9. Reliability

Aspects and Definitions

- A measure of success with which a system conforms to some authoritative specification of its behavior.
- Probability that the system does not experience failures within a given period.
- Typically used to describe systems that cannot be repaired or where the continuous operation of the system is critical.
- In transactional context: How to maintain *Atomicity* and *Durability*

Crash and crash recovery

- By *crash* all kinds of failures are denoted that bring down a server and cause all data in volatile memory to be lost (*soft crash*), but leave all data on stable secondary storage intact, i.e. not a (*hard crash*).
- A *crash recovery* algorithm restarts the server and brings its permanent data back to its most recent, consistent state

During crash recovery after a system failure, a server and its data are unavailable to clients. Goal: minimize recovery time

Recovery performance and system availability

MTBF: *mean time between failure*

MTTR: *mean time to repair*

Availability: probability for a server to be ready to serve:

$$\frac{MTBF}{MTBF + MTTR}$$

Examples

- Server fails once a month and takes 2 hours to recover: availability of 99.7%, downtime of 26 h a year.
- Server fails once every 48 h and takes 30 sec to recover: availability of 99.98%, downtime of 105 min a year.

$\Longrightarrow$ Fast recovery is the key to high availability!

During crash recovery after a system failure, a server and its data are unavailable to clients. Goal: minimize recovery time

Recovery performance and system availability

MTBF: *mean time between failure*

MTTR: *mean time to repair*

Availability: probability for a server to be ready to serve:

$$\frac{MTBF}{MTBF + MTTR}$$

Examples

- Server fails once a month and takes 2 hours to recover: availability of 99.7%, downtime of 26 h a year.
- Server fails once every 48 h and takes 30 sec to recover: availability of 99.98%, downtime of 105 min a year.

$\Longrightarrow$ Fast recovery is the key to high availability!

During crash recovery after a system failure, a server and its data are unavailable to clients. Goal: minimize recovery time

Recovery performance and system availability

MTBF: *mean time between failure*

MTTR: *mean time to repair*

Availability: probability for a server to be ready to serve:

$$\frac{MTBF}{MTBF + MTTR}$$

Examples

- Server fails once a month and takes 2 hours to recover: availability of 99.7%, downtime of 26 h a year.
- Server fails once every 48 h and takes 30 sec to recover: availability of 99.98%, downtime of 105 min a year.

$\Longrightarrow$ Fast recovery is the key to high availability!

Local Reliablity Protocols

ARIES:

- Write-ahead Logging
- Repeating History on Crash

Distributed Reliability Protocols

- Commit Protocols
  - How to execute commit command for distributed transactions?
  - How to ensure Atomicity and Durability?
- Termination Protocols
  - If a failure occurs, how can the remaining operational sites deal with it?
  - *Non-blocking*: the occurrence of failures should not force the sites to wait until the failure is repaired to terminate the transaction.
- Recovery Protocols
  - When a failure occurs, how do the sites where it occurred deal with it?
  - *Independent*: a failed site can determine the outcome of a transaction without having to obtain remote information.

$\Longrightarrow$ Independent recovery $\rightarrow$ Non-blocking termination

Local Recovery (Refresh)

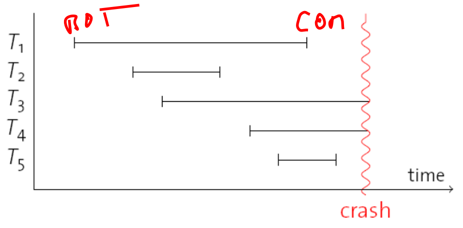
Failure Recovery

We want to deal with three types of failures:

- **transaction failure (also: process failure)**: A transaction voluntarily or involuntarily aborts. All of its updates need to be *undone*
- **system failure**: Database or operating system crash, power outage, etc. All information in main memory is lost. Must make sure that no committed transaction is lost (or *redo* their effects) and that all other transactions are *undone*.
- **media failure (also: device failure)**: Hard disk crash, catastrophic error (fire, water, ...). Must recover database from *stable storage*

In spite of all these failures, we we want to guarantee *atomicity* and *durability*.

