

# Write-Ahead Logging (WAL) Implementation: Design Document

---

## Overview

A crash-resistant transaction logging system that ensures database durability by writing all changes to a sequential log before applying them to the database. The key architectural challenge is implementing ARIES-style recovery that can reconstruct consistent database state after crashes by replaying committed transactions and rolling back incomplete ones.

This guide is meant to help you understand the big picture before diving into each milestone. Refer back to it whenever you need context on how components connect.

## Context and Problem Statement

**Milestone(s):** Foundational concepts for all milestones; primarily supports Milestone 1 (Log Record Format) and Milestone 3 (Crash Recovery)

## The Durability Challenge

Think of a database as a meticulous librarian managing thousands of books. When patrons request changes—adding new books, updating catalog entries, or removing damaged volumes—the librarian must ensure that every change is permanent and survives even if the library loses power or the building shakes. If the librarian simply made changes directly to the catalog without any backup system, a power outage in the middle of an update could leave the catalog in an inconsistent state: half-updated entries, missing records, or corrupted indexes.

The **durability problem** in databases stems from the fundamental mismatch between how applications expect data persistence to work and how computer hardware actually operates. Applications treat database operations as atomic, all-or-nothing events: when a transaction commits, the changes should be permanently stored and survive any subsequent failure. However, modern storage systems operate through multiple layers of caching, buffering, and delayed writes that make this guarantee extraordinarily difficult to achieve reliably.

When an application issues a write operation to a database, the data typically flows through several volatile layers before reaching permanent storage. The database engine maintains in-memory buffers for frequently accessed pages, the operating system caches file system writes in kernel buffers, and even the storage device itself may buffer writes in volatile memory before committing them to persistent media like SSDs or

magnetic disks. A crash at any point in this pipeline can result in lost writes, partially applied transactions, or inconsistent database state.

The naive approach of directly modifying database pages on disk creates several critical problems. First, **partial page writes** can occur when a crash interrupts the modification of a database page, leaving it in a state where some fields reflect the new values while others retain old values. Second, **ordering dependencies** between related changes cannot be guaranteed—a crash might persist a foreign key reference while losing the corresponding primary key record. Third, **transaction atomicity** becomes impossible to maintain when individual operations within a transaction are applied directly to persistent storage without coordination.

Consider a banking system processing a transfer of \$100 from Account A to Account B. The transaction involves two operations: decrementing A's balance and incrementing B's balance. If these updates are applied directly to the database pages on disk, a crash between the two operations could result in money being deducted from A without being added to B—violating the fundamental integrity constraint that money cannot be created or destroyed. Even more subtle issues arise from the fact that each account's balance might be stored on a different database page, and the operating system provides no guarantees about the order in which cached page writes reach the disk.

The **Write-Ahead Logging (WAL)** approach solves these problems by introducing a sequential, append-only log that records all intended changes before they are applied to the actual database pages. This design leverages the fundamental insight that sequential writes to storage are much faster and more atomic than random updates scattered across many database pages. Instead of modifying database pages directly, the system first writes a complete description of the intended change to the log, forces that log entry to persistent storage, and only then applies the change to the in-memory database buffers.

The WAL acts as a contract between the database engine and the storage system: the log entry serves as a durable record of what the database has committed to do, even if it hasn't yet done it. During normal operation, the database can apply logged changes to the actual database pages in the background, potentially batching multiple updates and optimizing disk I/O patterns. During crash recovery, the database can read the log to determine exactly what changes were committed but might not have been applied to the database pages, then replay those changes to restore a consistent state.

This approach transforms the durability problem from "ensure every database page update is atomic and durable" to the much more manageable "ensure log entries are written atomically to a sequential file." The latter problem has well-understood solutions involving careful use of operating system sync primitives and atomic append operations.

The key insight of WAL is that durability can be achieved more reliably through redundancy than through perfect atomic updates. By logging intended changes before applying them, the system creates a durable record that can be used to recover consistent state even when the primary database pages are corrupted or inconsistent.

## Crash Recovery Scenarios

Understanding WAL requires examining the specific failure scenarios that can occur during database operation and how different system states at crash time affect the recovery process. Each scenario presents unique challenges for determining which changes should be preserved and which should be rolled back.

The most straightforward scenario is a **clean shutdown** where all transactions have completed and committed changes have been fully applied to database pages. In this case, recovery is trivial—the system can restart immediately without needing to process any log entries. However, this scenario is relatively rare in practice, as most database systems maintain some uncommitted transactions or unapplied changes to optimize performance.

A **crash during active transaction processing** represents the most complex recovery scenario. At the moment of crash, the system may contain transactions in various states: some have committed and their changes are safely recorded in the log, others are still active and have made partial progress, and still others may have been in the process of committing when the crash occurred. The challenge lies in correctly identifying which changes belong to committed transactions (and must be preserved) versus uncommitted transactions (which must be rolled back).

Consider a scenario where three transactions T1, T2, and T3 are executing concurrently when a power failure occurs. Transaction T1 has written several log records but has not yet received a commit confirmation from the application. Transaction T2 has received a commit request from the application and has written a commit record to the log, but the log may not have been fsynced to disk yet. Transaction T3 has fully committed—its commit record is durably stored in the log—but some of its changes may not have been applied to the database pages yet due to lazy write policies.

The recovery system must be able to distinguish between these cases by examining the contents of the log after restart. The presence or absence of a commit record for each transaction determines whether its changes should be preserved (redo) or discarded (undo). However, the recovery process is complicated by the fact that some changes from committed transactions might already be present in database pages (applied before the crash) while others might be missing (not yet applied). Similarly, some changes from uncommitted transactions might have been applied to database pages and need to be reversed.

**Log corruption scenarios** add another layer of complexity to crash recovery. Storage devices can experience partial failures that corrupt some log records while leaving others intact. Network-attached storage or distributed file systems might return stale data or lose recent writes during network partitions. Even local SSDs can experience firmware bugs that result in lost or corrupted writes despite successful fsync operations.

The recovery system must be able to detect corrupted log records and determine how much of the log can be trusted. This typically involves checksums or hash values embedded in each log record, combined with structural validation of log record formats. When corruption is detected, the system faces a difficult trade-off: aborting recovery preserves correctness but may make the database permanently unavailable, while continuing recovery risks propagating corrupted data into the restored database state.

**Partial write scenarios** occur when a crash interrupts the writing of a log record itself, leaving the log file in a state where the last record is incomplete. Unlike database page updates, which can be partially applied, log records must be treated as atomic units—a partially written log record cannot be meaningfully interpreted and must be discarded during recovery.

Detecting partial writes requires careful attention to log record formatting and atomic append semantics. Most WAL implementations use fixed-size record headers containing length fields that allow the recovery system to determine whether a complete record is present. Advanced implementations may use techniques like double-write buffers or record-level checksums that span the entire record to ensure atomicity.

**Cascading failure scenarios** involve multiple system components failing in sequence, potentially creating complex recovery challenges. For example, a primary database server might crash and fail over to a replica, which then experiences its own crash before completing the recovery process. Or a database might experience repeated crashes during recovery due to hardware issues, requiring the recovery process to be restarted multiple times.

These scenarios test the **idempotency** of recovery operations—the ability to safely re-execute recovery procedures multiple times without causing inconsistencies. A well-designed recovery system should produce the same final database state regardless of how many times recovery is interrupted and restarted, as long as the log remains intact.

Crash Scenario	Transaction State	Recovery Challenge	Detection Method
Clean Shutdown	All committed, all applied	None - immediate restart	No active transaction records
Active Transactions	Mixed committed/uncommitted	Distinguish redo vs undo operations	Presence/absence of commit records
Log Corruption	Indeterminate	Determine trustworthy log prefix	Record checksums and structural validation
Partial Log Write	Last record incomplete	Identify atomic record boundary	Record length fields and completion markers
Cascading Failures	Recovery interrupted	Ensure idempotent recovery operations	Recovery state checkpoints

## Existing WAL Approaches

Real-world database systems have evolved sophisticated WAL implementations that balance performance, correctness, and operational complexity. Examining how PostgreSQL, SQLite, and MongoDB approach WAL illuminates the key design decisions and trade-offs involved in building production-ready logging systems.

**PostgreSQL** implements a comprehensive WAL system based closely on the ARIES (Algorithm for Recovery and Isolation Exploiting Semantics) protocol developed by IBM Research. PostgreSQL's approach prioritizes correctness and full ACID compliance, making it an excellent reference for understanding the complete complexity of enterprise-grade WAL systems.

PostgreSQL's WAL records contain detailed before and after images for every modified tuple (row), along with sufficient metadata to support both redo and undo operations. The system maintains separate WAL files (typically 16MB each) that are written sequentially and rotated when full. Each WAL record includes a Log Sequence Number (LSN) that serves as both a unique identifier and a logical timestamp for ordering recovery operations.

The PostgreSQL recovery process implements the full three-phase ARIES algorithm: analysis (scan the log to identify committed transactions and dirty pages), redo (replay all committed changes to restore database state), and undo (roll back incomplete transactions). This approach guarantees that recovery will produce a consistent database state regardless of when the crash occurred or what transactions were active.

PostgreSQL's checkpointing system uses a "fuzzy checkpoint" approach that allows normal transaction processing to continue while checkpoint data is being gathered and written. The checkpoint record contains the LSN of the oldest WAL record that might be needed for recovery, enabling the system to truncate older WAL files once the checkpoint is complete. This design bounds recovery time by limiting how much log history must be processed during startup.

### Decision: Full ARIES Implementation in PostgreSQL

- **Context:** PostgreSQL targets enterprise workloads requiring strict ACID compliance and predictable recovery behavior
- **Options Considered:** Simplified logging protocols, shadow paging approaches, log-structured storage
- **Decision:** Implement full ARIES with three-phase recovery and comprehensive undo logging
- **Rationale:** ARIES provides mathematical guarantees about consistency and has been proven in production systems; comprehensive logging supports advanced features like point-in-time recovery and logical replication
- **Consequences:** Higher logging overhead and more complex recovery logic, but guarantees correctness and enables advanced features

**SQLite** takes a significantly different approach optimized for embedded use cases and single-writer scenarios. SQLite's WAL mode (introduced as an alternative to its original rollback journal) prioritizes simplicity and minimal resource usage while still providing crash safety.

SQLite's WAL implementation uses a much simpler record format that primarily contains page-level changes rather than tuple-level modifications. Since SQLite operates with a single writer, it can avoid much of the complexity around concurrent transaction coordination that PostgreSQL must handle. WAL records in SQLite contain the page number, page size, and complete page contents after modification.

The SQLite recovery process is correspondingly simpler: during startup, the system reads the WAL file and applies any committed changes to the main database file. Since SQLite doesn't support multi-statement transactions with partial rollback, the recovery process primarily involves determining which complete transactions were committed and applying their changes in LSN order.

SQLite's checkpointing is also simplified—it involves copying committed changes from the WAL file back to the main database file and then truncating the WAL. This process can be performed automatically in the background or triggered explicitly by applications. The single-writer constraint eliminates many of the coordination challenges that make checkpointing complex in multi-user systems.

### Decision: Page-Level Logging in SQLite

- **Context:** SQLite targets embedded applications with single writers and limited memory/CPU resources
- **Options Considered:** Tuple-level logging (like PostgreSQL), operation-level logging, shadow paging
- **Decision:** Use page-level WAL records containing complete page images
- **Rationale:** Page-level logging is simpler to implement and debug; avoids complex tuple parsing during recovery; leverages SQLite's page-based storage architecture
- **Consequences:** Higher storage overhead for WAL files, but simpler recovery logic and better alignment with SQLite's architectural constraints

**MongoDB** represents a third approach that evolved from a document-oriented storage model and eventually added WAL for improved durability guarantees. MongoDB's WAL implementation (called the "oplog" in replica set contexts and "journal" for single-node durability) reflects the challenges of retrofitting logging onto a system originally designed without it.

MongoDB's WAL records operate at the operation level rather than the page or tuple level. Each WAL record describes a high-level operation like "insert document X into collection Y" or "update documents matching query Z with changes W." This approach aligns well with MongoDB's document-oriented data model and provides sufficient information for replica set synchronization as well as crash recovery.

The MongoDB recovery process reads the journal files to identify operations that were committed but might not have been applied to the data files. Since MongoDB uses memory-mapped files for data storage, the recovery process must be careful to coordinate with the operating system's virtual memory subsystem to ensure that recovered changes are properly synchronized to persistent storage.

MongoDB's journaling system uses group commit optimization extensively—multiple operations are batched together into single journal writes to amortize the cost of fsync operations. This approach significantly improves write throughput but complicates the recovery process, which must be able to handle partial batches and determine which operations within a batch were successfully committed.

## Decision: Operation-Level Logging in MongoDB

- **Context:** MongoDB's document-oriented data model and need for replica set synchronization
- **Options Considered:** Page-level logging, tuple-level logging with schema awareness
- **Decision:** Log high-level operations that can be replayed against the document storage engine
- **Rationale:** Operation-level logs can serve dual purpose for crash recovery and replica synchronization; natural fit for document-oriented operations; enables features like change streams
- **Consequences:** More complex operation replay logic; potential for replay inconsistencies if document schema changes; dependency on storage engine's idempotency guarantees

System	Record Granularity	Recovery Approach	Checkpoint Strategy	Primary Trade-off
PostgreSQL	Tuple-level with before/after images	Full ARIES (Analysis/Redo/Undo)	Fuzzy checkpoints	Correctness vs complexity
SQLite	Page-level complete images	Simple committed change replay	Background WAL copying	Simplicity vs storage overhead
MongoDB	Operation-level commands	Operation replay with group commit	Memory-mapped file synchronization	Flexibility vs replay complexity

These different approaches illuminate the key design tensions in WAL implementation:

**Logging granularity** affects both storage overhead and recovery complexity. Tuple-level logging provides fine-grained control and efficient storage usage but requires complex parsing during recovery. Page-level logging simplifies recovery but increases storage overhead. Operation-level logging provides semantic clarity but creates dependencies on storage engine idempotency.

**Recovery protocol sophistication** determines the guarantees the system can provide and the complexity of the implementation. Full ARIES provides mathematical correctness guarantees but requires significant implementation complexity. Simpler protocols reduce implementation burden but may provide weaker consistency guarantees or require more restrictive operating assumptions.

**Checkpoint coordination** balances recovery time bounds against runtime performance impact. Fuzzy checkpoints minimize performance disruption but require more sophisticated coordination logic. Simpler checkpoint approaches may block normal operations but are easier to implement correctly.

Understanding these trade-offs and seeing how production systems resolve them provides essential context for designing a WAL implementation that matches the requirements and constraints of the target use case.

## Implementation Guidance

### Technology Recommendations

Component	Simple Option	Advanced Option
File I/O	Standard library file operations ( <code>std::fs::File</code> )	Async I/O with <code>tokio</code> ( <code>tokio::fs::File</code> )
Serialization	Manual binary encoding with <code>byteorder</code> crate	Protocol Buffers with <code>prost</code> crate
Checksums	CRC32 with <code>crc32fast</code> crate	BLAKE3 cryptographic hash
Concurrent Access	<code>std::sync::Mutex</code> and <code>RwLock</code>	Lock-free structures with <code>crossbeam</code>
Error Handling	<code>Result&lt;T, E&gt;</code> with custom error types	<code>anyhow</code> for error chaining and context

### Recommended File Structure

Understanding the relationship between WAL concepts and code organization helps learners structure their implementation effectively from the beginning:

```

wal-project/
  src/
    main.rs           ← CLI tool for testing WAL operations
    lib.rs            ← Public API exports
    error.rs          ← WAL-specific error types

    log/
      mod.rs          ← Core logging functionality
      record.rs       ← LogRecord types and serialization (Milestone 1)
      writer.rs       ← LogWriter implementation (Milestone 2)
      reader.rs       ← LogReader for scanning log files
      segment.rs      ← Log file rotation and segment management

    recovery/
      mod.rs          ← Crash recovery implementation
      manager.rs      ← Module exports
      aries.rs         ← RecoveryManager main logic (Milestone 3)
      transaction_table.rs  ← ARIES algorithm implementation
      dirty_page_table.rs  ← Active transaction tracking
      ← Modified page tracking

    checkpoint/
      mod.rs          ← Checkpointing system
      manager.rs      ← Module exports
      fuzzy.rs         ← CheckpointManager (Milestone 4)
      ← Fuzzy checkpoint algorithm

    storage/
      mod.rs          ← Mock database for testing
      page.rs         ← Module exports
      buffer_pool.rs  ← Database page abstraction
      ← In-memory page cache

  tests/
    integration/
      crash_scenarios.rs  ← End-to-end crash recovery tests
      recovery_correctness.rs  ← Simulated crash testing
      ← ARIES algorithm verification

  examples/
    basic_wal.rs      ← Simple WAL usage demonstration
    crash_recovery_demo.rs  ← Recovery process walkthrough

```

This structure separates concerns clearly: the `log` module handles the mechanical aspects of writing and reading log files, the `recovery` module implements the complex ARIES algorithm, and the `checkpoint` module manages long-term log maintenance. The `storage` module provides a simple database abstraction for testing without requiring a full database implementation.

