by acquiring an **exclusive lock** on any updated data item and holding the lock until the transaction commits; in other words, by using **strict two-phase locking**.

4. Transaction Commit

We say that a transaction has **committed** when it's commit log record, which is the last log record of the transaction, has been output to stable storage; at that point all earlier log records have already been output to stable storage. Thus, there is enough information in the log to ensure that even if there is a system crash, the updates of the transaction can be redone.

If a system crash occurs before a log record < *Ti* commit> is output to stable storage, transaction *Ti* will be rolled back. Thus, the output of the block containing the commit log record is the single atomic action that results in a transaction getting committed

5. Using the Log to Redo and Undo Transactions

We now provide an overview of how the log can be used to recover from a system crash, and to roll back transactions during normal operation.

Consider our simplified banking system.

Let *T*0 be a transaction that transfers $50 from account *A* to account *B*:
Let *T*1 be a transaction that withdraws $100 from account *C*:

<*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>

Portion of the system log corresponding to T0 and T1.

*T*0: read(*A*);
A := *A* − 50;
write(*A*);
read(*B*);
B := *B* + 50;
write(*B*).

Let *T*1 be a transaction that withdraws $100 from account *C*:

*T*1: read(*C*);
C := *C* − 100;
write(*C*).

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. Both these procedures make use of the log to find the set of data items updated by each transaction *Ti* , and their respective old and new values.

- **redo(*Ti*)** sets the value of all data items updated by transaction *Ti* to the new values.
- **undo(*Ti*)** restores the value of all data items updated by transaction *Ti* to the old values.

6. Checkpoints

A simple checkpoint scheme that

(a) Does not permit any updates to be performed while the checkpoint operation is in progress, and
(b) Outputs all modified buffer blocks to disk when the checkpoint is performed.

A checkpoint is performed as follows:

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 of the form <checkpoint *L*>, where *L* is a list of transactions active at the time of the checkpoint.

The presence of a <checkpoint *L*> record in the log allows the system to streamline its recovery procedure.

After a system crash has occurred, the system examines the log to find the last <checkpoint *L*> record (this can be done by searching the log backward, from the end of the log, until the first <checkpoint *L*> record is found).

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.

- For all transactions *Tk* in *T* that have no <*Tk* commit> record or <*Tk* abort> record in the log, execute **undo(*Tk*).**
- 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*).**

Recovery Algorithm

We have identified transactions that need to be redone and those that need to be undone, but we have not given a precise algorithm for performing these actions. We are now ready to present the full recovery algorithm using log records for recovery from transaction failure and a combination of the most recent checkpoint and log records to recover from a system crash.

The recovery algorithm described that a data item that has been updated by an uncommitted transaction cannot be modified by any other transaction, until the first transaction has either committed or aborted.

Two types of Recovery Algorithms while System Crash

1. **Transaction Rollback**
2. **Recovery after a System Crash**

1. Transaction Rollback

First consider transaction rollback during normal operation (that is, not during recovery from a system crash).

Rollback of a transaction Ti is performed as follows:

1. The log is scanned backward, and for each log record of *Ti* of the form <*Ti* , *Xj* , *V*1, *V*2> that is found:

a. The value *V*1 is written to data item *Xj*, and

b. A special redo-only log record<*Ti* , *Xj* , *V*1>is written to the log, where *V*1 is the value being restored to data item *Xj* during the rollback.

These log records are sometimes called **compensation log records**.

Such records do not need undo information, since we never need to undo such an undo operation. We shall explain later how they are used.

2. Once the log record <*Ti* start> is found the backward scan is stopped, and a log record <*Ti* abort> is written to the log.

Observe that every update action performed by the transaction or on behalf of the transaction, including actions taken to restore data items to their old value, have now been recorded in the log

2. Recovery after a System Crash

Recovery actions, when the database system is restarted after a crash, take place in two phases:

1. **Redo Phase**
2. **Undo Phase**

1. **Redo Phase: T**he system replays updates of *all* transactions by scanning the log forward from the last checkpoint. The log records that are replayed include log records for transactions that were rolled back before system crash, and those that had not committed when the system crash occurred.

   This phase also determines all transactions that were incomplete at the time of the crash, and must therefore be rolled back. Such incomplete transactions would either have been active at the time of the checkpoint, and thus would appear in the transaction list in the checkpoint record, or would have started later; further, such incomplete transactions would have neither a<*Ti* abort> nor a <*Ti* commit> record in the log.