Example: System Failure



- Transactions T_1, T_2, and T_5 were committed before the crash.
  - **Durability**: Ensure that updates are *preserved* (or *redone*).
- Transactions T3 and T4 were not (yet) committed.
  - **Atomicity**: All of their effects need to be *undone*.

Types of Storage

We assume three different types of storage:

- **volatile storage**: This is essentially the *buffer manager* in *main memory*. We are going to use volatile storage to cache the "*write-ahead log*" in a moment.
- **non-volatile storage**: Typical candidate is a hard disk or SSD
- **stable storage**: Non-volatile storage that survives all types of failures which is hard to achieve in practice. Stability can be improved using, e.g., (network) *replication* of disk data. Backup tapes are another example.

Observe how these storage types correspond to the three types of failures.

Interaction between volatile and non-volatile storage

Coordination policies between transactions and storage on non-volatile memory

- Can modified pages written to disk even if there is no commit (**Steal**)?
- Can we delay writing modified pages after commit (**No-Force**)?

Steal+No-Force

- improve throughput and latency,
- but make recovery more complicated

Types of Storage

We assume three different types of storage:

- **volatile storage**: This is essentially the *buffer manager* in *main memory*. We are going to use volatile storage to cache the "*write-ahead log*"in a moment.
- **non-volatile storage**: Typical candidate is a hard disk or SSD
- **stable storage**: Non-volatile storage that survives all types of failures which is hard to achieve in practice. Stability can be improved using, e.g., (network) *replication* of disk data. Backup tapes are another example.

Observe how these storage types correspond to the three types of failures.

Interaction between volatile and non-volatile storage

Coordination policies between transactions and storage on non-volatile memory

- Can modified pages written to disk even if there is no commit (**Steal**)?
- Can we delay writing modified pages after commit (**No-Force**)?

Steal+No-Force

- improve throughput and latency,
- but make recovery more complicated

Effects of TA/storage coordination on recovery

The decisions force/no force and steal/no steal have implications on what we have to do during recovery:

write at commit

write during

	force	no force
no steal	no redo no undo	must redo no undo
steal	no redo must undo	must redo must undo

If we want to use steal and no force (to increase concurrency and performance), we have to implement redo and undo routines.

ARIES Algorithm

- **A**lgorithm for **R**ecovery and **I**solation **E**xploiting **S**emantics (?) 1992
- A better alternative to shadow paging which switches between active/committed page
- Works with steal and no-force
- Data pages are updated in place
- Uses "logging"
  - Log: An ordered list of REDO/UNDO actions.
  - Record REDO and UNDO information for every update. → append only
  - Sequential writes to log (usually kept on separate disk(s)).
  - Minimal info written to log ⇝ multiple updates fit in a single log page.

Random ~ 100 writes/s → 100 kB/s

Seq: ~ 100 MB/s 1000x

Three main principles of ARIES

1. Write-Ahead Logging
   - Record database changes in the log at stable storage before the actual change.
2. Repeating History During Redo
   - After a crash, bring the system back to the exact state at crash time; undo the transactions that were still active at crash time.
3. Logging Changes During Undo
   - Log the database changes during a transaction undo so that they are not repeated in case of repeated failures and restarts (i.e., never undo an undo action).

Three main principles of ARIES

1. Write-Ahead Logging → atomicity
   - Record database changes in the log at stable storage before the actual change.
2. Repeating History During Redo
   - After a crash, bring the system back to the exact state at crash time; undo the transactions that were still active at crash time.
3. Logging Changes During Undo
   - Log the database changes during a transaction undo so that they are not repeated in case of repeated failures and restarts (i.e., never undo an undo action).

Three main principles of ARIES

1. Write-Ahead Logging
   - Record database changes in the log at stable storage before the actual change.
2. Repeating History During Redo
   - After a crash, bring the system back to the exact state at crash time; undo the transactions that were still active at crash time.
3. Logging Changes During Undo
   - Log the database changes during a transaction undo so that they are not repeated in case of repeated failures and restarts (i.e., never undo an undo action).