## Infrastructure Starter Code

**Error Types ( `src/error.rs` )** This complete error handling foundation supports all WAL operations and provides clear diagnostic information:

```
use std::io;

use thiserror::Error;

#[derive(Error, Debug)]

pub enum WalError {

    #[error("I/O error: {0}")]
    Io(#[from] io::Error),


    #[error("Log corruption detected at LSN {lsn}: {reason}")]
    Corruption { lsn: u64, reason: String },


    #[error("Invalid log record format: {0}")]
    InvalidFormat(String),


    #[error("Transaction {txn_id} not found")]
    TransactionNotFound { txn_id: u64 },


    #[error("Checkpoint failed: {0}")]
    CheckpointError(String),


    #[error("Recovery failed: {0}")]
    RecoveryError(String),


    #[error("Log segment full, rotation required")]
    SegmentFull,


}

pub type WalResult<T> = Result<T, WalError>;
```

```
// Helper for creating corruption errors with context

pub fn corruption_error(lsn: u64, reason: impl Into<String>) -> WalError {

    WalError::Corruption {

        lsn,
        reason: reason.into(),
    }
}
```

**Basic Log Segment Management ( `src/log/segment.rs` )** This provides the file-level abstraction for log rotation without implementing the core WAL logic:

```
use std::fs::{File, OpenOptions};

use std::io::{Read, Seek, SeekFrom, Write};

use std::path::{Path, PathBuf};

use crate::error::{WalResult, WalError};

pub struct LogSegment {

    file: File,

    path: PathBuf,

    current_size: u64,

    max_size: u64,

}

impl LogSegment {

    pub fn create(dir: &Path, segment_id: u64, max_size: u64) -> WalResult<Self> {

        let path = dir.join(format!("wal-{:08}.log", segment_id));

        let file = OpenOptions::new()

            .create(true)

            .write(true)

            .read(true)

            .open(&path)?;

        Ok(LogSegment {

            file,

            path,

            current_size: 0,

            max_size,

        })

    }

}
```

```
pub fn open(path: PathBuf) -> WalResult<Self> {
    let file = OpenOptions::new()
        .write(true)
        .read(true)
        .open(&path)?;

    let size = file.metadata()?.len();

    Ok(LogSegment {
        file,
        path,
        current_size: size,
        max_size: 64 * 1024 * 1024, // 64MB default
    })
}

pub fn append(&mut self, data: &[u8]) -> WalResult<u64> {
    let offset = self.current_size;
    self.file.seek(SeekFrom::End(0))?;
    self.file.write_all(data)?;
    self.current_size += data.len() as u64;
    Ok(offset)
}

pub fn force_sync(&mut self) -> WalResult<()> {
    self.file.sync_all()?;
}
```

```
    Ok(())

}

pub fn is_full(&self) -> bool {
    self.current_size >= self.max_size
}

pub fn read_at(&mut self, offset: u64, buf: &mut [u8]) -> WalResult<u8> {
    self.file.seek(SeekFrom::Start(offset))?;
    Ok(self.file.read(buf)?)
}

pub fn size(&self) -> u64 {
    self.current_size
}

pub fn path(&self) -> &Path {
    &self.path
}

}
```

**Mock Database Storage ( `src/storage/page.rs` )** This provides a simple database page abstraction for testing WAL operations:

```
use std::collections::HashMap;

use std::sync::{Arc, RwLock};

pub type PageId = u32;

pub type TransactionId = u64;

#[derive(Debug, Clone, PartialEq)]

pub struct DatabasePage {

    pub page_id: PageId,

    pub data: Vec<u8>,

    pub lsn: u64, // Last LSN that modified this page

}

impl DatabasePage {

    pub fn new(page_id: PageId, size: usize) -> Self {

        Self {

            page_id,

            data: vec![0; size],

            lsn: 0,

        }

    }

    pub fn write_data(&mut self, offset: usize, data: &[u8], lsn: u64) {

        let end = offset + data.len();

        if end <= self.data.len() {

            self.data[offset..end].copy_from_slice(data);

            self.lsn = lsn;

        }

    }

}
```

```
}

pub fn read_data(&self, offset: usize, len: usize) -> &[u8] {
    let end = (offset + len).min(self.data.len());
    &self.data[offset..end]
}

}

// Simple in-memory database for testing WAL
#[derive(Debug)]
pub struct MockDatabase {
    pages: Arc<RwLock<HashMap<PageId, DatabasePage>>>,
    page_size: usize,
}

impl MockDatabase {
    pub fn new(page_size: usize) -> Self {
        Self {
            pages: Arc::new(RwLock::new(HashMap::new())),
            page_size,
        }
    }

    pub fn get_page(&self, page_id: PageId) -> Option<DatabasePage> {
        self.pages.read().unwrap().get(&page_id).cloned()
    }

    pub fn update_page(&self, page: DatabasePage) {

```

```
self.pages.write().unwrap().insert(page.page_id, page);

}

pub fn create_page(&self, page_id: PageId) -> DatabasePage {
    let page = DatabasePage::new(page_id, self.page_size);
    self.pages.write().unwrap().insert(page_id, page.clone());
    page
}

}
```

## Core Logic Skeleton Code

**Log Record Types ( `src/log/record.rs` )** This skeleton defines the data structures learners will implement serialization for:

```
use crate::storage::PageId;

pub type LSN = u64;

pub type TransactionId = u64;

/// Core log record types that WAL must support

#[derive(Debug, Clone, PartialEq)]

pub enum LogRecord {

    Redo(RedoRecord),

    Undo(UndoRecord),

    Commit(CommitRecord),

    Abort(AbortRecord),

    Checkpoint(CheckpointRecord),

}

#[derive(Debug, Clone, PartialEq)]

pub struct RedoRecord {

    pub lsn: LSN,

    pub txn_id: TransactionId,

    pub page_id: PageId,

    pub offset: u32,

    pub after_image: Vec<u8>,

    // TODO: Add prev_lsn field for linking transaction's log records

}

#[derive(Debug, Clone, PartialEq)]

pub struct UndoRecord {

    pub lsn: LSN,

    pub txn_id: TransactionId,
```

```
pub page_id: PageId,  
  
pub offset: u32,  
  
pub before_image: Vec<u8>,  
  
// TODO: Add prev_lsn field for linking transaction's log records  
  
}  
  
// TODO: Implement CommitRecord, AbortRecord, CheckpointRecord structs  
  
// Hint: Commit/Abort need LSN and txn_id  
  
// Hint: Checkpoint needs LSN, active transaction list, and dirty page table  
  
impl LogRecord {  
  
    /// Serialize record to binary format with CRC checksum  
  
    pub fn serialize(&self) -> Vec<u8> {  
  
        // TODO 1: Create record header with record type, length, LSN  
  
        // TODO 2: Serialize record-specific data based on enum variant  
  
        // TODO 3: Calculate CRC32 checksum over header + data  
  
        // TODO 4: Append checksum to create final binary record  
  
        // Hint: Use byteorder crate for consistent endianness  
  
        // Hint: Use crc32fast crate for checksum calculation  
  
        unimplemented!("Learner implements binary serialization")  
  
    }  
  
    /// Deserialize record from binary format with integrity verification  
  
    pub fn deserialize(data: &[u8]) -> crate::error::WalResult<Self> {  
  
        // TODO 1: Parse record header (type, length, LSN)  
  
        // TODO 2: Verify record length matches actual data length  
  
        // TODO 3: Extract record data and stored checksum  
  
        // TODO 4: Calculate checksum over header + data, compare with stored value
```

```

    // TODO 5: Deserialize record-specific data based on type

    // TODO 6: Return appropriate LogRecord enum variant

    // Hint: Return WalError::Corruption if checksum mismatch

    // Hint: Return WalError::InvalidFormat for malformed data

    unimplemented!("Learner implements binary deserialization")

}

pub fn lsn(&self) -> LSN {

    // TODO: Return LSN field from whichever record variant this is

    // Hint: Use match statement on self

    unimplemented!("Extract LSN from any record type")

}

pub fn txn_id(&self) -> Option<TransactionId> {

    // TODO: Return transaction ID if this record type has one

    // Hint: Checkpoint records don't have transaction IDs

    unimplemented!("Extract transaction ID where applicable")

}

}

```

## Language-Specific Hints

### Rust-Specific Implementation Tips:

- Use `std::fs::OpenOptions` with `create(true).append(true)` for atomic log appends
- Call `File::sync_all()` to implement fsync semantics—this is crucial for durability
- Use `byteorder::WriteBytesExt` and `ReadBytesExt` for consistent binary serialization
- Implement `From<io::Error>` for your error types to use the `?` operator effectively
- Use `std::mem::size_of::<T>()` to calculate fixed-size field lengths for binary format
- Consider using `memmap2` crate for memory-mapped log reading in advanced implementations

### File I/O and Durability:

- Always call `sync_all()` after writing commit records—this is the core durability guarantee
- Use `OpenOptions::create_new(true)` when creating new log segments to avoid overwriting
- Handle `io::ErrorKind::WriteZero` to detect when disk space is exhausted
- Consider using `fallocate` (Linux) or `SetEndOfFile` (Windows) to pre-allocate log space

### Binary Format Design:

- Use fixed-size headers to make parsing predictable and efficient
- Store lengths in native byte order but document the choice clearly
- Place CRC checksums at the end of records so they can be calculated incrementally
- Use magic numbers or version fields to detect format incompatibilities

### Concurrency Considerations:

- Use `Arc<Mutex<T>>` for shared WAL writer state, or `Arc<RwLock<T>>` for reader-heavy workloads
- Consider channels (`std::sync::mpsc`) for batching log writes from multiple threads
- Be careful about lock ordering to prevent deadlocks between WAL and database locks

## Milestone Checkpoints

### After implementing Milestone 1 (Log Record Format):

Run this test to verify binary serialization works correctly:

```
cargo test log_record_serialization
```

BASH

Expected behavior:

- Create a `RedoRecord` with sample data, serialize it, deserialize it, and verify roundtrip equality
- Corrupt a single byte in the serialized data and verify deserialization fails with `WalError::Corruption`
- Test with various record sizes including empty and maximum-size data fields

### Manual verification steps:

1. Create log records with different transaction IDs and verify LSN ordering
2. Serialize multiple records and inspect the binary output—headers should have consistent format
3. Verify CRC checksums change when any field is modified

### Signs of correct implementation:

- Binary format is deterministic—same record always produces same bytes
- Corruption detection catches single-bit flips reliably
- Record sizes match header length fields exactly

### Common issues to debug:

- Endianness mismatches cause deserialization failures on different architectures

- Variable-length fields without proper length prefixes cause parsing to drift
- CRC calculated over wrong byte ranges fails to detect corruption in headers

## Goals and Non-Goals

**Milestone(s):** All milestones - this section establishes the scope and requirements that drive design decisions across Milestone 1 (Log Record Format), Milestone 2 (Log Writer), Milestone 3 (Crash Recovery), and Milestone 4 (Checkpointing)

Think of this WAL implementation as building a reliable flight data recorder for database operations. Just as aviation recorders must capture every critical event before it happens and survive catastrophic failures, our WAL must log every database change before it's applied and guarantee those logs survive crashes. However, unlike aircraft recorders that only need to preserve data for post-incident analysis, our WAL must actively use those logs to rebuild a consistent database state after failures.

This section establishes clear boundaries around what our WAL implementation will deliver, ensuring we build a robust learning platform while keeping complexity manageable. The goals directly map to the four milestones, with each functional goal requiring specific non-functional characteristics to work correctly in practice.

## Functional Goals

The core functional requirements define what our WAL implementation must accomplish to demonstrate proper understanding of crash recovery and log-structured storage. These goals map directly to the milestone deliverables and represent the minimum viable functionality needed for a working Write-Ahead Logging system.

### Durable Transaction Logging with ACID Guarantees

Our WAL must ensure that once a transaction commits, its effects survive any system crash or power failure. This requires implementing the write-ahead rule: all changes must be recorded in the log and forced to disk before the transaction can commit successfully. The system must support multiple concurrent transactions while maintaining isolation and atomicity guarantees.

Transaction Property	WAL Implementation Responsibility
Atomicity	Log all operations within a transaction; provide undo capability for incomplete transactions
Consistency	Ensure log records accurately reflect database state changes with before/after images
Isolation	Log records include transaction IDs to distinguish concurrent transaction effects
Durability	Force log records to disk via fsync before acknowledging transaction commit

### ARIES-Style Crash Recovery

The system must implement the full ARIES recovery protocol with its three phases: Analysis, Redo, and Undo. This represents the gold standard for database crash recovery and demonstrates sophisticated understanding of consistency restoration. Recovery must handle arbitrary crash points and restore the database to a consistent state that includes all committed transactions while removing effects of incomplete transactions.

The Analysis phase scans the log from the most recent checkpoint, building transaction and dirty page tables that capture the database state at crash time. The Redo phase replays all committed operations to ensure their effects are present in the database. The Undo phase rolls back any operations from transactions that were active but uncommitted at crash time.

Recovery Phase	Purpose	Key Data Structures	Output
Analysis	Determine database state at crash	Transaction table, Dirty page table	List of committed/active transactions
Redo	Replay committed changes	Redo records from log	Database with all committed changes
Undo	Rollback incomplete transactions	Undo records from log	Consistent database state

### Complete Log Record Management

The implementation must support all essential log record types needed for full transaction processing. Each record type serves a specific purpose in the recovery protocol and must include sufficient information for both redo and undo operations.

Record Type	Core Fields	Recovery Purpose
RedoRecord	lsn, txn_id, page_id, offset, after_image	Replay committed changes during Redo phase
UndoRecord	lsn, txn_id, page_id, offset, before_image	Rollback uncommitted changes during Undo phase
CommitRecord	lsn, txn_id	Mark transaction as committed for recovery decisions
AbortRecord	lsn, txn_id	Mark transaction as aborted, no redo needed
CheckpointRecord	lsn, active_transactions, dirty_pages	Establish recovery starting point

### Efficient Checkpointing with Log Truncation

The system must implement fuzzy checkpointing that allows normal transaction processing to continue during checkpoint creation. Checkpoints serve two critical functions: reducing recovery time by establishing a known consistent starting point, and enabling log truncation to prevent unbounded disk usage.

Fuzzy checkpoints capture a consistent snapshot of system state without blocking concurrent transactions. This requires careful coordination to ensure the checkpoint record accurately reflects which transactions were active and which pages contained uncommitted changes at the checkpoint moment.

**Key Design Insight:** Fuzzy checkpoints must record not just the current state, but enough information to handle transactions that begin or end while the checkpoint is being written. This is why we capture active transaction lists and dirty page sets atomically within the checkpoint record.

### Non-Functional Goals

The non-functional requirements ensure our WAL implementation performs adequately and handles real-world failure scenarios correctly. These requirements drive many of the detailed design decisions in component implementations.

### Performance Characteristics

While performance is not the primary learning objective, the WAL must demonstrate understanding of key performance techniques used in production systems. The implementation should exhibit reasonable performance characteristics that don't interfere with the learning experience.

Performance Aspect	Target Requirement	Implementation Technique
Log write throughput	Support concurrent writers without serialization bottlenecks	Lock-free append buffering with batch flushing
Recovery time	Complete recovery in under 30 seconds for 1GB log	Efficient log scanning with early termination optimizations
Checkpoint overhead	Less than 10% impact on normal transaction throughput	Fuzzy checkpointing with background processing
Memory usage	Bounded memory growth regardless of log size	Streaming log processing without loading entire log

## Crash Safety and Reliability

The WAL must correctly handle all realistic failure scenarios that could occur in practice. This includes not just clean shutdowns, but unexpected crashes, power failures, disk errors, and partial write scenarios.

Failure Scenario	Detection Method	Recovery Strategy
Power failure during log write	CRC checksum validation	Truncate to last valid record, continue recovery
Partial write to log file	Record length validation	Ignore incomplete record, treat as end of valid log
Log file corruption	CRC mismatch on read	Stop recovery, report corruption location for manual intervention
Disk full during logging	Write system call failure	Abort current transaction, signal application error
Recovery process crash	Recovery completion marker	Restart recovery from beginning, ensure idempotent operations

**Critical Safety Principle:** The WAL must never report a transaction as committed unless its log records are guaranteed to be durable on disk. This means fsync must complete successfully before any commit acknowledgment reaches the application.

## Concurrency and Isolation

The implementation must support multiple concurrent transactions accessing the WAL simultaneously. This requires careful synchronization design that maintains consistency without unnecessary serialization.

Concurrency Aspect	Requirement	Synchronization Strategy
Multiple writers	Support 10+ concurrent transactions	Lock-free append buffer with atomic sequence number allocation
Reader/writer coordination	Readers don't block writers	Read-only access to immutable log segments
Checkpoint coordination	Checkpoints don't block transactions	Snapshot active state without holding transaction locks
Recovery exclusivity	Only one recovery process at a time	File-based locking to prevent concurrent recovery attempts

## Resource Management

The WAL must demonstrate proper resource management practices, including bounded memory usage and predictable disk space consumption. This prevents resource exhaustion scenarios that could mask correctness bugs.