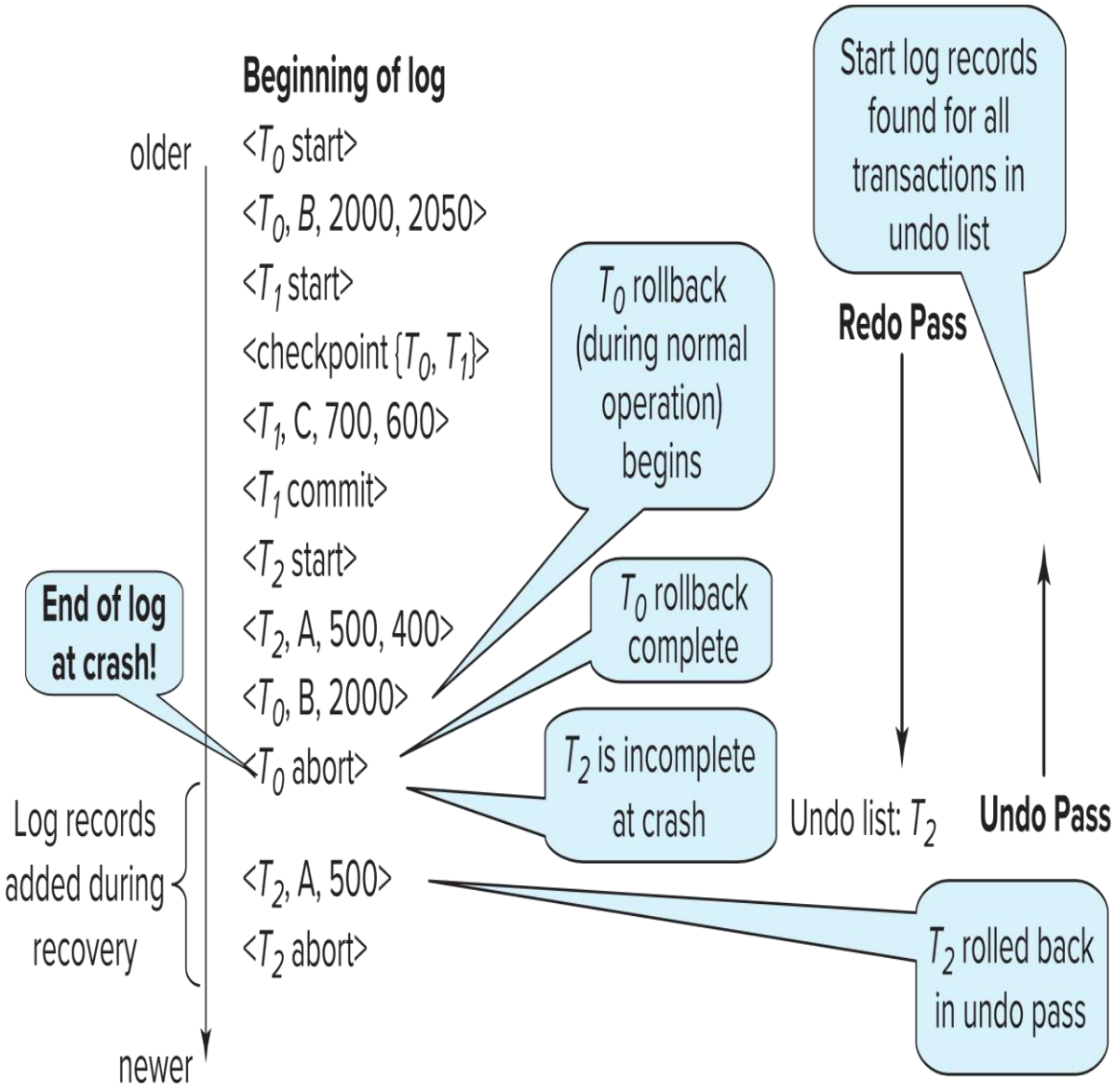
   The specific steps taken while scanning the log are as follows:

   a. The list of transactions to be rolled back, undo-list, is initially set to the list *L* in the <checkpoint *L*> log record.
   b. Whenever a normal log record of the form <*Ti* , *Xj* , *V*1, *V*2>, or a redo-only log record of the form <*Ti* , *Xj* , *V*2> is encountered, the operation is redone; that is, the value *V*2 is written to data item *Xj* .
   c. Whenever a log record of the form <*Ti* start> is found, *Ti* is added to undo-list.
   d. Whenever a log record of the form <*Ti* abort> or <*Ti* commit> is found, *Ti* is removed from undo-list.

At the end of the redo phase, undo-list contains the list of all transactions that are incomplete, that is, they neither committed nor completed rollback before the crash.

2. **Undo Phase:** The system rolls back all transactions in the undo-list. It performs rollback by scanning the log backward from the end.
   a. Whenever it finds a log record belonging to a transaction in the undolist, it performs undo actions just as if the log record had been found during the rollback of a failed transaction.
   b. When the system finds a <*Ti* start> log record for a transaction *Ti* in undo-list, it writes a <*Ti* abort> log record to the log, and removes *Ti* from undo-list.
   c. The undo phase terminates once undo-list becomes empty, that is, the system has found <*Ti* start> log records for all transactions that were initially in undo-list.