Three main principles of ARIES

1. Write-Ahead Logging
   - Record database changes in the log at stable storage before the actual change.
2. Repeating History During Redo
   - After a crash, bring the system back to the exact state at crash time; undo the transactions that were still active at crash time.
3. Logging Changes During Undo
   - Log the database changes during a transaction undo so that they are not repeated in case of repeated failures and restarts (i.e., never undo an undo action).

Write-Ahead Log (WAL)

- The ARIES recovery method uses a "write-ahead log" to implement the necessary redundancy.
  - Mohan et al., ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging, ACM TODS, 17(1), 1992.
- WAL: Any change to a database object is **first** recorded in the log, which must be written to stable storage **before** the change itself is written to disk.
  - To ensure atomicity and prepare for undo, undo information must be written to stable storage before a page update is written back to disk.
  - To ensure durability, redo information must be written to stable storage at commit time (no-force policy: the on-disk data page may still contain old information).

Write-Ahead Log (WAL)

- The ARIES recovery method uses a "write-ahead log" to implement the necessary redundancy.
  - Mohan et al., ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging, ACM TODS, 17(1), 1992.
- WAL: Any change to a database object is **first** recorded in the log, which must be written to stable storage **before** the change itself is written to disk.
  - To ensure atomicity and prepare for undo, undo information must be written to stable storage before a page update is written back to disk.
  - To ensure durability, redo information must be written to stable storage at commit time (no-force policy: the on-disk data page may still contain old information).

Write-Ahead Log (WAL)

- The ARIES recovery method uses a "write-ahead log"to implement the necessary redundancy.
  - Mohan et al., ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging, ACM TODS, 17(1), 1992.
- WAL: Any change to a database object is **first** recorded in the log, which must be written to stable storage **before** the change itself is written to disk.
  - To ensure atomicity and prepare for undo, undo information must be written to stable storage before a page update is written back to disk.
  - To ensure durability, redo information must be written to stable storage at commit time (no-force policy: the on-disk data page may still contain old information).

Write-Ahead Log (WAL)

- The ARIES recovery method uses a "write-ahead log" to implement the necessary redundancy.
  - Mohan et al., ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging, ACM TODS, 17(1), 1992.
- WAL: Any change to a database object is **first** recorded in the log, which must be written to stable storage **before** the change itself is written to disk.
  - To ensure atomicity and prepare for undo, undo information must be written to stable storage before a page update is written back to disk.
  - To ensure durability, redo information must be written to stable storage at commit time (no-force policy: the on-disk data page may still contain old information).

Log Information

The log consists of entries in the following form:

$$< LSN, Type, TID, PrevLSN, PageID, NextLSN, Redo, Undo >$$

- LSN: Log Sequence Number: Monotonically increasing number to identify each log record.
- Type (Record Type): Begin, Commit, Abort, Update, Compensation
- TID: Transaction Identifier
- PrevLSN: Previous LSN of the same transaction
- PageID: Page which was modified
- NextLSN: Next LSN of the same transaction
- Redo Information described by this log entry
- Undo Information described by this log entry

Example of transactions and logs

Transaction 1	Transaction 2	LSN	Type	TX	Prev	Page	UNxt	Redo	Undo
$a \leftarrow$ read(A);									
	$c \leftarrow$ read(C);								
$a \leftarrow a - 50$;									
	$c \leftarrow c + 10$;								
write(a,A);		1	UPD	T_1	–	…		$A := A - 50$	$A := A + 50$
	write(c,C);	2	UPD	T_2	–	…		$C := C + 10$	$C := C - 10$
$b \leftarrow$ read(B);									
$b \leftarrow b + 50$;									
write(b,B);		3	UPD	T_1	1	…		$B := B + 50$	$B := B - 50$
commit;		4	EOT	T_1	3	…			
	$a \leftarrow$ read(A);								
	$a \leftarrow a - 10$;								
	write(a,A);	5	UPD	T_2	2	…		$A := A - 10$	$A := A + 10$
	commit;	6	EOT	T_2	5	…			