Log segment rotation ensures that individual log files don't grow unboundedly, making file management more predictable. The default segment size of 64MB provides a reasonable balance between file management overhead and segment size.

Resource Type	Management Policy	Implementation
Memory buffers	Fixed-size write buffers with overflow handling	1MB write buffer with force-flush when full
Log disk space	Automatic truncation after successful checkpoints	Remove log segments older than most recent checkpoint
File descriptors	Bounded number of open log segments	Close old segments after rotation, reopen for reading as needed
Lock duration	Minimize critical section time	Copy data to private buffers before I/O operations

## Explicit Non-Goals

Clearly defining what this WAL implementation will NOT include helps maintain focus on the core learning objectives. These exclusions are deliberate choices to keep the project scope manageable while still covering the essential concepts.

## Advanced Performance Optimizations

While the implementation should perform reasonably well, it explicitly excludes sophisticated performance optimizations that would add complexity without proportional educational value. These optimizations are

important in production systems but don't enhance understanding of the core recovery algorithms.

Excluded Optimization	Rationale for Exclusion	Alternative Learning Path
Group commit batching	Adds complexity to fsync timing without changing recovery logic	Can be added as future extension after core implementation works
Parallel recovery processing	Requires complex dependency analysis between log records	Focus on correctness first, then optimize recovery in advanced projects
Log compression	Compression/decompression obscures log record format learning	Raw binary format makes debugging and inspection easier
Asynchronous checkpointing	Complex coordination between checkpoint and normal operations	Synchronous checkpoints are simpler and still demonstrate key concepts

## Distributed System Features

This implementation targets single-node database systems only. Distributed features require understanding of distributed consensus and replication protocols that are separate learning objectives from crash recovery.

Distributed Feature	Why Excluded	Single-Node Alternative
Multi-node replication	Requires distributed consensus algorithms (Raft, Paxos)	Single node provides full crash recovery learning
Distributed transactions	Needs two-phase commit coordination	Local transactions demonstrate ACID properties sufficiently
Cross-datacenter WAL shipping	Network partitions and consistency models add complexity	Focus on local durability guarantees first
Byzantine fault tolerance	Requires cryptographic verification beyond crash failures	Crash failures are the most common real-world scenario

## Production Database Integration

The implementation uses a simplified mock database rather than integrating with a full database engine. This allows focusing on WAL-specific concepts without getting distracted by query processing, indexing, or storage engine details.

Database Feature	Exclusion Rationale	Mock Implementation
SQL query processing	Query optimization and execution are separate learning domains	Direct page-level operations with simple read/write interface
Index management	B-tree maintenance doesn't interact with WAL recovery logic	Flat key-value storage using HashMap for simplicity
Buffer pool management	Buffer replacement policies are orthogonal to WAL concepts	In-memory pages with explicit durability control
Lock management	Concurrency control has separate learning objectives	Transaction isolation through sequential log record processing

## Advanced Recovery Features

Several sophisticated recovery features are excluded to keep the implementation focused on core ARIES concepts. These features are valuable in production but add complexity that can obscure the fundamental recovery algorithm.

Advanced Feature	Educational Trade-off	Core Alternative
Point-in-time recovery	Requires timestamp indexing and selective replay	Full recovery to crash point demonstrates core concepts
Logical log records	Abstract operations need application-specific undo logic	Physical before/after images work for any data format
Incremental checkpointing	Complex dirty page tracking across multiple checkpoints	Full checkpoint captures complete state simply
Log archive management	Long-term storage policies don't affect recovery correctness	Focus on active log management sufficient for learning

**Scope Management Principle:** Each excluded feature represents a deliberate choice to prioritize depth over breadth. Students will gain thorough understanding of core WAL concepts rather than superficial exposure to many advanced features.

## Error Recovery Beyond Corruption Detection

While the implementation detects log corruption and reports it clearly, it doesn't attempt sophisticated corruption repair or alternative recovery strategies. This keeps error handling focused on detection and clean failure rather than complex repair logic.

Error Scenario	Detection Approach	Response Strategy
CRC mismatch in log record	Checksum validation during log scanning	Report corruption location, halt recovery with clear error
Truncated log file	Record boundary validation	Stop processing at last complete record, continue recovery
Missing log segment	File existence check during scanning	Report missing segment, require manual intervention
Invalid log record format	Structure validation during deserialization	Report format error with specific field that failed validation

This approach teaches proper error detection principles while avoiding the complexity of automated repair mechanisms that could mask underlying correctness issues.

## Implementation Guidance

The goals and non-goals directly influence technology choices and implementation structure. This guidance helps translate the high-level requirements into concrete technical decisions for the four milestones.

### Technology Recommendations

Component	Simple Option	Advanced Option
Binary Serialization	Custom fixed-format structs	Protocol Buffers with schema evolution
File I/O	Standard library File + manual fsync	Memory-mapped files with explicit sync regions
Concurrency	RwLock for critical sections	Lock-free atomic operations with hazard pointers
Testing	Unit tests with mock crashes	Property-based testing with crash injection
Error Handling	Result types with detailed error variants	Error chaining with full causality tracking

### Project Structure Aligned with Goals

The file organization should reflect the goal-driven component separation, making it clear how each module contributes to the overall durability guarantee:

```
wal-implementation/
├── src/
│   ├── lib.rs                      # Public API exposing WalManager
│   ├── log_record.rs               # Milestone 1: All LogRecord variants and serialization
│   ├── log_writer.rs               # Milestone 2: LogSegment management and append
├── operations
│   ├── recovery.rs                 # Milestone 3: ARIES recovery implementation
│   ├── checkpoint.rs               # Milestone 4: CheckpointManager and fuzzy checkpoint
├── logic
│   ├── mock_database.rs            # Simplified database for demonstrating WAL operations
│   └── error.rs                   # WalError variants aligned with goal requirements
└── tests/
    ├── integration_tests.rs       # End-to-end crash recovery scenarios
    ├── crash_simulation.rs        # Power failure and corruption testing
    └── performance_benchmarks.rs  # Verify non-functional goal compliance
└── examples/
    ├── basic_transaction.rs       # Simple transaction demonstrating durability
    └── crash_recovery_demo.rs     # Recovery walkthrough with explanatory output
```

## Core Type Definitions Supporting Goals

The fundamental types directly implement the functional goals, with each field serving a specific purpose in the durability guarantee:

```
// Primary goal: Support all log record types needed for ARIES recovery

#[derive(Debug, Clone)]

pub enum LogRecord {

    Redo(RedoRecord),

    Undo(UndoRecord),

    Commit(CommitRecord),

    Abort(AbortRecord),

    Checkpoint(CheckpointRecord),
}

// Goal: Provide complete information for replaying committed changes

#[derive(Debug, Clone)]

pub struct RedoRecord {

    pub lsn: LSN,                                // TODO: Implement monotonic LSN generation

    pub txn_id: TransactionId,                   // TODO: Link to transaction management

    pub page_id: PageId,                          // TODO: Identify target page for change

    pub offset: u32,                             // TODO: Specify exact location within page

    pub after_image: Vec<u8>,                   // TODO: Store new data value for redo operation
}

// Goal: Support rolling back uncommitted transactions

#[derive(Debug, Clone)]

pub struct UndoRecord {

    pub lsn: LSN,                                // TODO: Maintain log ordering for undo sequence

    pub txn_id: TransactionId,                  // TODO: Identify which transaction to roll back

    pub page_id: PageId,                          // TODO: Target page for rollback operation

    pub offset: u32,                             // TODO: Exact location to restore

    pub before_image: Vec<u8>,                  // TODO: Original data value before change
}
```

```
}
```

## Durability Implementation Pattern

The core durability guarantee requires a specific sequence of operations that must be followed consistently:

```
impl LogWriter {  
  
    // Goal: Ensure durability before commit acknowledgment  
  
    pub fn write_and_force(&mut self, record: &LogRecord) -> WalResult<LSN> {  
  
        // TODO 1: Serialize log record with CRC checksum  
  
        // TODO 2: Append to current log segment atomically  
  
        // TODO 3: Call fsync to force data to disk  
  
        // TODO 4: Return LSN only after successful fsync  
  
        // TODO 5: Handle segment rotation if current segment is full  
  
        todo!()  
  
    }  
}
```

RUST

## Recovery Goal Implementation Skeleton

The ARIES recovery goals require implementing the three-phase algorithm with proper state management:

```
impl RecoveryManager {  
  
    // Goal: Restore database to consistent state after crash  
  
    pub fn recover(&mut self) -> WalResult<()> {  
  
        // TODO Phase 1: Analysis - scan log and build transaction/dirty page tables  
  
        // TODO Phase 2: Redo - replay all committed transaction changes  
  
        // TODO Phase 3: Undo - rollback all uncommitted transaction changes  
  
        // TODO Phase 4: Write recovery completion record to log  
  
        todo!()  
  
    }  
  
    // Goal: Build recovery state from log contents  
  
    fn analysis_pass(&mut self) -> WalResult<AnalysisResult> {  
  
        // TODO 1: Start from most recent checkpoint record  
  
        // TODO 2: Scan forward through all log records  
  
        // TODO 3: Track active transactions in transaction table  
  
        // TODO 4: Track dirty pages in dirty page table  
  
        // TODO 5: Identify committed vs uncommitted transactions  
  
        todo!()  
  
    }  
}
```

## Checkpoint Goal Verification

After implementing checkpointing, verify that the goals are met through specific behavioral tests:

Checkpoint Behavior	Verification Method	Expected Outcome
Recovery starts from checkpoint	Create checkpoint, crash, restart	Recovery scans from checkpoint LSN, not log beginning
Fuzzy checkpoint doesn't block	Start long checkpoint, run transactions	Transactions complete normally during checkpoint
Log truncation works	Create checkpoint, restart	Log segments before checkpoint are removed
Active transactions recorded	Checkpoint during active transaction	Recovery correctly identifies transaction as active

## Common Implementation Pitfalls

**⚠ Pitfall: Committing Before Fsync** Many implementations incorrectly acknowledge transaction commits before the log records are durable on disk. This violates the fundamental durability guarantee and can lead to committed transactions disappearing after crashes.

**Why it's wrong:** If the system crashes after acknowledging the commit but before the fsync completes, the transaction's log records may be lost from the operating system's buffer cache.

**How to fix:** Always call `fsync()` on the log file and wait for it to return successfully before acknowledging any transaction commit to the client.

**⚠ Pitfall: Non-Idempotent Recovery Operations** Recovery operations must be idempotent because recovery might be interrupted and restarted. If redo operations aren't idempotent, restarting recovery could leave the database in an inconsistent state.

**Why it's wrong:** If recovery crashes partway through and restarts, some operations might be applied twice, leading to incorrect final state.

**How to fix:** Design redo operations to check current page state and only apply changes if needed, or use LSN tracking to avoid reapplying already-processed operations.

**⚠ Pitfall: Incomplete Checkpoint Information** Fuzzy checkpoints must capture enough information to handle transactions that start or end during checkpoint creation. Missing this information makes recovery incorrect.

**Why it's wrong:** If a transaction starts after the checkpoint begins scanning but before the checkpoint record is written, recovery might miss this transaction entirely.

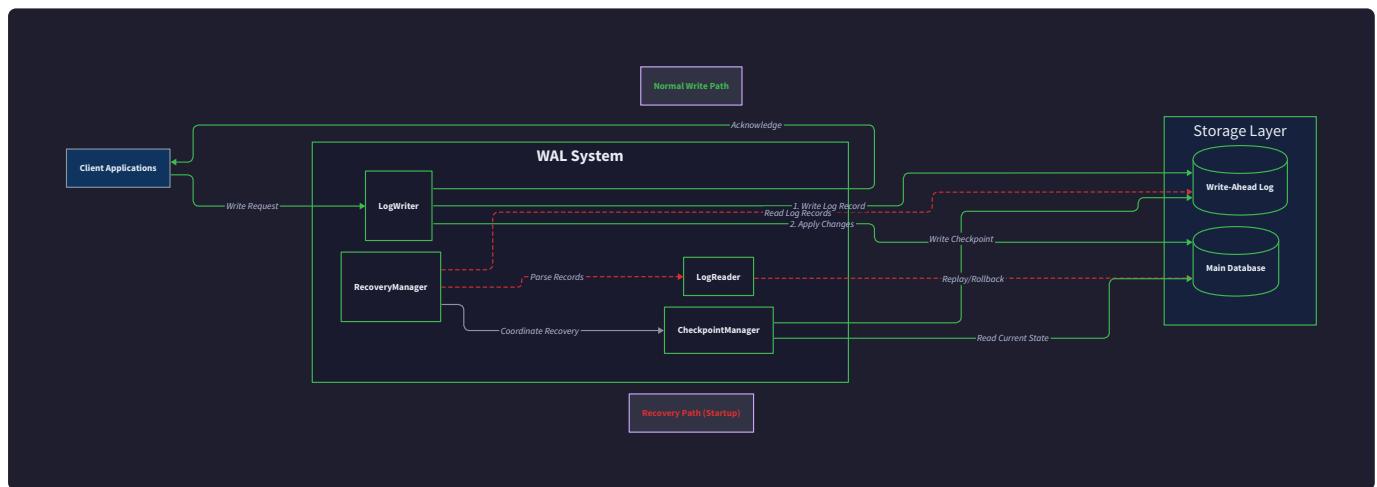
**How to fix:** Use atomic snapshots of transaction and dirty page tables, and ensure checkpoint records include all transactions that could possibly affect recovery.

# High-Level Architecture

**Milestone(s):** All milestones - this section establishes the foundational architecture that supports Milestone 1 (Log Record Format), Milestone 2 (Log Writer), Milestone 3 (Crash Recovery), and Milestone 4 (Checkpointing)

Think of the Write-Ahead Logging system as a sophisticated flight recorder for your database. Just as an aircraft's flight recorder captures every critical system event and pilot action before they occur, our WAL system captures every database change in a sequential log before it touches the actual data files. If the "aircraft" crashes, we can replay the "flight recorder" to reconstruct exactly what happened and restore the system to a consistent state.

The WAL architecture consists of four primary components that work together like a well-orchestrated emergency response team. The **LogWriter** acts as the dispatcher, ensuring every change is documented and safely stored before any action is taken. The **RecoveryManager** serves as the detective, analyzing the log after a crash to determine what transactions were completed and which need to be undone. The **CheckpointManager** functions as the archivist, periodically creating snapshots that allow recovery to start from a recent known-good state rather than replaying the entire history. Finally, the **LogReader** acts as the interpreter, parsing the binary log records and making them understandable to the other components.



## Component Overview

The WAL system architecture is designed around four core subsystems that each handle distinct aspects of the logging and recovery process. This separation of concerns ensures that each component can be developed, tested, and reasoned about independently while maintaining clear interfaces between them.

### LogWriter Component

The **LogWriter** component serves as the primary entry point for all database modifications. Think of it as a meticulous secretary who must document every decision before any action is taken. When a transaction wants to modify data, it must first tell the LogWriter what it intends to do. The LogWriter creates appropriate

log records, assigns them monotonically increasing log sequence numbers (LSNs), and ensures they are durably written to disk before allowing the transaction to proceed.

Responsibility	Description	Key Operations
Record Creation	Generate properly formatted log records with LSNs, transaction IDs, and operation details	<code>create_redo_record()</code> , <code>create_undo_record()</code> , <code>create_commit_record()</code>
Sequential Writing	Append records to the active log segment in strict LSN order	<code>append()</code> , <code>write_and_force()</code>
Durability Enforcement	Ensure log records reach persistent storage via <code>fsync</code> before acknowledging writes	<code>force_sync()</code> , <code>group_commit()</code>
Buffer Management	Batch multiple log records in memory to amortize I/O costs while maintaining ordering	<code>flush_buffer()</code> , <code>buffer_record()</code>
Segment Management	Rotate to new log files when current segment reaches size limits	<code>rotate_segment()</code> , <code>create_new_segment()</code>

The LogWriter maintains an in-memory buffer that collects log records before flushing them to disk in batches. This group commit optimization significantly improves performance by allowing multiple transactions to share the cost of a single `fsync` operation. However, the LogWriter must be extremely careful about ordering - records must be written to the log file in exactly the same order as their LSNs, and no transaction can be told its commit succeeded until its commit record is safely on disk.

**Design Insight:** The LogWriter's append-only design is crucial for both performance and correctness. Sequential writes are much faster than random writes on traditional storage media, and the append-only pattern makes it impossible to accidentally overwrite existing log records during normal operation.

## RecoveryManager Component

The `RecoveryManager` implements the ARIES (Algorithm for Recovery and Isolation Exploiting Semantics) recovery protocol, which is the gold standard for database crash recovery. Think of it as a forensic investigator who arrives at the scene of a crash and must piece together exactly what happened by examining the evidence left in the log files.