After the undo phase of recovery terminates, normal transaction processing can resume.



Example of logged actions, and actions during recovery.

Buffer Management

It is essential to the implementation of a crash-recovery scheme that ensures data consistency and imposes a minimal amount of overhead on interactions with the database.

There are three types of Buffering

1. **Log-Record Buffering**
2. **Database Buffering**
3. **Fuzzy Checkpointing**

1. Log-Record Buffering

We have assumed that every log record is output to stable storage at the time it is created. This assumption imposes a high overhead on system execution for several reasons:

Typically, output to stable storage is in units of blocks. In most cases, a log record is much smaller than a block. Thus, the output of each log record translates to a much larger output at the physical level.

As a result of log buffering, a log record may reside in only main memory (volatile storage) for a considerable time before it is output to stable storage. Since such log records are lost if the system crashes, we must impose additional requirements on the recovery techniques to ensure transaction atomicity:

- Transaction *Ti* enters the commit state after the <*Ti* commit> log record has been output to stable storage.
- Before the <*Ti* commit> log record can be output to stable storage, all log records pertaining to transaction *Ti* must have been output to stable storage.
- Before a block of data in main memory can be output to the database (in nonvolatile storage), all log records pertaining to data in that block must have been output to stable storage.

This rule is called the **Write-Ahead Logging (WAL)** rule. (Strictly speaking, the WAL rule requires only that the undo information in the log has been output to stable storage, and it permits the redo information to be written later. The difference is relevant in systems where undo information and redo information are stored in separate log records.)

The three rules state situations in which certain log records *must* have been output to stable storage. There is no problem resulting from the output of log records *earlier* than necessary. Thus, when the system finds it necessary to output a log record to stable storage, it outputs an entire block of log records, if there are enough log records in main memory to fill a block. If there are insufficient log records to fill the block, all log records in main memory are combined into a partially full block and are output to stable storage.

Writing the buffered log to disk is sometimes referred to as a **log force**.

2. Database Buffering

The system stores the database in nonvolatile storage (disk), and brings blocks of data into main memory as needed. Since main memory is typically much smaller than the entire database, it may be necessary to overwrite a block $B1$ in main memory when another block $B2$ needs to be brought into memory. If $B1$ has been modified, $B1$ must be output prior to the input of $B2$.

One might expect that transactions would force-output all modified blocks to disk when they commit. Such a policy is called the **force** policy.

The alternative, the **no-force** policy, allows a transaction to commit even if it has modified some blocks that have not yet been written back to disk.

As a result, the standard approach taken by most systems is the **no-force** policy.

Similarly, one might expect that blocks modified by a transaction that is still active should not be written to disk. This policy is called the **no-steal** policy.

The alternative, the **steal** policy, allows the system to write modified blocks to disk even if the transactions that made those modifications have not all committed.

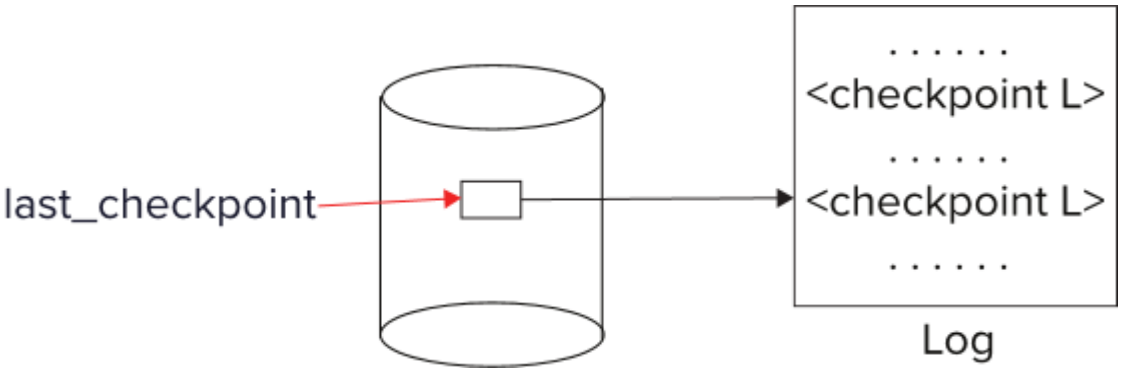
As a result, the standard approach taken by most systems is the **steal** policy.

3. Fuzzy Checkpointing

The checkpointing technique requires that all updates to the database be temporarily suspended while the checkpoint is in progress.

If the number of pages in the buffer is large, a checkpoint may take a long time to finish, which can result in an unacceptable interruption in processing of transactions.

To avoid such interruptions, the checkpointing technique can be modified to permit updates to start once the checkpoint record has been written, but before the modified buffer blocks are written to disk.