Redo Information

- ARIES assumes **page-oriented** redo
- stores byte images of the pages
- *before* and *after* the modification
- Restore exact same pages as execution without failures

Undo Information

- ARIES assumes **logical undo**
- Record the actual tuple changes, e.g. account A increased by 50
- Faster undo

Redo Information

- ARIES assumes **page-oriented** redo
- stores byte images of the pages
- *before* and *after* the modification
- Restore exact same pages as execution without failures

System crash → many transactions

Undo Information

- ARIES assumes **logical undo**
- Record the actual tuple changes, e.g. account A increased by 50
- Faster undo

Failure of TA "normal" → often, individual

Writing Log Records → atomicity

RAM

- For performance reasons, all log records are first written to volatile storage.
- At certain times, the log is forced to stable storage up to a certain LSN:
  - Commit of a transaction for Redo
  - Page writing of uncommitted for Undo
- Committed transaction = all log records (including commit) are on stable storage

Normal Processing

- During normal transaction processing, keep two pieces of information in each transaction control block:
  - LastLSN: LSN of the last log record written for this transaction.
  - NextLSN: LSN of the next log record to be processed during rollback.
- Whenever an update to a page p is performed
  - a log record r is written to the WAL, and
  - the LSN of r is recorded in the page header of p.

Writing Log Records

- For performance reasons, all log records are first written to volatile storage.
- At certain times, the log is forced to stable storage up to a certain LSN:
  - Commit of a transaction for Redo
  - Page writing of uncommitted for Undo
- Committed transaction = all log records (including commit) are on stable storage

Normal Processing

- During normal transaction processing, keep two pieces of information in each transaction control block:
  - LastLSN: LSN of the last log record written for this transaction.
  - NextLSN: LSN of the next log record to be processed during rollback.
- Whenever an update to a page p is performed
  - a log record r is written to the WAL, and
  - the LSN of r is recorded in the page header of p.

ARIES Transaction Rollback

- To roll back a transaction T after a **transaction failure** (e.g. ABORT):
  - Process the log in a backward fashion.
  - Start the undo operation at the log entry pointed to by the UNxt field in the transaction control block of T.
  - Find the remaining log entries for T by following the Prev and UNxt fields in the log.
  - Perform the changes in the Undo part of the log entry
- Undo operations modify pages, too!
  - Log all undo operations to the WAL.
  - Use compensation log records (CLRs) for this purpose.
  - Note: We never undo an undo action, but we might need to redo an undo action.

ARIES Crash Recovery

Restart after a system failure is performed in three phases

1. Analysis Phase:
   - Read log in forward direction.
   - Determine all transactions that were active when the failure happened. Such transactions are called "losers".
2. Redo Phase:
   - Replay the log (in forward direction) to bring the system into the state as of the time of system failure.
   - Put after imagesin place of before images
   - Also restores the losers
3. Undo Phase
   - Roll back all loser transactions, reading the log in a backward fashion (similar to "normal" rollback).

Media Recovery → Disk gone, too

- To allow for recovery from non-volatile media failure, periodically back up data to stable storage.
- Can be done during normal processing, if WAL is archived, too.
- Other approach: Use log to mirror database on a remote host (send log to network and to stable storage).

Checkpointing

- WAL file keeps growing unbounded
- For recovery, we need to visit entire WAL file
- Generate checkpoints with current transaction state
  - Recovery only from checkpoint
  - Bound WAL file and allow truncation

Media Recovery

- To allow for recovery from non-volatile media failure, periodically back up data to stable storage.
- Can be done during normal processing, if WAL is archived, too.
- Other approach: Use log to mirror database on a remote host (send log to network and to stable storage).

Checkpointing

- WAL file keeps growing unbounded
- For recovery, we need to visit entire WAL file
- Generate checkpoints with current transaction state
  - Recovery only from checkpoint
  - Bound WAL file and allow truncation