The RecoveryManager operates in three distinct phases, each with a specific purpose in reconstructing a consistent database state:

Recovery Phase	Purpose	Key Data Structures	Primary Operations
Analysis	Scan the log to identify committed and active transactions at crash time	Transaction Table, Dirty Page Table	<code>analysis_pass()</code> , <code>scan_log_records()</code> , <code>build_transaction_table()</code>
Redo	Replay all changes from committed transactions to ensure durability	Dirty Page Table, Log Records	<code>redo_pass()</code> , <code>apply_redo_record()</code> , <code>update_page_lsn()</code>
Undo	Roll back all changes from transactions that were active at crash time	Transaction Table, Undo Chain	<code>undo_pass()</code> , <code>apply_undo_record()</code> , <code>follow_undo_chain()</code>

The Analysis phase builds two critical data structures by scanning the log from the most recent checkpoint to the end. The Transaction Table tracks which transactions were active, committed, or aborted at the time of the crash. The Dirty Page Table identifies which database pages had uncommitted modifications and therefore need to be examined during the Redo phase.

During the Redo phase, the RecoveryManager replays all operations from committed transactions, regardless of whether those changes actually made it to the database files before the crash. This ensures that no committed work is lost. The Redo phase applies changes in LSN order, which guarantees that pages are updated in the same sequence as the original execution.

The Undo phase works backwards through the log, following undo chains to roll back all operations performed by transactions that were still active at crash time. This ensures that no partial or uncommitted work remains visible in the recovered database state.

**Critical Design Principle:** ARIES recovery is idempotent - it can be run multiple times safely. If recovery itself is interrupted by another crash, restarting recovery will produce the same final database state.

## CheckpointManager Component

The `CheckpointManager` periodically creates **fuzzy checkpoints** that serve as recovery starting points. Imagine a bookmark in a very long novel - instead of starting from page 1 every time you want to resume reading, you can jump directly to your bookmark. Similarly, checkpoints allow recovery to start from a recent known state rather than scanning the entire log history from the beginning.

Checkpoint Component	Purpose	Data Captured	Timing Considerations
Active Transaction Snapshot	Record which transactions were in progress when checkpoint began	Transaction IDs, current LSNs, undo chain pointers	Must be consistent with Dirty Page Table
Dirty Page Recording	Identify which pages have uncommitted changes in memory buffers	Page IDs, oldest LSN that modified each page	Critical for determining Redo scan starting point
Checkpoint Coordination	Ensure checkpoint state is consistent without blocking normal operations	Master record location, checkpoint completion LSN	Must handle concurrent transaction commits during checkpoint
Log Truncation Planning	Mark old log segments that can be safely deleted after checkpoint	Oldest required LSN, reclaimable segment list	Cannot truncate segments still needed for active transaction undo

Fuzzy checkpoints are "fuzzy" because they allow normal transaction processing to continue while the checkpoint is being created. This is achieved by capturing a snapshot of the system state at a specific LSN, but allowing newer transactions to continue executing and logging their operations. The checkpoint record includes enough information for recovery to determine exactly which transactions were active at checkpoint time and which pages might need redo operations.

The CheckpointManager must coordinate carefully with the LogWriter to ensure that checkpoint records are durably written and that the master record (which points to the most recent valid checkpoint) is updated atomically. If a crash occurs while a checkpoint is being written, recovery will simply use the previous valid checkpoint and scan a bit more of the log.

**Performance Trade-off:** More frequent checkpoints reduce recovery time but consume I/O bandwidth and CPU cycles. Less frequent checkpoints improve normal-case performance but increase crash recovery time. Most systems use adaptive checkpointing that triggers based on log volume or elapsed time.

## LogReader Component

The `LogReader` component handles the complex task of parsing binary log records and making them available to other components in a structured format. Think of it as a translator who converts the compact binary format used for efficient storage into the rich object representations needed for recovery processing.

LogReader Capability	Technical Challenge	Implementation Strategy	Error Handling
Binary Deserialization	Converting packed binary data back to structured log records	Fixed-size headers with variable-length payloads	CRC validation, format version checks
Record Boundary Detection	Finding record boundaries in potentially corrupted log files	Length prefixes, magic numbers, checksums	Partial record detection and skipping
Backwards Scanning	Reading log records in reverse LSN order for undo processing	Reverse iteration through log segments	Handle segment boundaries and corruption
Corruption Recovery	Continuing operation when individual records are damaged	Skip corrupted records, maintain scan position	Log corruption warnings, stop at severe damage
Segment Coordination	Reading across multiple log segment files seamlessly	LSN-to-file mapping, segment boundary handling	Missing segment detection, read-only fallback

The LogReader must handle several challenging scenarios that don't occur in normal forward-only processing. During the Undo phase, it needs to traverse log records in reverse LSN order, which requires sophisticated segment management. When corruption is detected, it must decide whether to skip individual records or abort recovery entirely. The LogReader also implements efficient seeking to specific LSNs, which is needed when recovery starts from a checkpoint in the middle of the log.

**Robustness Consideration:** The LogReader is often the first component to encounter log corruption after a crash. Its error handling decisions directly impact whether recovery succeeds or fails, making robust corruption detection and recovery strategies essential.

## Data Flow

Understanding how data flows through the WAL system during normal operations and crash recovery is essential for implementing the components correctly. The data flow patterns reveal the critical dependencies between components and highlight where durability guarantees must be enforced.

### Normal Operation Write Path

During normal database operation, every write transaction follows a carefully orchestrated sequence that ensures durability while maintaining performance. This sequence represents the core contract that applications rely on: once a transaction is acknowledged as committed, its changes will survive any subsequent crash.

The write transaction data flow follows this sequence:

1. **Transaction Begins:** The application starts a transaction and receives a unique `TransactionId` from the transaction manager. This identifier will tag all log records generated by this transaction, enabling recovery to group related operations together.
2. **Operation Logging:** For each database modification within the transaction, the application generates both redo and undo log records before making any changes to data pages. The redo record captures the after-image (what the data should look like after the change), while the undo record captures the before-image (what the data looked like before the change).
3. **Log Record Creation:** The `LogWriter` receives the operation details and constructs properly formatted log records. Each record receives a unique, monotonically increasing LSN that establishes the global order of all operations across all transactions.
4. **Buffer Accumulation:** The LogWriter adds the new log records to its in-memory buffer rather than immediately writing to disk. This batching allows multiple transactions to share the cost of I/O operations through group commit optimization.
5. **Database Modification:** After the log records are buffered, the transaction can modify the actual database pages. Each modified page is tagged with the LSN of the most recent operation that changed it, creating a clear connection between log records and data changes.
6. **Transaction Commit:** When the transaction is ready to commit, it sends a commit request to the LogWriter. The LogWriter creates a `CommitRecord` and adds it to the buffer.
7. **Force Write:** The LogWriter flushes its buffer to the log file and calls `fsync` to ensure all buffered records (including the commit record) reach persistent storage. Only after the `fsync` completes successfully can the transaction be acknowledged as committed.
8. **Commit Acknowledgment:** The application receives confirmation that the transaction committed successfully. At this point, the durability guarantee is in effect - these changes will be recovered even if a crash occurs immediately.

Data Flow Stage	Component Involved	Data Transformed	Durability Checkpoint
Begin Transaction	Transaction Manager	Generate <code>TransactionId</code>	None - transaction not yet durable
Log Operations	LogWriter	Operations → Log Records with LSNs	None - records still in memory
Modify Pages	Database Buffer Manager	Apply changes, tag pages with LSNs	None - changes might be in memory only
Commit Request	LogWriter	Create <code>CommitRecord</code>	None - commit record not yet persistent
Force Log	LogWriter	Buffer → Disk + fsync	<b>Critical Point</b> - transaction now durable
Acknowledge	Transaction Manager	Success response to application	Durability guarantee in effect

The critical insight in this data flow is that there are exactly two durability checkpoints: the log force operation and the eventual writing of dirty pages to the database files. The log force is mandatory and occurs synchronously during transaction commit. The database page writes can happen asynchronously later, because the log contains enough information to reconstruct the changes if needed.

**Performance vs. Durability Trade-off:** The synchronous fsync during commit is the most expensive operation in the write path, often taking milliseconds compared to microseconds for in-memory operations. Group commit batching allows the system to amortize this cost across multiple concurrent transactions.

## Recovery Data Flow

When the database starts up after a crash, the recovery data flow reverses the normal operation process, using the durable log records to reconstruct a consistent database state. This process must handle the fact that the crash could have occurred at any point during normal operation, potentially leaving transactions in various states of completion.

The recovery data flow implements the three-phase ARIES algorithm:

### Analysis Phase Flow:

- Checkpoint Location:** The `RecoveryManager` reads the master record to find the most recent valid checkpoint, providing a starting point that reduces the amount of log that needs to be scanned.
- Log Scanning:** Starting from the checkpoint, the `LogReader` sequentially reads log records until it reaches the end of the log. This scan identifies all transactions and data page modifications that occurred after the checkpoint.

3. **State Reconstruction:** For each log record encountered, the RecoveryManager updates its internal data structures - adding transactions to the Transaction Table, recording page modifications in the Dirty Page Table, and tracking transaction status changes (commits, aborts).

4. **Transaction Classification:** At the end of the scan, the RecoveryManager knows exactly which transactions committed successfully, which were explicitly aborted, and which were still active at crash time and need to be rolled back.

#### Redo Phase Flow:

1. **Forward Replay:** The RecoveryManager scans forward through the log again, this time applying all redo operations from committed transactions to ensure their changes are present in the database files.
2. **Page LSN Checking:** Before applying each redo operation, the system checks the LSN stored on the target database page. If the page's LSN is already greater than or equal to the redo record's LSN, the operation is skipped (it was already applied before the crash).
3. **Idempotent Application:** Redo operations are applied idempotently, meaning they can be safely executed multiple times. This property allows recovery to be interrupted and restarted without corrupting the database.

#### Undo Phase Flow:

1. **Reverse Processing:** The RecoveryManager identifies all transactions that were still active at crash time and begins processing their operations in reverse chronological order (highest LSN to lowest LSN).
2. **Undo Chain Following:** For each active transaction, the system follows the chain of undo records backward, applying the before-images to remove the effects of uncommitted operations.
3. **Compensation Logging:** As undo operations are applied, the system writes compensating log records (CLRs) that record the undo actions. These CLRs ensure that if recovery is interrupted, the undo work won't be lost.

Recovery Phase	Scan Direction	Records Processed	Database Changes	End State
Analysis	Forward (Checkpoint → End)	All record types	None - read-only analysis	Transaction/Dirty Page Tables built
Redo	Forward (Checkpoint → End)	Redo records from committed transactions	Apply committed changes	All committed work restored
Undo	Backward (End → sufficient point)	Undo records from active transactions	Remove uncommitted changes	Only committed transactions visible

**Recovery Correctness Guarantee:** After recovery completes, the database state will be identical to what it would have been if the system had processed all committed transactions and then cleanly shut down, with no uncommitted transactions ever having been started.

## Checkpoint Data Flow

Checkpointing creates a different data flow pattern that runs concurrently with normal transaction processing. The goal is to create a consistent snapshot that can serve as a recovery starting point without disrupting ongoing operations.

The fuzzy checkpoint data flow operates as follows:

1. **Checkpoint Initiation:** The CheckpointManager decides to create a checkpoint based on configurable criteria (elapsed time, log volume, or administrative request). It begins by recording the current LSN as the checkpoint's starting point.
2. **Active State Capture:** The system takes a snapshot of all currently active transactions, recording their IDs, current states, and undo chain information. This snapshot represents the transactions that would need to be handled specially if recovery started from this checkpoint.
3. **Dirty Page Enumeration:** The buffer manager provides a list of all dirty pages (pages modified but not yet written to disk) along with the oldest LSN that modified each page. This information tells recovery how far back in the log it needs to scan to ensure all changes to each page are properly redone.
4. **Concurrent Operation Continuation:** While the checkpoint data is being collected and written, normal transaction processing continues uninterrupted. New transactions can start, existing transactions can commit or abort, and pages can be modified. The checkpoint represents a consistent point-in-time snapshot despite this concurrent activity.
5. **Checkpoint Record Creation:** The CheckpointManager creates a `CheckpointRecord` containing all the collected state information and assigns it the next available LSN. This record is written to the log just like any other log record.
6. **Master Record Update:** After the checkpoint record is safely written and synced to disk, the system updates the master record to point to this new checkpoint. The master record update must be atomic to ensure recovery always finds a valid checkpoint.
7. **Log Truncation Opportunity:** With a new checkpoint established, the system can potentially truncate older log segments that are no longer needed for recovery, freeing disk space.

**Checkpoint Consistency Model:** Fuzzy checkpoints are "consistent" in the sense that they represent a valid state the system could have been in, even though some recorded active transactions might have committed by the time the checkpoint record is written.

## File Organization

The physical organization of the WAL implementation requires careful consideration of module boundaries, data encapsulation, and testing strategies. A well-organized file structure makes the system easier to understand, test, and extend while enforcing clear separation of concerns between components.

The recommended file organization follows standard software engineering practices while accommodating the specific needs of a crash recovery system. Each component should be isolated in its own module with clearly defined public interfaces, comprehensive test coverage, and minimal dependencies on other components.

### Architectural Decision: Modular Component Design

- **Context:** WAL systems are complex with multiple interacting components that must be reliable and testable
- **Options Considered:**
  1. Monolithic design with all components in a single module
  2. Component-based design with separate modules
  3. Layered architecture with strict dependency hierarchies
- **Decision:** Component-based design with separate modules and clear interfaces
- **Rationale:** Enables independent testing, parallel development, clear ownership of responsibilities, and easier debugging of component interactions
- **Consequences:** Slightly more complex initial setup, but much better maintainability, testability, and extensibility

## Core Module Structure

The WAL implementation should be organized into distinct modules that reflect the component architecture while providing shared utilities and clear testing boundaries.

```
wal-implementation/
├── src/
│   ├── lib.rs                         # Public API and module exports
│   ├── types.rs                        # Shared type definitions and constants
│   ├── errors.rs                       # Error types and result handling
│   ├── log_writer/
│   │   ├── mod.rs                      # LogWriter public interface
│   │   ├── writer.rs                   # Core LogWriter implementation
│   │   ├── buffer.rs                  # Write buffering and group commit
│   │   ├── segment.rs                 # Log segment management and rotation
│   │   └── sync.rs                     # fsync and durability operations
│   ├── log_reader/
│   │   ├── mod.rs                      # LogReader public interface
│   │   ├── reader.rs                  # Sequential and random log reading
│   │   ├── parser.rs                  # Binary record deserialization
│   │   └── cursor.rs                  # Position tracking and seeking
│   ├── recovery/
│   │   ├── mod.rs                      # RecoveryManager public interface
│   │   ├── manager.rs                 # ARIES recovery coordination
│   │   ├── analysis.rs                # Analysis phase implementation
│   │   ├── redo.rs                     # Redo phase implementation
│   │   ├── undo.rs                     # Undo phase implementation
│   │   └── state.rs                   # Transaction and dirty page tables
│   ├── checkpoint/
│   │   ├── mod.rs                      # CheckpointManager public interface
│   │   ├── manager.rs                 # Checkpoint coordination
│   │   ├── fuzzy.rs                   # Fuzzy checkpoint algorithm
│   │   └── truncation.rs              # Log truncation after checkpoints
│   ├── records/
│   │   ├── mod.rs                      # Log record type definitions
│   │   ├── format.rs                  # Binary serialization/deserialization
│   │   ├── validation.rs              # CRC checking and corruption detection
│   │   └── builder.rs                 # Helper functions for creating records
│   └── storage/
│       ├── mod.rs                      # Storage abstraction interface
│       ├── file_storage.rs            # File system based storage
│       ├── memory_storage.rs          # In-memory storage for testing
│       └── mock_database.rs           # Mock database for testing WAL
└── tests/
    ├── integration/
    │   ├── end_to_end.rs               # Full WAL + recovery integration tests
    │   ├── crash_simulation.rs        # Simulated crash and recovery tests
    │   └── checkpoint_recovery.rs    # Checkpoint-based recovery tests
    └── common/
        ├── mod.rs                      # Shared test utilities
        ├── fixtures.rs                 # Test data generation
        └── assertions.rs               # Custom test assertions for WAL
└── examples/
    ├── basic_usage.rs               # Simple WAL usage example
    ├── recovery_demo.rs             # Crash recovery demonstration
    └── checkpoint_demo.rs           # Checkpointing example
└── benches/
    └── write_throughput.rs          # LogWriter performance benchmarks
```

```

└── recovery_time.rs          # Recovery performance benchmarks
└── checkpoint_overhead.rs    # Checkpoint performance impact

```

## Module Responsibility Matrix

Each module has clearly defined responsibilities and interfaces that other modules can depend on. This organization enforces the architectural boundaries and makes testing much more manageable.