Operating System Role in Buffer Management

We can manage the database buffer by using one of two approaches:

1. The database system reserves part of main memory to serve as a buffer that it, rather than the operating system, manages. The database system manages data-block transfer in accordance with the requirements.

   This approach has the drawback of limiting flexibility in the use of main memory. The buffer must be kept small enough that other applications have sufficient main memory available for their needs.

   However, even when the other applications are not running, the database will not be able to make use of all the available memory. Likewise, non-database applications may not use that part of main memory reserved for the database buffer, even if some of the pages in the database buffer are not being used.

2. The database system implements its buffer within the virtual memory provided by the operating system. Since the operating system knows about the memory requirements of all processes in the system, ideally it should be in charge of deciding what buffer blocks must be force-output to disk, and when. But, to ensure the write-ahead logging requirements the operating system should not write out the database buffer pages itself, but instead should request the database system to force-output the buffer blocks. The database system in turn would force-output the buffer blocks to the database, after writing relevant log records to stable storage.

Unfortunately, almost all current-generation operating systems retain complete control of virtual memory. The operating system reserves space on disk for storing virtual-memory pages that are not currently in main memory; this space is called **swap space**. If the operating system decides to output a block *Bx*, that block is output to the swap space on disk, and there is no way for the database system to get control of the output of buffer blocks.

Failure with Loss of Nonvolatile Storage

Until now, we have considered only the case where a failure results in the loss of information residing in volatile storage while the content of the nonvolatile storage remains intact. Although failures in which the content of nonvolatile storage is lost are rare, we nevertheless need to be prepared to deal with this type of failure.

The basic scheme is to **dump** the entire contents of the database to stable storage periodically—say, once per day.

For example, we may dump the database to one or more magnetic tapes. If a failure occurs that results in the loss of physical database blocks, the system uses the most recent dump in restoring the database to a previous consistent state. Once this restoration has been accomplished, the system uses the log to bring the database system to the most recent consistent state.

One approach to database dumping requires that no transaction may be active during the dump procedure, and uses a procedure similar to checkpointing:

1. Output all log records currently residing in main memory onto stable storage.
2. Output all buffer blocks onto the disk.
3. Copy the contents of the database to stable storage.
4. Output a log record <dump> onto the stable storage.

Steps 1, 2, and 4 correspond to the three steps used for checkpoints.

To recover from the loss of nonvolatile storage, the system restores the database to disk by using the most recent dump. Then, it consults the log and redoes all the actions since the most recent dump occurred. Notice that no undo operations need to be executed.

In case of a partial failure of nonvolatile storage, such as the failure of a single block or a few blocks, only those blocks need to be restored, and redo actions performed only for those blocks.

A dump of the database contents is also referred to as an **archival dump**, since we can archive the dumps and use them later to examine old states of the database. Dumps of a database and checkpointing of buffers are similar.

Most database systems also support an **SQL dump**, which writes out SQL DDL statements and SQL insert statements to a file, which can then be re executed to re-create the database. Such dumps are useful when migrating data to a different instance

of the database, or to a different version of the database software, since the physical locations and layout may be different in the other database instance or database software version.

The simple dump procedure described here is costly for the following two reasons.

- First, the entire database must be copied to stable storage, resulting in considerable data transfer.
- Second, since transaction processing is halted during the dump procedure, CPU cycles are wasted.

Fuzzy dump schemes have been developed that allow transactions to be active while the dump is in progress.

They are similar to fuzzy-checkpointing schemes; see the bibliographical notes for more details.

ARIES
(Algorithms for Recovery and Isolation Exploiting Semantics)

The state of the art in recovery methods is best illustrated by the ARIES recovery method. ARIES uses a number of techniques to reduce the time taken for recovery, and to reduce the overhead of checkpointing.

In particular, ARIES is able to avoid redoing many logged operations that have already been applied and to reduce the amount of information logged. The price paid is greater complexity; the benefits are worth the price.

The major differences between ARIES and the recovery algorithm presented earlier are that ARIES:

1. Uses a **Log Sequence Number** (LSN) to identify log records, and stores LSNs in database pages to identify which operations have been applied to a database page.
2. Supports **Physiological Redo** operations, which are physical in that the affected page is physically identified, but can be logical within the page.

   For instance, the deletion of a record from a page may result in many other records in the page being shifted, if a slotted page structure is used. With physical redo logging, all bytes of the page affected by the shifting of records must be logged. With physiological logging, the deletion operation can be logged, resulting in a much smaller log record. Redo of the deletion operation would delete the record and shift other records as required.

3. Uses a **Dirty Page Table** to minimize unnecessary redos during recovery. As mentioned earlier, dirty pages are those that have been updated in memory, and the disk version is not up-to-date.
4. Uses a **Fuzzy-Checkpointing** scheme that records only information about dirty pages and associated information and does not even require writing of dirty pages to disk. It flushes dirty pages in the background, continuously, instead of writing them during checkpoints.

1. Data Structures

Each log record in ARIES has a Log Sequence Number (LSN) that uniquely identifies the record. The number is conceptually just a logical identifier whose value is greater for log records that occur later in the log. In practice, the LSN is generated in such a way that it can also be used to locate the log record on disk. Typically, ARIES splits a log into multiple log files, each of which has a file number. When a log file grows to some limit, ARIES appends further log records to a new log file; the new log file has a file number that is higher by 1 than the previous log file.

The LSN then consists of a file number and an offset within the file. Each page also maintains an identifier called the **PageLSN**. Whenever an update operation (whether physical or physiological) occurs on a page, the operation stores the LSN of its log record in the PageLSN field of the page.

During the redo phase of recovery, any log records with LSN less than or equal to the PageLSN of a page should not be executed on the page, since their actions are already reflected on the page.

In combination with a scheme for recording PageLSNs as part of checkpointing, ARIES can avoid even reading many pages for which logged operations are already reflected on disk. Thereby, recovery time is reduced significantly.

The PageLSN is essential for ensuring idempotence in the presence of physiological redo operations, since reapplying a physiological redo that has already been applied to a page could cause incorrect changes to a page.

Pages should not be flushed to disk while an update is in progress, since physiological operations cannot be redone on the partially updated state of the page on disk.

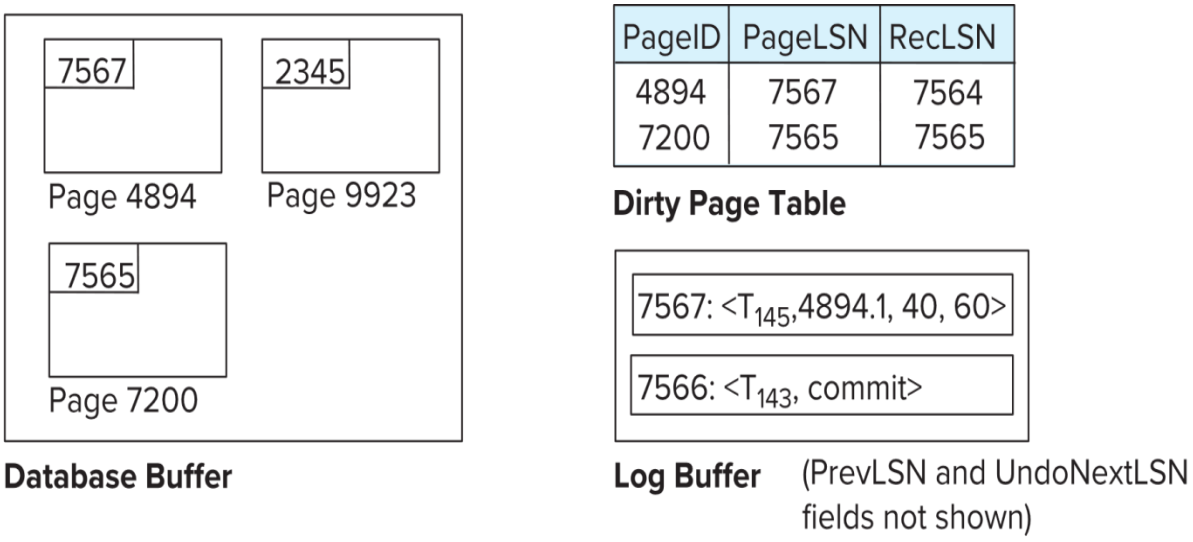
Therefore, ARIES uses latches on buffer pages to prevent them from being written to disk while they are being updated. It releases the buffer page latch only after the update is completed, and the log record for the update has been written to the log.