Module	Primary Responsibility	Public Interface	Key Dependencies	Testing Strategy
types	Shared type definitions, constants	Type aliases, enums, constants	None	Unit tests for serialization, validation
records	Log record format and serialization	<code>LogRecord</code> enum, serialize/deserialize	types, errors	Property-based testing of round-trip serialization
log_writer	Durable sequential log writing	<code>LogWriter</code> struct, write/force methods	records, storage	Unit tests + crash simulation
log_reader	Sequential and random log reading	<code>LogReader</code> struct, read/seek methods	records, storage	Unit tests + corruption handling
recovery	ARIES crash recovery implementation	<code>RecoveryManager</code> struct, recover method	log_reader, records	Integration tests with simulated crashes
checkpoint	Fuzzy checkpoint creation and management	<code>CheckpointManager</code> struct, checkpoint methods	log_writer, recovery	Integration tests with concurrent operations
storage	Storage abstraction for testability	<code>Storage</code> trait, file/memory implementations	types, errors	Unit tests + mock implementations

## Interface Design Principles

The module interfaces are designed to be testable, composable, and easy to reason about. Each module exposes only the minimum necessary interface while hiding implementation details that might change.

**Dependency Injection Pattern:** Components accept their dependencies through constructor parameters rather than creating them internally. This makes testing much easier because dependencies can be mocked or stubbed.

**Error Propagation:** All modules use the shared `WalResult<T>` type for error handling, ensuring consistent error propagation throughout the system. Errors include enough context to diagnose problems during development and debugging.

**Async Boundary Management:** The interface clearly distinguishes between synchronous operations (memory-only) and operations that might block on I/O. This helps implementers understand when they need to consider thread safety and when operations might fail due to external factors.

Interface Design Principle	Example	Benefit	Testing Impact
Dependency Injection	<code>LogWriter::new(storage: Box&lt;dyn Storage&gt;)</code>	Testable with mock storage	Can test without real files
Builder Pattern	<code>LogRecord::builder().lsn(1).txn_id(42).build()</code>	Flexible record creation	Easy to create test fixtures
Result Types	<code>fn recover(&amp;mut self) -&gt; WalResult&lt;()&gt;</code>	Explicit error handling	Can test error conditions
Trait Abstractions	<code>trait Storage { fn append(&amp;mut self, data: &amp;[u8]) -&gt; WalResult&lt;u64&gt;; }</code>	Implementation flexibility	Multiple storage backends for testing

**Testing Philosophy:** Each module should be testable in isolation using dependency injection and mock implementations. Integration tests should verify cross-module interactions, while unit tests should focus on individual component logic.

## Common Pitfalls in File Organization

**⚠ Pitfall: Circular Dependencies** Many developers accidentally create circular dependencies between WAL components. For example, the RecoveryManager needs to read log records, but the LogWriter might want to trigger recovery in certain error conditions. This creates a circular dependency that makes testing and reasoning about the system much more difficult.

**Solution:** Use event-driven communication or callback interfaces to break cycles. The LogWriter can accept a callback for error conditions rather than directly depending on the RecoveryManager.

**⚠ Pitfall: Shared Mutable State** Putting shared mutable state (like the current LSN or active transaction list) in a global module is tempting but creates testing and concurrency problems. Tests become interdependent,

and reasoning about thread safety becomes much harder.

**Solution:** Encapsulate state within component instances and pass state explicitly through method parameters when needed. Use dependency injection to provide access to shared resources.

**⚠ Pitfall: Overly Complex Interfaces** Trying to make interfaces too generic or flexible often leads to complex trait hierarchies that are difficult to implement and test. The interface becomes more complex than any individual implementation.

**Solution:** Start with concrete types and simple interfaces. Extract traits only when you actually need multiple implementations, and keep trait methods focused on essential operations.

**⚠ Pitfall: Inadequate Error Context** WAL systems have many possible failure modes (disk full, corruption, I/O errors), but developers often use generic error types that don't provide enough context for debugging. When something fails during recovery, it's crucial to know exactly what operation failed and why.

**Solution:** Create specific error variants for different failure modes, and include relevant context (LSN, transaction ID, file path, etc.) in error messages. This makes debugging much more efficient.

## Implementation Guidance

This section provides concrete implementation recommendations for setting up the WAL architecture in Rust. The guidance focuses on establishing the foundation that will support all four milestones while providing a clear development path.

## Technology Recommendations

Component	Simple Option	Advanced Option	Recommendation
File I/O	<code>std::fs::File</code> with manual <code>fsync</code>	<code>tokio::fs::File</code> with async I/O	Start simple - WAL operations are inherently sequential
Serialization	Manual byte packing with <code>byteorder</code> crate	<code>serde</code> with <code>bincode</code>	Manual packing for learning, but <code>serde</code> for production
Error Handling	<code>std::error::Error</code> with custom enums	<code>anyhow</code> or <code>thiserror</code> crate	<code>thiserror</code> for rich error context
Concurrency	<code>std::sync::Mutex</code> for critical sections	<code>tokio::sync::RwLock</code> for async	Start with std library, optimize later
Testing	Built-in <code>#[test]</code> with manual fixtures	<code>proptest</code> for property-based testing	Both - unit tests for logic, property tests for serialization
Storage Backend	Direct file operations	Abstract Storage trait	Abstract trait from the beginning for testability

## Recommended Dependencies (Cargo.toml)

```
[dependencies]                                     TOML

byteorder = "1.4"                      # Binary serialization helpers

crc32fast = "1.3"                       # CRC32 checksums for corruption detection

thiserror = "1.0"                        # Rich error types

log = "0.4"                            # Logging infrastructure

uuid = { version = "1.0", features = ["v4"] }  # Transaction ID generation

[dev-dependencies]

tempfile = "3.0"                        # Temporary directories for testing

proptest = "1.0"                         # Property-based testing

env_logger = "0.10"                      # Test logging output

criterion = "0.4"                        # Benchmarking
```

## Core Type Definitions

Create the foundational types that all components will use. These must match the naming conventions exactly.

`src/types.rs:`

```
/// Log Sequence Number - globally unique, monotonically increasing identifier

pub type LSN = u64;

/// Transaction identifier - unique within a WAL instance

pub type TransactionId = u64;

/// Database page identifier

pub type PageId = u32;

/// Standard result type for all WAL operations

pub type WalResult<T> = Result<T, WalError>;

/// Common constants used throughout the WAL system

pub const DEFAULT_SEGMENT_SIZE: u64 = 64 * 1024 * 1024; // 64MB

pub const PAGE_SIZE: usize = 8192; // 8KB pages

pub const RECORD_HEADER_SIZE: usize = 32; // Fixed header size

pub const CRC_SIZE: usize = 4; // CRC32 checksum size
```

src/errors.rs:

```

use thiserror::Error;

#[derive(Error, Debug)]

pub enum WalError {
    #[error("I/O error: {0}")]
    Io(#[from] std::io::Error),

    #[error("Corruption detected at LSN {lsn}: {reason}")]
    Corruption { lsn: crate::types::LSN, reason: String },

    #[error("Invalid log record format: {0}")]
    InvalidFormat(String),

    #[error("Transaction {txn_id} not found")]
    TransactionNotFound { txn_id: crate::types::TransactionId },

    #[error("Log segment full, rotation required")]
    SegmentFull,

    #[error("Recovery failed: {reason}")]
    RecoveryFailed { reason: String },
}

```

## Storage Abstraction

Create a storage abstraction that enables testing with in-memory implementations while supporting real file I/O for production use.

**src/storage/mod.rs:**

```
use crate::types::{WalResult, LSN};

use std::path::PathBuf;

/// Abstract storage interface for WAL segments

pub trait Storage: Send + Sync {

    /// Append data to the storage, returning the offset where data was written

    fn append(&mut self, data: &[u8]) -> WalResult<u64>;


    /// Force all buffered data to persistent storage

    fn sync(&mut self) -> WalResult<()>;


    /// Read data from the given offset

    fn read_at(&self, offset: u64, buffer: &mut [u8]) -> WalResult<usize>;


    /// Get the current size of the storage

    fn size(&self) -> WalResult<u64>;


    /// Check if storage needs rotation based on size limits

    fn should_rotate(&self, max_size: u64) -> WalResult<bool>;


}

pub mod file_storage;

pub mod memory_storage;
```

**src/storage/memory\_storage.rs (Complete Implementation):**

```
use super::Storage;

use crate::types::WalResult;

use std::sync::{Arc, RwLock};

/// In-memory storage implementation for testing

#[derive(Debug, Clone)]

pub struct MemoryStorage {

    data: Arc<RwLock<Vec<u8>>,

    sync_count: Arc<RwLock<usize>>, // Track sync calls for testing

}

impl MemoryStorage {

    pub fn new() -> Self {

        Self {

            data: Arc::new(RwLock::new(Vec::new())),

            sync_count: Arc::new(RwLock::new(0)),

        }

    }

    pub fn sync_count(&self) -> usize {

        *self.sync_count.read().unwrap()

    }

    pub fn data(&self) -> Vec<u8> {

        self.data.read().unwrap().clone()

    }

}
```

```
impl Storage for MemoryStorage {

    fn append(&mut self, data: &[u8]) -> WalResult<u64> {

        let mut storage = self.data.write().unwrap();

        let offset = storage.len() as u64;

        storage.extend_from_slice(data);

        Ok(offset)

    }

    fn sync(&mut self) -> WalResult<()> {

        let mut count = self.sync_count.write().unwrap();

        *count += 1;

        Ok(())

    }

    fn read_at(&self, offset: u64, buffer: &mut [u8]) -> WalResult<usize> {

        let storage = self.data.read().unwrap();

        let start = offset as usize;

        let end = std::cmp::min(start + buffer.len(), storage.len());



        if start >= storage.len() {

            return Ok(0);

        }

        let bytes_to_copy = end - start;

        buffer[..bytes_to_copy].copy_from_slice(&storage[start..end]);

        Ok(bytes_to_copy)

    }

}
```

```
fn size(&self) -> WalResult<u64> {
    Ok(self.data.read().unwrap().len() as u64)
}

fn should_rotate(&self, max_size: u64) -> WalResult<bool> {
    Ok(self.size()? >= max_size)
}

}
```

## Component Skeletons

Provide skeleton implementations for the main components with detailed TODO comments that map to the implementation milestones.

**src/log\_writer/mod.rs:**

```
use crate::types::*;

use crate::records::LogRecord;

use crate::storage::Storage;

pub struct LogWriter {

    storage: Box<dyn Storage>,

    buffer: Vec<u8>,

    current_lsn: LSN,

    buffer_size: usize,

}

impl LogWriter {

    pub fn new(storage: Box<dyn Storage>) -> Self {

        Self {

            storage,

            buffer: Vec::new(),

            current_lsn: 1, // LSN 0 is reserved

            buffer_size: 64 * 1024, // 64KB buffer

        }

    }

    /// Write a log record and ensure it reaches persistent storage

    pub fn write_and_force(&mut self, record: &LogRecord) -> WalResult<LSN> {

        // TODO Milestone 2: Implement the core write-and-force operation

        // TODO 1: Assign the next LSN to this record

        // TODO 2: Serialize the record to binary format with CRC

        // TODO 3: Add serialized record to buffer

        // TODO 4: If buffer is full or this is a commit record, flush to storage

    }

}
```

```
// TODO 5: Call storage.sync() to ensure durability

// TODO 6: Return the assigned LSN

// Hint: Commit records must be flushed immediately for durability

todo!("Milestone 2: Implement write_and_force")

}

/// Flush any buffered records to storage without forcing sync

pub fn flush_buffer(&mut self) -> WalResult<()> {

    // TODO Milestone 2: Implement buffer flushing for group commit

    // TODO 1: Check if buffer has any data to flush

    // TODO 2: Write buffer contents to storage using storage.append()

    // TODO 3: Clear the buffer after successful write

    // TODO 4: Handle partial write errors appropriately

    todo!("Milestone 2: Implement flush_buffer")

}

}
```

**src/recovery/mod.rs:**

```
use crate::types::*;

use crate::log_reader::LogReader;

use std::collections::{HashMap, HashSet};

pub struct RecoveryManager {

    log_reader: LogReader,

    // Analysis phase results

    transaction_table: HashMap<TransactionId, TransactionState>,

    dirty_page_table: HashMap<PageId, LSN>,

}

#[derive(Debug, Clone)]

pub struct TransactionState {

    pub status: TransactionStatus,

    pub last_lsn: LSN,

    pub undo_next_lsn: Option<LSN>,

}

#[derive(Debug, Clone, PartialEq)]

pub enum TransactionStatus {

    Active,

    Committed,

    Aborted,

}

impl RecoveryManager {

    pub fn new(log_reader: LogReader) -> Self {

        Self {

            log_reader,
```

```

        transaction_table: HashMap::new(),
        dirty_page_table: HashMap::new(),
    }

}

/// Perform complete ARIES recovery: Analysis, Redo, Undo

pub fn recover(&mut self) -> WalResult<()> {
    // TODO Milestone 3: Implement complete ARIES recovery

    // TODO 1: Find the most recent checkpoint (or start from beginning)

    // TODO 2: Run analysis_pass() to build transaction and dirty page tables

    // TODO 3: Run redo_pass() to replay all committed changes

    // TODO 4: Run undo_pass() to roll back incomplete transactions

    // TODO 5: Write a recovery completion record to the log

    // Hint: Each phase builds on the results of the previous phase

    todo!("Milestone 3: Implement recover")
}

/// Analysis phase: scan log and build recovery data structures

pub fn analysis_pass(&mut self) -> WalResult<()> {
    // TODO Milestone 3: Implement ARIES analysis phase

    // TODO 1: Start scanning from checkpoint (or log beginning)

    // TODO 2: For each log record, update transaction_table appropriately

    // TODO 3: For redo records, update dirty_page_table with page modifications

    // TODO 4: For commit/abort records, mark transactions as completed

    // TODO 5: Continue until end of log is reached

    todo!("Milestone 3: Implement analysis_pass")
}

```

```
}
```

## File Structure Setup Script

Create a setup script that generates the complete directory structure:

**setup.sh:**

BASH

```
#!/bin/bash

# WAL Implementation Project Structure Setup

# Create main directory structure

mkdir -p src/{log_writer,log_reader,recovery,checkpoint,records,storage}

mkdir -p tests/{integration,common}

mkdir -p examples benches

# Create module files with basic structure

cat > src/lib.rs << 'EOF'

///! Write-Ahead Logging (WAL) Implementation

///!

///! This crate provides a complete WAL system with ARIES-style recovery,
///! fuzzy checkpointing, and crash safety guarantees.

pub mod types;

pub mod errors;

pub mod records;

pub mod storage;

pub mod log_writer;

pub mod log_reader;

pub mod recovery;

pub mod checkpoint;

// Re-export main types for convenience

pub use types::*;

pub use errors::*;

pub use records::LogRecord;

pub use log_writer::LogWriter;
```

```
pub use recovery::RecoveryManager;

EOF

# Create mod.rs files for each module

for dir in log_writer log_reader recovery checkpoint records storage; do

    echo "// Module: $dir" > src/$dir/mod.rs

done

echo "WAL project structure created successfully!"

echo "Next steps:"

echo "1. Run 'cargo check' to verify basic setup"

echo "2. Implement types.rs and errors.rs"

echo "3. Start with Milestone 1 (Log Record Format)"
```

## Milestone Checkpoints

After completing the architecture setup, verify these behaviors:

1. **Project Structure Verification:** Run `cargo check` - should compile without errors
2. **Type System Check:** Create a simple test that instantiates the main types
3. **Storage Abstraction Test:** Write a test that uses `MemoryStorage` to verify the interface works
4. **Module Boundaries:** Verify that each module can be imported and basic types are accessible

**Verification Test (tests/integration/architecture\_test.rs):**

```
use wal_implementation::*;

use wal_implementation::storage::memory_storage::MemoryStorage;

#[test]

fn test_basic_architecture_setup() {

    // Verify type aliases work

    let lsn: LSN = 42;

    let txn_id: TransactionId = 100;

    let page_id: PageId = 1;

    // Verify storage abstraction

    let storage = MemoryStorage::new();

    let writer = LogWriter::new(Box::new(storage));

    // This test should compile and pass if architecture is set up correctly

    assert_eq!(lsn, 42);

}
```

Run this test with: `cargo test test_basic_architecture_setup`

Expected output: The test should pass, confirming that the basic architecture compiles and the main types are properly defined.

## Data Model

**Milestone(s):** Milestone 1 (Log Record Format) - this section defines the core data structures and serialization formats that form the foundation for all subsequent WAL operations

## Log Record Format: Binary layout of log records with LSN, transaction ID, and operation data

Think of a **log record** as a medical chart entry in a hospital - each entry has a unique timestamp, identifies which patient (transaction) it belongs to, describes what procedure was performed, and includes before/after vital signs. Just as doctors must record every treatment before performing it so they can track patient history and reverse harmful treatments, our Write-Ahead Logging system records every database change before applying it so we can replay successful changes and undo incomplete ones after a crash.

The **Log Sequence Number (LSN)** serves as our global timestamp - a monotonically increasing 64-bit counter that uniquely identifies each log record and establishes a total ordering of all database operations. Unlike wall-clock time, LSNs never go backwards, never repeat, and provide a precise sequence that recovery can follow to reconstruct database state. Think of LSNs as frame numbers in a movie reel - each frame has a unique number, they're always in order, and you can jump to any specific frame to see exactly what the database looked like at that moment.