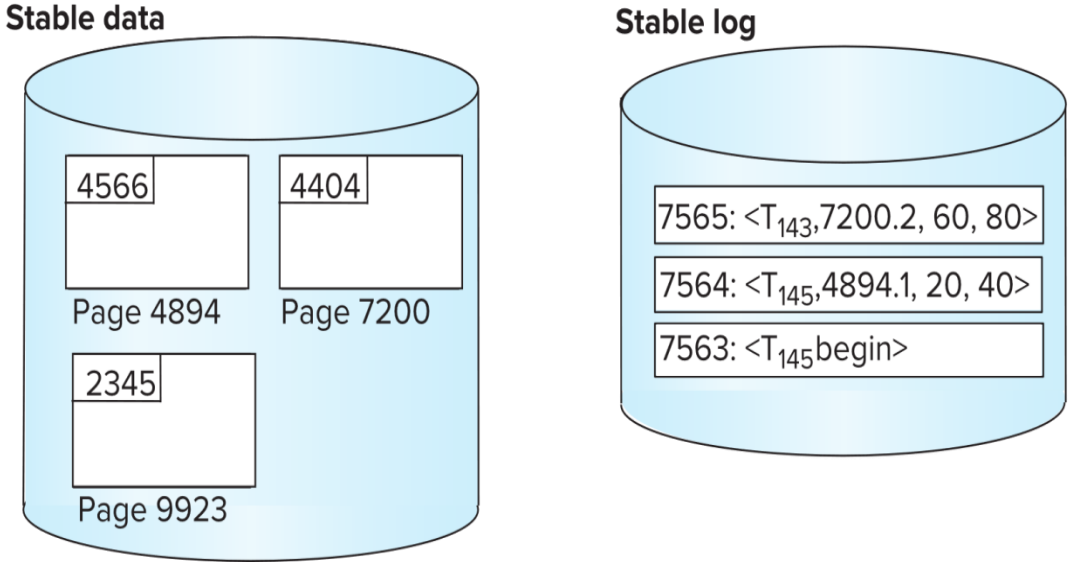
Each log record also contains the LSN of the previous log record of the same transaction. This value, stored in the PrevLSN field, permits log records of a transaction to be fetched backward, without reading the whole log.

There are special redo-only log records generated during transaction rollback, called **Compensation Log Records** (**CLR**s) in ARIES.

These serve the same purpose as the **redo-only log records** in our earlier recovery scheme. In addition CLRs serve the role of the operation-abort log records in our scheme.

The CLRs have an extra field, called the **UndoNextLSN**, that records the LSN of the log that needs to be undone next, when the transaction is being rolled back. This field serves the same purpose as the operation identifier in the **operation-abort log** record in our earlier recovery scheme, which helps to skip over log records that have already been rolled back.





Data Structures used in ARIES.

The **DirtyPageTable** contains a list of pages that have been updated in the database buffer. For each page, it stores the **PageLSN** and a field called the **RecLSN**, which helps identify log records that have been applied already to the version of the page on disk. When a page is inserted into the DirtyPageTable (when it is first modified in the buffer pool), the value of RecLSN is set to the current end of log. Whenever the page is flushed to disk, the page is removed from the DirtyPageTable.

A **Checkpoint Log Record** contains the DirtyPageTable and a list of active transactions. For each transaction, the checkpoint log record also notes LastLSN, the LSN of the last log record written by the transaction. A fixed position on disk also notes the LSN of the last (complete) checkpoint log record.

In the above figure illustrates some of the data structures used in ARIES. The log records shown in the figure are prefixed by their LSN; these may not be explicitly stored, but inferred from the position in the log, in an actual implementation.

The data item identifier in a log record is shown in two parts, for example **4894.1**; the **first part** identifies the **page**, and the **second part** identifies a **record** within the page (we assume a slotted page record organization within a page).

Note that the log is shown with newest records on top, since older log records, which are on disk, are shown lower in the figure.

Each page (whether in the buffer or on disk) has an associated PageLSN field. You can verify that the LSN for the last log record that updated page 4894 is 7567.

By comparing PageLSNs for the pages in the buffer with the PageLSNs for the corresponding pages in stable storage, you can observe that the DirtyPageTable contains entries for all pages in the buffer that have been modified since they were fetched from stable storage.

The RecLSN entry in the DirtyPageTable reflects the LSN at the end of the log when the page was added to DirtyPageTable, and would be greater than or equal to the PageLSN for that page on stable storage.

2. Recovery Algorithm

ARIES recovers from a system crash in three passes.

- **Analysis Pass**: This pass determines which transactions to undo, which pages were dirty at the time of the crash, and the LSN from which the redo pass should start.
- **Redo Pass**: This pass starts from a position determined during analysis, and performs a redo, repeating history, to bring the database to a state it was in before the crash.
- **Undo Pass**: This pass rolls back all transactions that were incomplete at the time of crash.

1. Analysis Pass

The analysis pass finds the last complete checkpoint log record, and reads in the DirtyPageTable from this record. It then sets RedoLSN to the minimum of the RecLSNs of the pages in the DirtyPageTable.

If there are no dirty pages, it sets RedoLSN to the LSN of the checkpoint log record. The redo pass starts its scan of the log from RedoLSN. All the log records earlier than this point have already been applied to the database pages on disk. The analysis pass initially sets the list of transactions to be undone, undo-list, to the list of transactions in the checkpoint log record. The analysis pass also reads from the checkpoint log record the LSNs of the last log record for each transaction in undo-list.

The analysis pass continues scanning forward from the checkpoint. Whenever it finds a log record for a transaction not in the undo-list, it adds the transaction to undo-list. Whenever it finds a transaction end log record, it deletes the transaction from undo-list. All transactions left in undo-list at the end of analysis have to be rolled back later, in the undo pass. The analysis pass also keeps track of the last record of each transaction in undo-list, which is used in the undo pass.

The analysis pass also updates DirtyPageTable whenever it finds a log record for an update on a page. If the page is not in DirtyPageTable, the analysis pass adds it to DirtyPageTable, and sets the RecLSN of the page to the LSN of the log record.

2. Redo Pass

The redo pass repeats history by replaying every action that is not already reflected in the page on disk. The redo pass scans the log forward from RedoLSN. Whenever it finds an update log record, it takes this action:

1. If the page is not in DirtyPageTable or the LSN of the update log record is less than the RecLSN of the page in DirtyPageTable, then the redo pass skips the log record.
2. Otherwise the redo pass fetches the page from disk, and if the PageLSN is less than the LSN of the log record, it redoes the log record.

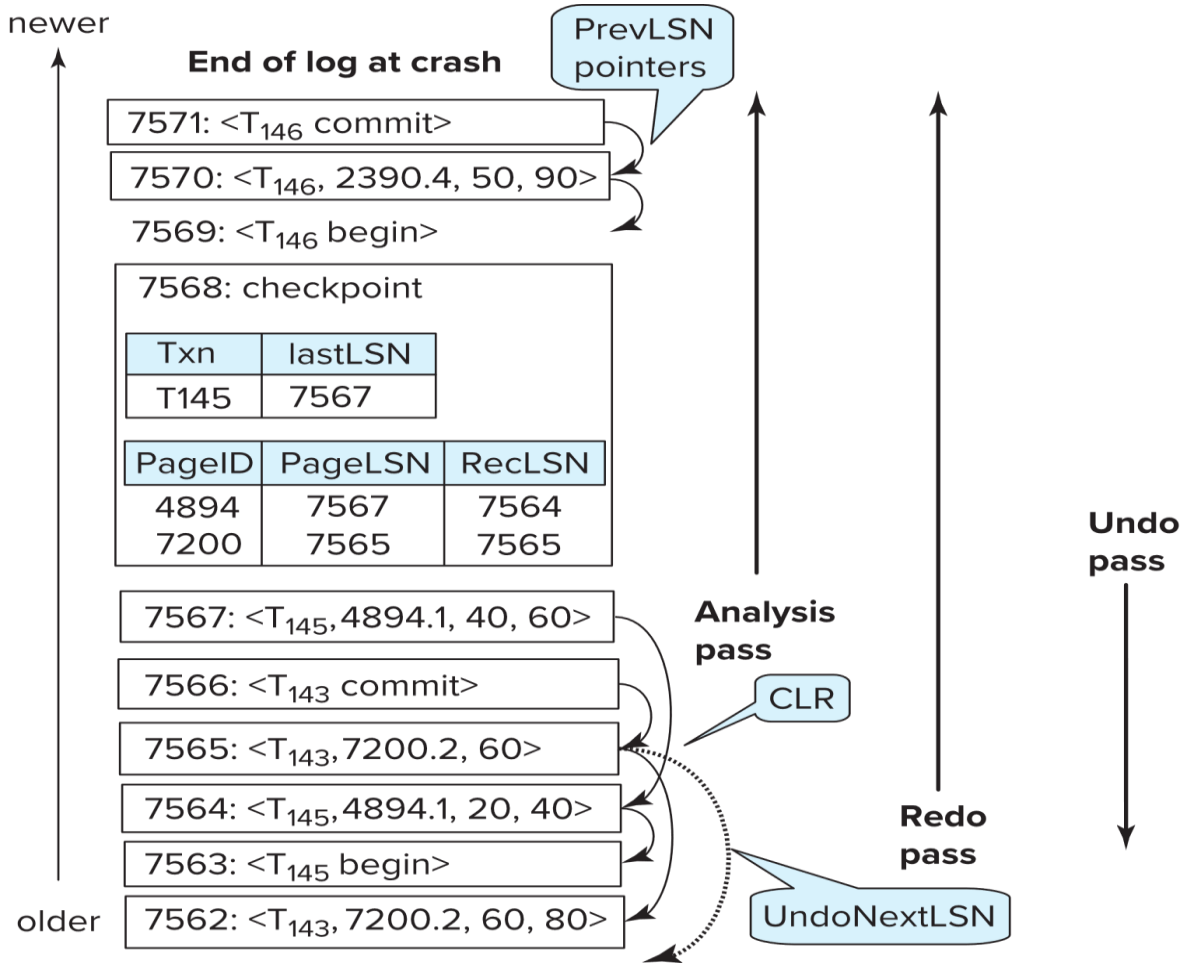
Note that if either of the tests is negative, then the effects of the log record have already appeared on the page; otherwise the effects of the log record are not reflected on the page. Since ARIES allows non-idempotent physiological log records, a log record should not be redone if its effect is already reflected on the page. If the first test is negative, it is not even necessary to fetch the page from disk to check its PageLSN.