Every log record shares a common header structure that enables efficient parsing and validation. The header contains the essential metadata needed to process any record type, while the payload section varies based on the specific operation being logged.

### Log Record Header Structure:

Field	Type	Size (bytes)	Description
magic	u32	4	Magic number (0xDEADBEEF) for corruption detection and format validation
record_type	u8	1	Discriminant identifying record variant (Redo=1, Undo=2, Commit=3, Abort=4, Checkpoint=5)
payload_length	u32	4	Length of variable-sized payload following this header
lsn	LSN (u64)	8	Unique log sequence number for this record
checksum	u32	4	CRC32 checksum covering header + payload for corruption detection

The header design balances efficiency with robustness. The magic number enables quick detection of corrupted or misaligned reads - if we encounter bytes that don't start with our expected magic value, we know we've hit corruption or are reading from the wrong offset. The record type field uses a single byte to efficiently discriminate between different log record variants. The payload length allows variable-sized records while enabling efficient skipping during log scans.

## Decision: Fixed-Size Header with Variable Payload

- **Context:** Log records need different amounts of data (commits need little, data changes need before/after images)
- **Options Considered:** All fixed-size records, all variable-size records, fixed header + variable payload
- **Decision:** Fixed header + variable payload
- **Rationale:** Fixed headers enable efficient parsing and offset calculation, while variable payloads avoid wasting space for small records like commits. Recovery can quickly skip over records by reading the header and jumping by payload\_length bytes.
- **Consequences:** Enables efficient log scanning and space utilization, but requires careful handling of variable-length fields during serialization

The LSN placement in the header ensures every record can be quickly identified during recovery without parsing the full payload. This enables efficient binary search within log segments when looking for specific LSN ranges during checkpoint recovery.

### Endianness and Portability Considerations:

All multi-byte integer fields use little-endian encoding for consistency with x86/x64 architectures where this WAL implementation will primarily run. The serialization format includes explicit byte-order handling to prevent corruption when log files are moved between different architecture systems.

### Checksum Coverage and Validation:

The CRC32 checksum covers both the header (excluding the checksum field itself) and the entire payload. This provides strong corruption detection while remaining computationally efficient. During deserialization, we recompute the checksum and compare it against the stored value - any mismatch indicates corruption and triggers error recovery procedures.

### Record Size Limits and Fragmentation:

Individual log records are limited to 16MB payload size to prevent excessive memory usage during serialization and to ensure reasonable recovery performance. Operations that would exceed this limit (such as large BLOB updates) are automatically fragmented across multiple redo records that reference the same transaction and page but cover different offset ranges.

## Record Types: Different types of log records: redo, undo, checkpoint, and commit records

The WAL system uses five distinct record types, each optimized for its specific role in transaction processing and recovery. Think of these as different types of entries in a detailed project journal - some describe work performed (redo records), others describe how to undo work (undo records), some mark project milestones (commits), and others capture the overall project status at key points (checkpoints).

## RedoRecord: Capturing Forward-Rolling Changes

Redo records contain all information necessary to reapply a committed transaction's changes during crash recovery. They store the **after-image** - the new data that should be written to the database page after the operation completes. Think of redo records as "recipe cards" that tell recovery exactly how to recreate a dish (database change) from scratch.

Field	Type	Description
lsn	LSN	Unique sequence number for this log record
txn_id	TransactionId	Transaction that performed this operation
page_id	PageId	Database page being modified
offset	u32	Byte offset within the page where change begins
after_image	Vec	New data bytes to write at the specified location

The `after_image` field contains the exact bytes that should exist in the database page after this operation completes. During redo recovery, we simply copy these bytes to the specified offset within the target page, regardless of what data currently exists there. This makes redo operations **idempotent** - applying the same redo record multiple times produces identical results.

## UndoRecord: Enabling Transaction Rollback

Undo records contain the **before-image** - the original data that existed before a transaction modified it. These records enable rolling back incomplete transactions during crash recovery or explicit transaction aborts. Think of undo records as "control-Z" commands that restore the previous state of any change.

Field	Type	Description
lsn	LSN	Unique sequence number for this log record
txn_id	TransactionId	Transaction that performed the original operation
page_id	PageId	Database page that was modified
offset	u32	Byte offset within the page where original data should be restored
before_image	Vec	Original data bytes that existed before the change

Undo records are written to the log **before** the corresponding database change is applied. This ensures that if a crash occurs after the log write but before the database update, recovery can safely ignore the incomplete operation. The before-image captures enough information to completely reverse the operation's effects.

## CommitRecord: Marking Transaction Success

Commit records serve as definitive markers that a transaction completed successfully and its changes should be permanently preserved. These records are small but critical - once a commit record is forced to disk with fsync, the transaction's durability is guaranteed even if a crash occurs immediately afterward.

Field	Type	Description
lsn	LSN	Unique sequence number for this commit record
txn_id	TransactionId	Transaction being committed

The absence of additional fields is intentional - commit records need only establish that a specific transaction completed successfully at a specific LSN. All the transaction's data changes are recorded in preceding redo/undo records.

### AbortRecord: Recording Transaction Failures

Abort records mark transactions that were explicitly rolled back or failed during processing. They serve as definitive markers that a transaction's changes should be undone, which is particularly important for distinguishing between transactions that were aborted before a crash versus transactions that were still active when the crash occurred.

Field	Type	Description
lsn	LSN	Unique sequence number for this abort record
txn_id	TransactionId	Transaction being aborted

### CheckpointRecord: Capturing Recovery State

Checkpoint records capture a snapshot of the recovery state at a specific point in time, enabling future recoveries to start from a known-good state rather than scanning the entire log from the beginning. Think of checkpoints as "save games" that let you restart from a recent point rather than replaying the entire game.

Field	Type	Description
lsn	LSN	Unique sequence number for this checkpoint record
active_transactions	Vec	List of transactions that were active when checkpoint was taken
dirty_pages	Vec	List of database pages modified since the last checkpoint

The `active_transactions` list tells recovery which transactions were incomplete when the checkpoint was created. During recovery, any transactions in this list that don't have corresponding commit or abort records after the checkpoint must be rolled back. The `dirty_pages` list identifies which database pages might contain uncommitted changes and need to be processed during redo recovery.

## Decision: Separate Redo and Undo Records

- **Context:** Need to support both crash recovery (redo) and transaction rollback (undo) operations
- **Options Considered:** Combined records with both images, separate redo/undo records, operation-based logging
- **Decision:** Separate redo and undo records
- **Rationale:** Separate records optimize storage (only store needed image), simplify recovery logic (clear separation between redo and undo phases), and enable efficient log scanning (can skip irrelevant record types)
- **Consequences:** Requires writing two records per operation (undo before, redo after), but provides cleaner recovery semantics and better performance during large transaction rollbacks

## Record Type Relationships and Ordering

The different record types follow strict ordering rules within the log:

1. **UndoRecord** → **DatabaseUpdate** → **RedoRecord**: For each database modification, the undo record must be written and forced to disk before applying the change, and the redo record must be written after the change completes
2. **CommitRecord**: Written only after all of a transaction's redo records are safely on disk
3. **CheckpointRecord**: Written periodically during normal operations to bound recovery time

This ordering ensures that recovery always has sufficient information to correctly handle any crash scenario, whether it occurs before, during, or after individual operations.

## Serialization Format: Binary encoding scheme with CRC checksums and variable-length field handling

The serialization format transforms the structured log records into compact binary representations suitable for efficient disk storage and network transmission. Think of serialization as **packing a suitcase** - we need to fit variable-sized items (transaction data, page images) into a standardized format while ensuring nothing gets damaged or lost during transport.

### Binary Encoding Strategy

All log records serialize to a contiguous byte array following a consistent layout pattern. Fixed-size fields like LSNs and transaction IDs are encoded directly as little-endian integers, while variable-length fields like data images use length-prefixed encoding to enable safe deserialization.

### Fixed-Field Encoding Rules:

Type	Size	Encoding
u8	1 byte	Direct binary value
u32	4 bytes	Little-endian 32-bit integer
u64 (LSN, TransactionId)	8 bytes	Little-endian 64-bit integer
Vec<u8>	Variable	4-byte length prefix + data bytes
Vec<TransactionId>	Variable	4-byte count + (count × 8) bytes of transaction IDs
Vec<PageId>	Variable	4-byte count + (count × 4) bytes of page IDs

## Variable-Length Field Handling

Variable-length fields present unique challenges because we must store both the data and enough metadata to parse it correctly during deserialization. The length-prefixed approach provides efficient parsing while maintaining safety against buffer overruns.

For byte vectors ( `Vec<u8>` ), we serialize a 4-byte length field followed by the raw data bytes:

```
[length: u32][data_byte_1][data_byte_2]...[data_byte_N]
```

For typed vectors like `Vec<TransactionId>` , we serialize a 4-byte count followed by the fixed-size elements:

```
[count: u32][txn_id_1: u64][txn_id_2: u64]...[txn_id_N: u64]
```

This encoding enables efficient deserialization - we read the count, allocate appropriately-sized collections, then read the known number of fixed-size elements.

## RedoRecord Serialization Layout

```
[magic: u32][type: u8][length: u32][lsn: u64][checksum: u32]
[txn_id: u64][page_id: u32][offset: u32][image_len: u32][after_image: Vec<u8>]
```

The serialization process:

1. Write the common header with magic number, record type (1 for Redo), and placeholder for payload length and checksum
2. Serialize the transaction ID, page ID, and offset as little-endian integers
3. Write the after-image length as a u32, followed by the raw image bytes
4. Calculate the total payload length and update the header length field
5. Compute CRC32 over the entire header (excluding checksum field) plus payload

6. Update the header checksum field with the computed CRC

### UndoRecord Serialization Layout

```
[magic: u32][type: u8][length: u32][lsn: u64][checksum: u32]  
[txn_id: u64][page_id: u32][offset: u32][image_len: u32][before_image: Vec<u8>]
```

Undo records follow identical serialization logic to redo records, with type field set to 2 and the variable-length field containing the before-image instead of after-image.

### CheckpointRecord Serialization Layout

```
[magic: u32][type: u8][length: u32][lsn: u64][checksum: u32]  
[active_txn_count: u32][txn_id_1: u64]...[txn_id_N: u64]  
[dirty_page_count: u32][page_id_1: u32]...[page_id_M: u32]
```

Checkpoint records require more complex serialization due to their two variable-length arrays. The process:

1. Write common header with type 5 and placeholder fields
2. Write active transaction count, followed by each transaction ID as u64
3. Write dirty page count, followed by each page ID as u32
4. Update header length and checksum fields as with other record types

### Decision: Length-Prefixed Variable Fields

- **Context:** Need to serialize variable-length data (page images, transaction lists) safely
- **Options Considered:** Null-terminated strings, fixed-size padding, length-prefixed encoding, self-describing formats
- **Decision:** Length-prefixed encoding with explicit counts
- **Rationale:** Length prefixes enable efficient parsing without scanning for terminators, prevent buffer overruns, and support binary data containing any byte values. Explicit counts allow pre-allocation during deserialization.
- **Consequences:** Requires 4 extra bytes per variable field, but provides safety and performance benefits that outweigh the storage overhead

### Checksum Calculation and Validation

The CRC32 checksum provides strong corruption detection while remaining computationally efficient for high-throughput logging. The checksum covers all record data except the checksum field itself, ensuring any bit-level corruption will be detected during deserialization.

#### Checksum Calculation Process:

1. Initialize CRC32 calculator with standard polynomial (0xEDB88320)

2. Feed header bytes (magic through LSN) into CRC calculation
3. Feed entire payload bytes into CRC calculation
4. Store final CRC32 value in the header checksum field

#### **Deserialization and Validation Process:**

1. Read the fixed-size header and extract payload length
2. Read the payload bytes based on the length field
3. Recompute CRC32 over header (excluding checksum) and payload
4. Compare computed checksum against stored checksum - mismatch indicates corruption
5. Parse payload fields according to record type using length-prefixed decoding

#### **Corruption Detection and Recovery**

The serialization format includes multiple layers of corruption detection:

**Magic Number Validation:** Each record must begin with the expected magic number (0xDEADBEEF). Invalid magic numbers indicate either corruption or incorrect read positioning within the log file.

**CRC Checksum Validation:** The CRC32 checksum detects bit-level corruption within record contents. CRC32 can detect all single-bit errors and most multi-bit error patterns.

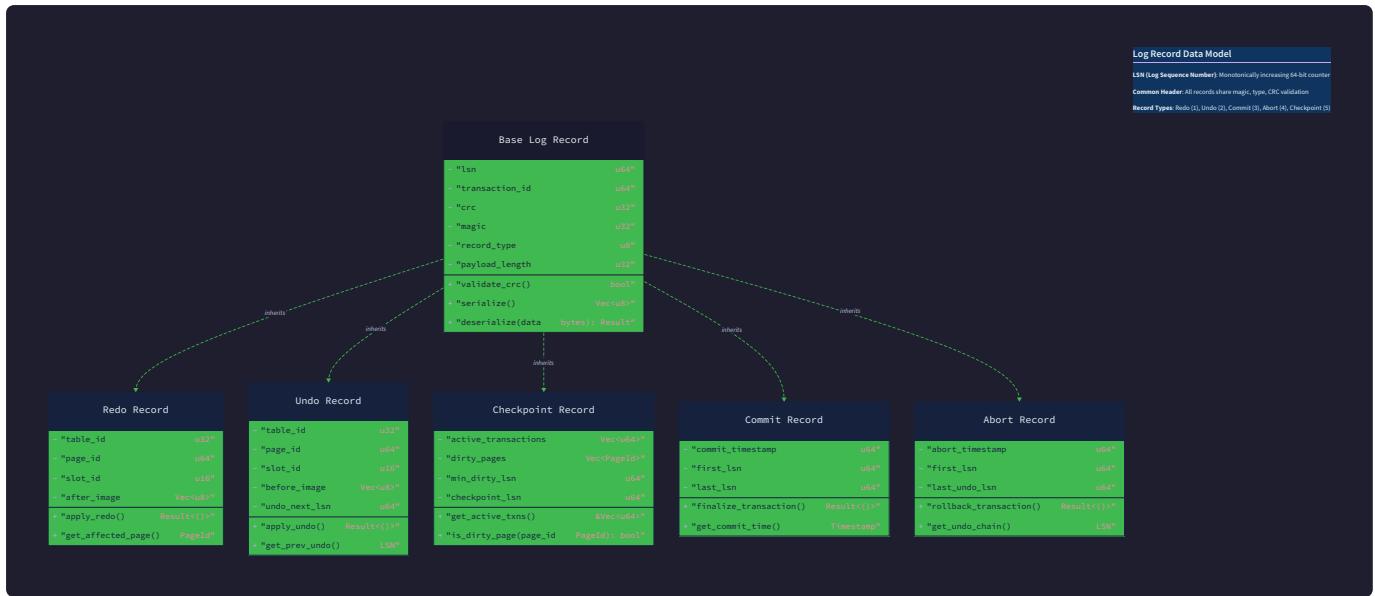
**Length Field Consistency:** The payload length field must match the actual serialized payload size. Inconsistencies indicate either header corruption or truncated records.

**Record Type Validation:** The record type field must contain a valid discriminant value (1-5). Invalid types suggest corruption or version mismatches.

**⚠ Pitfall: Partial Record Writes** A common mistake is failing to handle scenarios where a crash occurs during record serialization, leaving incomplete records at the end of the log. Always validate that you can read a complete record (header + full payload) before attempting to deserialize it. Implement a "safe read" function that returns an error if insufficient bytes remain in the log file rather than reading garbage data.

**⚠ Pitfall: Endianness Assumptions** Don't assume the serialization and deserialization environments use the same byte order. Always use explicit little-endian encoding for portability. In Rust, use the `byteorder` crate's `LittleEndian` types rather than transmuting native integers.

**⚠ Pitfall: Variable-Length Field Buffer Overruns** When deserializing variable-length fields, always validate that the declared length doesn't exceed the remaining payload bytes before attempting to read the data. Malformed or corrupted length fields can cause reads beyond the record boundary, leading to crashes or garbage data.



## Serialization Performance Considerations

The binary serialization format is optimized for write throughput since WAL performance directly impacts transaction processing latency. Several design decisions optimize for the common case of sequential log writing:

**Minimized Memory Allocations:** Fixed-size fields serialize directly to the output buffer without intermediate allocations. Variable-length fields are serialized with a single allocation for the length prefix plus data.

**Cache-Friendly Layout:** Fields are arranged to minimize padding bytes and enable efficient CPU cache utilization during serialization and deserialization.

**Vectorized I/O Support:** The contiguous binary layout enables efficient use of vectorized write operations (writev) to minimize system calls during log writing.