3. Undo Pass and Transaction Rollback

The undo pass is relatively straightforward. It performs a single backward scan of the log, undoing all transactions in undo-list. The undo pass examines only log records of transactions in undo-list; the last LSN recorded during the analysis pass is used to find the last log record for each transaction in undo-list.

Whenever an update log record is found, it is used to perform an undo (whether for transaction rollback during normal processing or during the restart undo pass). The undo pass generates a CLR containing the undo action performed (which must be physiological). It sets the UndoNextLSN of the CLR to the PrevLSN value of the update log record.

If a CLR is found, its UndoNextLSN value indicates the LSN of the next log record to be undone for that transaction; later log records for that transaction have already been rolled back. For log records other than CLRs, the PrevLSN field of the log record indicates the LSN of the next log record to be undone for that transaction. The next log record to be processed at each stop in the undo pass is the maximum, across all transactions in undo-list, of next log record LSN.



Recovery actions in ARIES

The above Figure illustrates the recovery actions performed by ARIES, on an example log.

We assume that the last completed checkpoint pointer on disk points to the checkpoint log record with LSN 7568. The PrevLSN values in the log records are shown using arrows in the figure, while the UndoNextLSN value is shown using a dashed arrow for the one compensation log record, with LSN 7565, in the figure.

The analysis pass would start from LSN 7568, and when it is complete, RedoLSN would be 7564. Thus, the redo pass must start at the log record with LSN 7564.

Note that this LSN is less than the LSN of the checkpoint log record, since the ARIES checkpointing algorithm does not flush modified pages to stable storage.

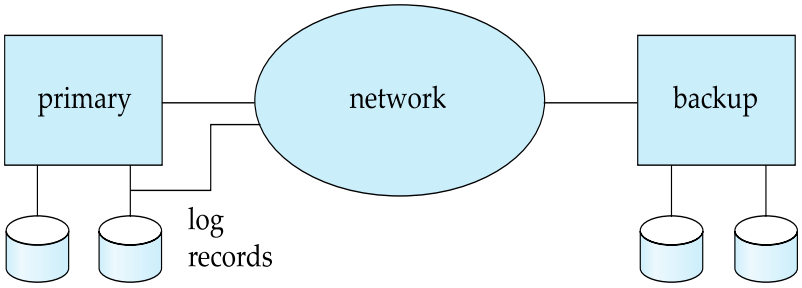
The DirtyPageTable at the end of analysis would include pages 4894, 7200 from the checkpoint log record, and 2390 which is updated by the log record with LSN 7570. At the end of the analysis pass, the list of transactions to be undone consists of only *T*145 in this example.

The redo pass for the above example starts from LSN 7564 and performs redo of log records whose pages appear in DirtyPageTable. The undo pass needs to undo only transaction *T*145, and hence starts from its LastLSN value 7567, and continues backwards until the record < *T*145 start> is found at LSN 7563.

Remote Backup Systems

Traditional transaction-processing systems are centralized or client–server systems. Such systems are vulnerable to environmental disasters such as fire, flooding, or earthquakes. Increasingly, there is a need for transaction-processing systems that can function in spite of system failures or environmental disasters. Such systems must provide **high availability**; that is, the time for which the system is unusable must be extremely small.

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. The below Figure shows the architecture of a remote backup system.



Architecture of remote backup system.

When the primary site fails, the remote backup site takes over processing. First, however, it performs recovery, using its (perhaps outdated) 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. Standard recovery algorithms, with minor modifications, can be used for recovery at the remote backup site. Once recovery has been performed, the remote backup site starts processing transactions.

Availability is greatly increased over a single-site system, since the system can recover even if all data at the primary site are lost.

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. For example, several independent network connections, including perhaps a modem connection over a telephone line, may be used. These connections may be backed up via manual intervention by operators, who can communicate over the telephone system.
- **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.

  The simplest way of transferring control is for the old primary to receive redo logs from the old backup site, and to catch up with the updates by applying them locally. The old primary can then act as a remote backup site. If control must be transferred back, the old backup site can pretend to have failed, resulting in the old primary taking over.

- **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. The delay before the remote backup takes over can be significantly reduced as a result.

  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.

    The problem with this scheme is that the updates of a committed transaction may not have made it to the backup site, when the backup site takes over processing. Thus, the updates may appear to be lost. When the primary site recovers, the lost updates cannot be merged in directly, since the updates may conflict with later updates performed at the backup site. Thus, human intervention may be required to bring the database to a consistent state.
  - **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.

    The problem with this scheme is that transaction processing cannot proceed if either the primary or the backup site is down. Thus, availability is actually less than in the single-site case, although the probability of data loss is much less.
  - **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.

  This scheme provides better availability than does two-very-safe, while avoiding the problem of lost transactions faced by the one-safe scheme. It results in a slower commit than the one-safe scheme, but the benefits generally outweigh the cost.