## Implementation Guidance

### Technology Recommendations:

Component	Simple Option	Advanced Option
Serialization	Manual byte manipulation with <code>Vec&lt;u8&gt;</code>	<code>serde</code> with custom binary format
CRC Calculation	<code>crc32fast</code> crate	<code>crc</code> crate with hardware acceleration
Byte Order Handling	<code>byteorder</code> crate	<code>zerocopy</code> for zero-copy parsing
Error Handling	<code>thiserror</code> for error types	<code>anyhow</code> for error context chaining

### Recommended File Structure:

```
src/
  wal/
    mod.rs           ← public API and re-exports
    record.rs        ← log record types and serialization (this milestone)
    writer.rs        ← log writer component (milestone 2)
    recovery.rs      ← crash recovery logic (milestone 3)
    checkpoint.rs    ← checkpointing system (milestone 4)
  storage/
    mod.rs           ← storage abstraction
    memory.rs        ← in-memory storage for testing
  error.rs          ← error types
  lib.rs            ← crate root
```

### Core Data Structure Definitions (Complete Implementation):

```
use std::collections::HashMap;

use std::sync::{Arc, RwLock};

use std::path::PathBuf;

use std::fs::File;

// Type aliases matching naming conventions

pub type LSN = u64;

pub type TransactionId = u64;

pub type PageId = u32;

pub type WalResult<T> = Result<T, WalError>;

// Error types for comprehensive error handling

#[derive(Debug, Clone)]

pub enum WalError {

    IoError(String),

    CorruptionError(String),

    SerializationError(String),

    InvalidRecord(String),

    InsufficientData,

}

impl std::fmt::Display for WalError {

    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {

        match self {

            WalError::IoError(msg) => write!(f, "I/O error: {}", msg),

            WalError::CorruptionError(msg) => write!(f, "Corruption detected: {}", msg),

            WalError::SerializationError(msg) => write!(f, "Serialization error: {}", msg),

            WalError::InvalidRecord(msg) => write!(f, "Invalid record: {}", msg),

        }
    }
}
```

```
        WalError::InsufficientData => write!(f, "Insufficient data to read complete
record"),
    }
}

}

impl std::error::Error for WalError {}

// Transaction state tracking for recovery

#[derive(Debug, Clone, PartialEq)]

pub enum TransactionStatus {

    Active,
    Committed,
    Aborted,
}

#[derive(Debug, Clone)]

pub struct TransactionState {

    pub status: TransactionStatus,
    pub last_lsn: LSN,
    pub undo_next_lsn: Option<LSN>,
}

// Database page representation for testing

#[derive(Debug, Clone)]

pub struct DatabasePage {

    pub page_id: PageId,
    pub data: Vec<u8>,
    pub lsn: u64,
}
```

```
}

// Mock database for testing WAL operations

#[derive(Debug)]

pub struct MockDatabase {

    pub pages: Arc<RwLock<HashMap<PageId, DatabasePage>>>,
    pub page_size: usize,
}

impl MockDatabase {

    pub fn new(page_size: usize) -> Self {
        Self {
            pages: Arc::new(RwLock::new(HashMap::new())),
            page_size,
        }
    }

    pub fn get_page(&self, page_id: PageId) -> Option<DatabasePage> {
        self.pages.read().unwrap().get(&page_id).cloned()
    }

    pub fn update_page(&self, page: DatabasePage) {
        self.pages.write().unwrap().insert(page.page_id, page);
    }

    pub fn write_data(&self, page_id: PageId, offset: usize, data: &[u8], lsn: u64) {
        let mut pages = self.pages.write().unwrap();
        let page = pages.entry(page_id).or_insert_with(|| DatabasePage {
            page_id,
            offset,
            data,
            lsn,
        });
        page.data = data;
        page.lsn = lsn;
    }
}
```

```
        data: vec![0; self.page_size],  
  
        lsn: 0,  
    });  
  
  
    let end_offset = offset + data.len();  
  
    if end_offset <= page.data.len() {  
  
        page.data[offset..end_offset].copy_from_slice(data);  
  
        page.lsn = lsn;  
    }  
  
}  
  
}  
  
// Log segment management  
  
#[derive(Debug)]  
  
pub struct LogSegment {  
  
    pub file: File,  
  
    pub path: PathBuf,  
  
    pub current_size: u64,  
  
    pub max_size: u64,  
}  
  
impl LogSegment {  
  
    pub fn append(&mut self, data: &[u8]) -> WalResult<u64> {  
  
        // TODO: Implement append operation  
  
        // TODO: Update current_size  
  
        // TODO: Return offset where data was written  
  
        todo!("Implement append - write data to file and return offset")  
    }  
}
```

```
pub fn force_sync(&mut self) -> WalResult<()> {
    // TODO: Call fsync on the file
    // TODO: Handle sync errors appropriately
    todo!("Implement fsync for durability")
}

pub fn is_full(&self) -> bool {
    // TODO: Check if current_size >= max_size
    todo!("Check if segment needs rotation")
}

// Constants matching naming conventions

pub const DEFAULT_SEGMENT_SIZE: u64 = 64 * 1024 * 1024; // 64MB

pub const PAGE_SIZE: usize = 4096; // 4KB pages

pub const RECORD_HEADER_SIZE: usize = 21; // magic(4) + type(1) + length(4) + lsn(8) +
checksum(4)

pub const CRC_SIZE: usize = 4;
```

### Log Record Implementation Skeleton:

```
// Log record types - core learning implementation

#[derive(Debug, Clone, PartialEq)]

pub enum LogRecord {

    Redo(RedoRecord),
    Undo(UndoRecord),
    Commit(CommitRecord),
    Abort(AbortRecord),
    Checkpoint(CheckpointRecord),
}

#[derive(Debug, Clone, PartialEq)]

pub struct RedoRecord {

    pub lsn: LSN,
    pub txn_id: TransactionId,
    pub page_id: PageId,
    pub offset: u32,
    pub after_image: Vec<u8>,
}

#[derive(Debug, Clone, PartialEq)]

pub struct UndoRecord {

    pub lsn: LSN,
    pub txn_id: TransactionId,
    pub page_id: PageId,
    pub offset: u32,
    pub before_image: Vec<u8>,
}
```

```
#[derive(Debug, Clone, PartialEq)]  
  
pub struct CommitRecord {  
  
    pub lsn: LSN,  
  
    pub txn_id: TransactionId,  
  
}  
  
#[derive(Debug, Clone, PartialEq)]  
  
pub struct AbortRecord {  
  
    pub lsn: LSN,  
  
    pub txn_id: TransactionId,  
  
}  
  
#[derive(Debug, Clone, PartialEq)]  
  
pub struct CheckpointRecord {  
  
    pub lsn: LSN,  
  
    pub active_transactions: Vec<TransactionId>,  
  
    pub dirty_pages: Vec<PageId>,  
  
}  
  
impl LogRecord {  
  
    /// Extract the LSN from any log record type  
  
    pub fn lsn(&self) -> LSN {  
  
        // TODO: Match on self and return the lsn field from each variant  
  
        // TODO: Handle all LogRecord variants (Redo, Undo, Commit, Abort, Checkpoint)  
  
        todo!("Extract LSN from log record variant")  
  
    }  
  
    /// Extract transaction ID where applicable (None for checkpoints)  
  
    pub fn txn_id(&self) -> Option<TransactionId> {
```

```
// TODO: Match on self and return Some(txn_id) for transaction records

// TODO: Return None for checkpoint records (they don't belong to a single
transaction)

todo!("Extract transaction ID from applicable record types")

}

/// Serialize log record to binary format with CRC checksum

pub fn serialize(&self) -> Vec<u8> {

    // TODO 1: Create output buffer starting with magic number (0xDEADBEEF)

    // TODO 2: Write record type discriminant (1=Redo, 2=Undo, 3=Commit, 4=Abort,
5=Checkpoint)

    // TODO 3: Reserve space for payload length (fill in later)

    // TODO 4: Write LSN as little-endian u64

    // TODO 5: Reserve space for CRC32 checksum (calculate later)

    // TODO 6: Serialize record-specific payload based on type

    // TODO 7: Calculate total payload length and update length field

    // TODO 8: Calculate CRC32 over header (excluding checksum) + payload

    // TODO 9: Update checksum field with computed CRC

    // TODO 10: Return complete serialized bytes

todo!("Implement binary serialization with checksum")

}

/// Deserialize binary data to log record with validation

pub fn deserialize(data: &[u8]) -> WalResult<LogRecord> {

    // TODO 1: Check minimum length for complete header

    // TODO 2: Validate magic number at start of data

    // TODO 3: Extract record type and validate it's in range 1-5

    // TODO 4: Extract payload length and validate sufficient data remains

    // TODO 5: Extract LSN from header
```

```
// TODO 6: Extract stored checksum from header

// TODO 7: Recompute CRC32 over header (excluding checksum) + payload

// TODO 8: Validate computed checksum matches stored checksum

// TODO 9: Parse payload based on record type discriminant

// TODO 10: Return appropriate LogRecord variant

todo!("Implement binary deserialization with validation")

}

}

// Helper functions for serialization

fn serialize_redo_payload(record: &RedoRecord) -> Vec<u8> {

    // TODO 1: Create payload buffer

    // TODO 2: Write txn_id as little-endian u64

    // TODO 3: Write page_id as little-endian u32

    // TODO 4: Write offset as little-endian u32

    // TODO 5: Write after_image length as little-endian u32

    // TODO 6: Append after_image bytes

    // TODO 7: Return completed payload

    todo!("Serialize redo record payload")

}

fn serialize_checkpoint_payload(record: &CheckpointRecord) -> Vec<u8> {

    // TODO 1: Create payload buffer

    // TODO 2: Write active_transactions count as little-endian u32

    // TODO 3: Write each transaction ID as little-endian u64

    // TODO 4: Write dirty_pages count as little-endian u32

    // TODO 5: Write each page ID as little-endian u32

    // TODO 6: Return completed payload
```

```

    todo!("Serialize checkpoint record payload with variable-length arrays")

}

fn deserialize_redo_payload(lsn: LSN, payload: &[u8]) -> WalResult<RedoRecord> {

    // TODO 1: Check payload has minimum required bytes (u64 + u32 + u32 + u32)

    // TODO 2: Parse txn_id from bytes 0-8 as little-endian u64

    // TODO 3: Parse page_id from bytes 8-12 as little-endian u32

    // TODO 4: Parse offset from bytes 12-16 as little-endian u32

    // TODO 5: Parse image_length from bytes 16-20 as little-endian u32

    // TODO 6: Validate image_length matches remaining payload bytes

    // TODO 7: Extract after_image bytes from remaining payload

    // TODO 8: Return RedoRecord with parsed fields

    todo!("Parse redo record from binary payload")

}

fn deserialize_checkpoint_payload(lsn: LSN, payload: &[u8]) -> WalResult<CheckpointRecord> {

    // TODO 1: Check payload has minimum bytes for transaction count

    // TODO 2: Parse active_transactions count from first 4 bytes

    // TODO 3: Validate sufficient bytes remain for all transaction IDs

    // TODO 4: Parse each transaction ID as little-endian u64

    // TODO 5: Parse dirty_pages count after transaction IDs

    // TODO 6: Validate sufficient bytes remain for all page IDs

    // TODO 7: Parse each page ID as little-endian u32

    // TODO 8: Return CheckpointRecord with parsed arrays

    todo!("Parse checkpoint record with variable-length arrays")

}

```

## CRC32 Checksum Helper:

RUST

```
use crc32fast::Hasher;

/// Calculate CRC32 checksum for corruption detection

pub fn calculate_crc32(data: &[u8]) -> u32 {

    let mut hasher = Hasher::new();

    hasher.update(data);

    hasher.finalize()

}

/// Validate record integrity using CRC32 checksum

pub fn validate_checksum(data: &[u8], expected_checksum: u32) -> bool {

    let computed = calculate_crc32(data);

    computed == expected_checksum

}
```

### Milestone Checkpoint:

After implementing the log record data model, you should be able to:

#### 1. Create and serialize different record types:

```
cargo test record_serialization
```

BASH

Expected: All record types serialize to binary format with correct headers and checksums

#### 2. Round-trip serialization test:

```
cargo test record_roundtrip
```

BASH

Expected: Records serialize and deserialize back to identical structures

#### 3. Corruption detection test:

```
cargo test corruption_detection
```

BASH

Expected: Modified binary data fails checksum validation

#### 4. Manual verification:

- Create a redo record with test data
- Serialize it and inspect the binary output (should start with 0xDEADBEEF magic)
- Deserialize it back and verify all fields match
- Modify one byte in the serialized data and confirm deserialization fails

### Debugging Tips:

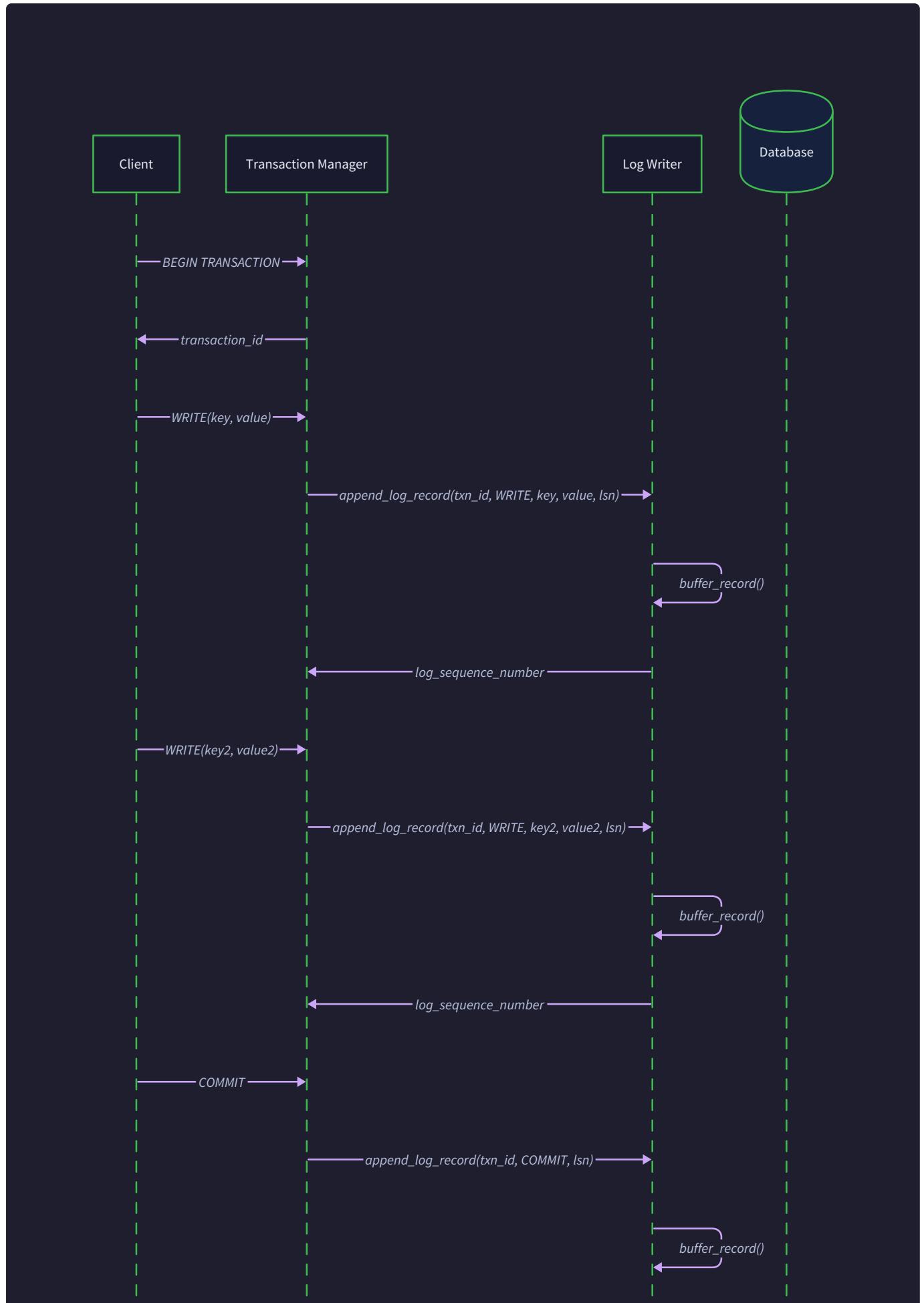
Symptom	Likely Cause	How to Diagnose	Fix
"Invalid magic number" error	Wrong read offset or corruption	Check if reading from correct file position	Implement log scanning to find next valid record
Checksum validation fails	Data corruption or wrong CRC calculation	Compare computed vs stored checksum values	Verify CRC covers correct byte ranges
"Insufficient data" during parsing	Truncated record or wrong length field	Check payload length against remaining bytes	Add length validation before parsing payload
Deserialization panic	Buffer overrun from malformed length	Add bounds checking before all byte reads	Validate all length fields before using them

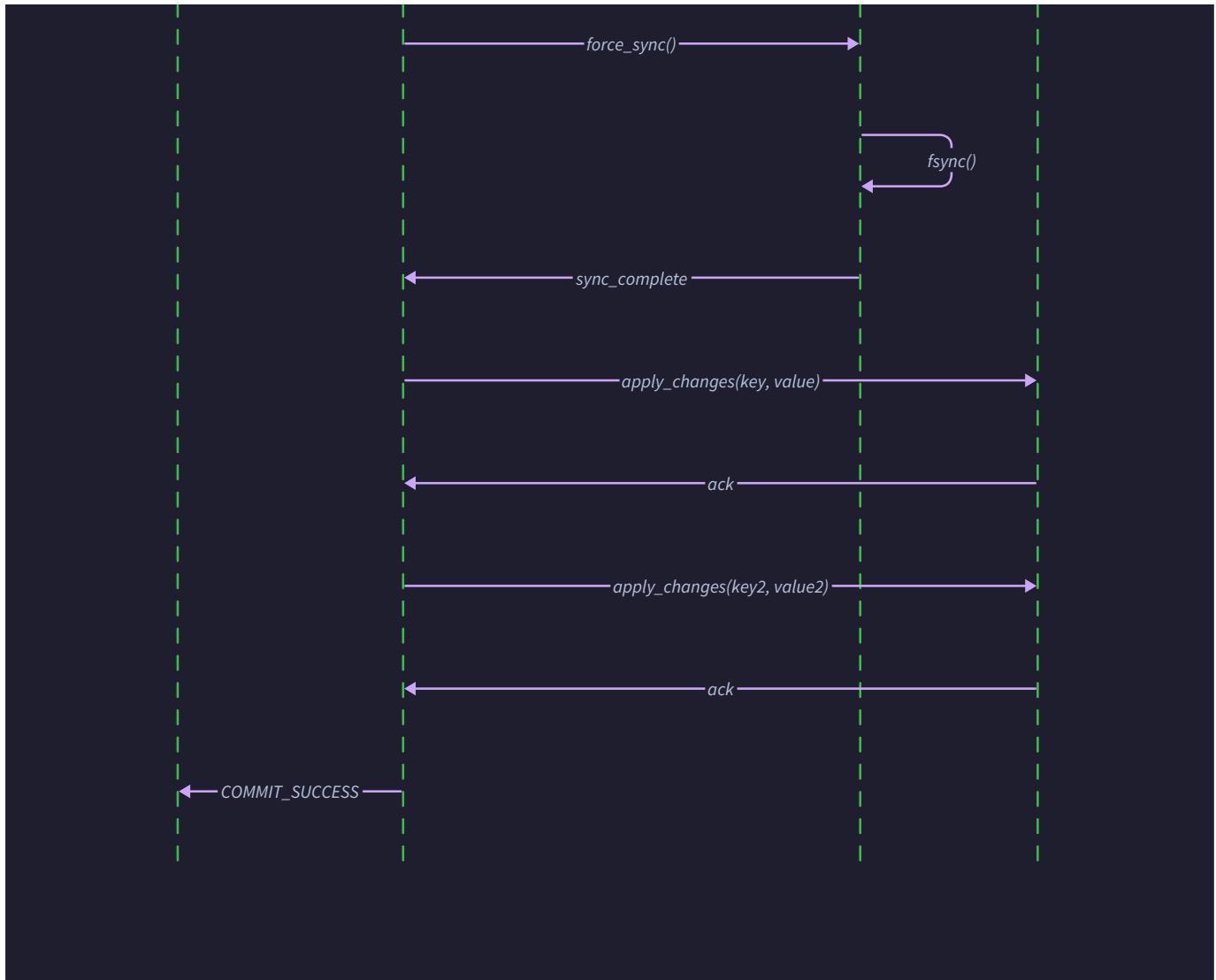
## Log Writer Component

**Milestone(s):** Milestone 2 (Log Writer) - this section implements the sequential, durable writing of log records to disk with atomic append semantics, fsync guarantees, buffer management, and log rotation

Think of the `LogWriter` as the postal service for your database's critical messages. Just as a postal service must guarantee that important letters are delivered in order, never lost, and with proof of delivery, the `LogWriter` ensures that every database change is recorded sequentially, survives crashes, and can be replayed exactly as it occurred. The key insight is that databases achieve durability not by immediately updating data files (which would be slow and complex), but by first writing a complete record of what they intend to do to a simple, append-only log file.

The `LogWriter` component sits at the heart of the Write-Ahead Logging system, serving as the authoritative recorder of all database state changes. Unlike the complex, random-access nature of database pages, the log writer operates on a beautifully simple principle: append new records to the end of a file and force them to disk before acknowledging success. This simplicity is what makes WAL both reliable and performant - sequential writes are the fastest type of disk I/O, and the append-only nature eliminates complex coordination between concurrent writers.





## Append-Only Semantics

The foundation of reliable write-ahead logging rests on **append-only semantics** - the principle that log records are always written to the end of the log file and never modified in place. This design choice creates several powerful guarantees that make crash recovery both possible and efficient.

Consider the alternative approach where log records could be written to arbitrary positions in the file. This would require complex coordination to prevent two writers from overwriting each other's data, sophisticated locking mechanisms to ensure consistency, and elaborate bookkeeping to track which regions of the file contain valid data. More critically, it would make crash recovery nearly impossible because after a crash, the system would have no reliable way to determine which writes completed successfully and in what order.

Append-only semantics eliminates these complexities through a simple invariant: **the log file grows monotonically, and every byte before the current end-of-file represents a committed, immutable record**. This creates a natural ordering where earlier records appear at lower file offsets and later records appear at higher offsets, directly corresponding to the temporal order in which operations occurred.

## Decision: Append-Only Log Structure

- **Context:** Need to ensure that log records are written atomically and can be reliably read during recovery, while supporting concurrent access from multiple transactions
- **Options Considered:**
  1. Random-access log with position-based addressing
  2. Append-only sequential log with monotonic growth
  3. Circular buffer with wraparound and garbage collection
- **Decision:** Implement append-only sequential log structure
- **Rationale:** Sequential writes provide optimal disk performance, atomic append operations are naturally supported by filesystem semantics, and recovery can simply scan from a known checkpoint to the end of file without complex position tracking
- **Consequences:** Simplifies crash recovery logic and maximizes write throughput, but requires periodic log rotation to prevent unbounded growth and cannot reuse disk space from old records until rotation occurs

Approach	Write Performance	Recovery Complexity	Crash Safety	Space Efficiency
Random Access	Poor (seeks required)	High (position tracking)	Complex	Good (reuse space)
<b>Append-Only</b>	<b>Excellent (sequential)</b>	<b>Low (scan forward)</b>	<b>Simple</b>	<b>Fair (needs rotation)</b>
Circular Buffer	Good (sequential)	Medium (wrap handling)	Medium	Excellent

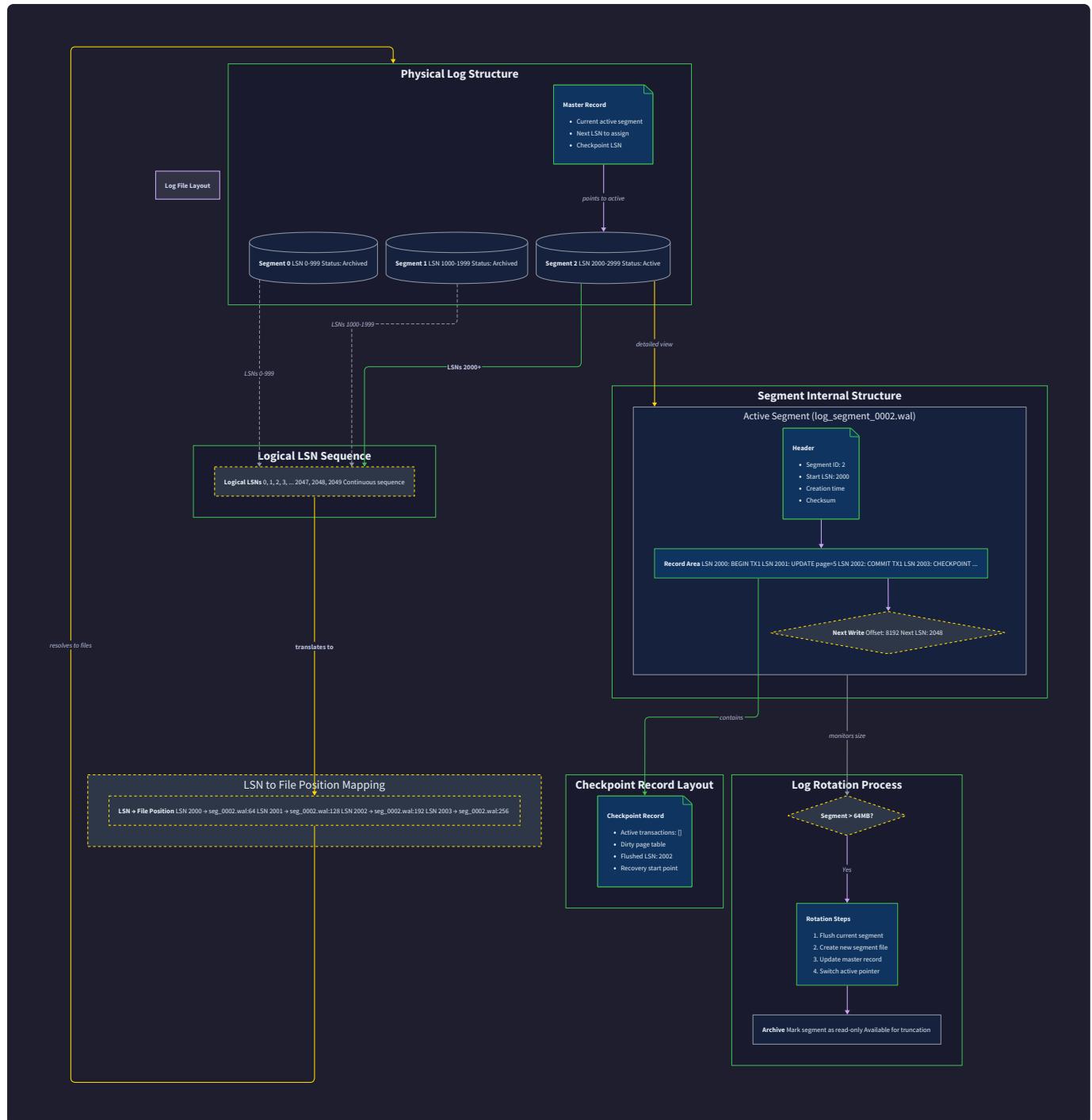
The atomic nature of append operations provides crucial crash safety guarantees. When the `LogWriter` appends a record to the log file, it performs a single `write()` system call that either completes entirely or fails without partial effects. This is because the filesystem's internal write path treats appends to regular files as atomic units up to the filesystem's block size (typically 4KB or larger). Since individual log records are much smaller than filesystem blocks, each append operation is naturally atomic from the perspective of crash recovery.

### Atomicity Mechanism Details:

1. **File Position Management:** The `LogWriter` maintains the current append position as the file's end-of-file offset. Each new record is written starting at this position, and the position advances by the record's total size only after the write completes successfully.

2. **Write Ordering:** Multiple transactions may generate log records concurrently, but the `LogWriter` serializes all append operations through a single write path. This ensures that records appear in the log in the exact order they were committed, preserving the temporal relationships necessary for correct recovery.
3. **Partial Write Prevention:** By sizing log records to be smaller than filesystem block boundaries and using single `write()` calls, the system avoids partial write scenarios where only part of a record reaches disk before a crash.
4. **End-of-File Detection:** During recovery, the system can detect the valid end of the log by scanning forward and validating record checksums. Any incomplete record at the end indicates the exact point where the last successful write occurred.

The append-only design also enables efficient **group commit optimization** where multiple transactions can batch their log records into a single sequential write operation, amortizing the cost of disk I/O across multiple transactions while maintaining the individual ordering of their operations.



Log Writer State	Field Name	Type	Description
Current Position	<code>current_offset</code>	<code>u64</code>	Byte offset where next record will be appended
Active Segment	<code>current_segment</code>	<code>LogSegment</code>	Currently open log file for writing
Write Lock	<code>write_mutex</code>	<code>Mutex&lt;()&gt;</code>	Serializes append operations across threads
Buffer State	<code>pending_records</code>	<code>Vec&lt;Vec&lt;u8&gt;&gt;</code>	Records awaiting batch write to disk
Sync State	<code>last_sync_offset</code>	<code>u64</code>	File position of last successful <code>fsync</code>

## Durability Guarantees

Durability in write-ahead logging means that once a transaction receives a commit acknowledgment, its changes are guaranteed to survive any subsequent crash, power failure, or system restart. The `LogWriter` achieves this through carefully orchestrated **`fsync` operations** that force buffered data from the operating system's page cache to persistent storage before reporting success to the transaction manager.

The challenge lies in understanding that modern storage systems involve multiple layers of caching and buffering. When application code calls `write()`, the data typically moves from user space to kernel buffers, but may not immediately reach the physical disk. The storage device itself often contains additional write caches that buffer data for performance optimization. Only an explicit `fsync()` system call forces the entire pipeline to flush, ensuring data reaches non-volatile storage.

Consider a transaction T1 that updates customer account balances. The transaction manager generates redo records capturing the before and after values, passes them to the `LogWriter` for persistence, and waits for confirmation before responding to the client. If the `LogWriter` returns success without calling `fsync()`, a crash occurring milliseconds later could lose T1's log records even though the client believes the transaction committed successfully. This violates the durability guarantee and could result in permanent data loss.

## Decision: Force-Write Before Commit Acknowledgment

- **Context:** Need to guarantee that committed transactions survive crashes while balancing performance with safety requirements
- **Options Considered:**
  1. Immediate fsync after every log record write
  2. Periodic background fsync with batching
  3. Fsync only before commit acknowledgment
- **Decision:** Implement fsync before commit acknowledgment with optional group commit batching
- **Rationale:** This approach provides strong durability guarantees while allowing multiple transactions to amortize fsync costs through group commit optimization
- **Consequences:** Slight increase in commit latency but ensures ACID compliance and enables performance optimization through intelligent batching

## Force-Write Timing Protocol:

1. **Record Generation Phase:** Transaction generates log records (`RedoRecord`, `UndoRecord`, etc.) and passes them to `LogWriter` via `write_and_force()` method.
2. **Buffer Accumulation Phase:** `LogWriter` appends records to its internal buffer, potentially batching multiple records from concurrent transactions.
3. **Atomic Write Phase:** When commit is requested, `LogWriter` performs a single `write()` system call to append all buffered records to the log file.
4. **Force Sync Phase:** `LogWriter` immediately calls `fsync()` on the log file descriptor, blocking until the storage subsystem confirms data persistence.
5. **Acknowledgment Phase:** Only after `fsync()` returns successfully does `LogWriter` return control to the transaction manager, which then acknowledges the commit to the client.

Durability Method	Write Latency	Crash Safety	Implementation Complexity
No fsync	~0.1ms	None	Low
Background fsync	~0.2ms	Partial (window of loss)	Medium
<b>Immediate fsync</b>	<b>~5-10ms</b>	<b>Complete</b>	<b>Low</b>
Group commit	~5-10ms	Complete	High

## Group Commit Optimization:

Group commit represents a sophisticated optimization that maintains full durability guarantees while dramatically improving throughput under high concurrency. The insight is that if multiple transactions are ready

to commit simultaneously, their log records can be written and fsync'd together, sharing the cost of the expensive disk synchronization operation.

The `LogWriter` implements group commit through a **commit batching window**. When the first transaction requests a force-write, the `LogWriter` briefly delays (typically 1-2 milliseconds) to collect additional commit requests from concurrent transactions. All accumulated records are then written and fsync'd together, and all waiting transactions receive their acknowledgments simultaneously.

This optimization can improve commit throughput from hundreds of transactions per second (limited by fsync latency) to thousands of transactions per second, while maintaining the same strong durability guarantees.

The trade-off is a small increase in average commit latency in exchange for dramatically higher overall system throughput.

Force-Write Operations	Method	Parameters	Returns	Description
Single Record Write	<code>write_and_force</code>	<code>record: &amp;LogRecord</code>	<code>WalResult&lt;LSN&gt;</code>	Write single record with immediate fsync
Batch Write	<code>write_batch_and_force</code>	<code>records: &amp;[LogRecord]</code>	<code>WalResult&lt;Vec&lt;LSN&gt;&gt;</code>	Write multiple records with single fsync
Force Sync Only	<code>force_sync</code>	<code>()</code>	<code>WalResult&lt;()&gt;</code>	Fsync without writing new records
Group Commit	<code>group_commit</code>	<code>timeout: Duration</code>	<code>WalResult&lt;Vec&lt;LSN&gt;&gt;</code>	Batch commits within time window

## Buffer Management

Efficient buffer management sits at the intersection of performance and correctness in the `LogWriter` component. The fundamental challenge is balancing the competing demands of write throughput (which benefits from large batches) and commit latency (which suffers from excessive buffering delays). Think of the buffer as a loading dock where packages (log records) accumulate before being shipped (written to disk) - too small and you make frequent expensive trips, too large and individual packages wait too long for delivery.

The `LogWriter` employs a **multi-level buffering strategy** that operates at three distinct layers: record assembly buffers that construct individual log records, batch accumulation buffers that collect multiple records

for group writing, and kernel page cache integration that optimizes the handoff to the operating system's I/O subsystem.

### Record Assembly Layer:

Individual log records must be serialized from their structured format ( `RedoRecord` , `UndoRecord` , etc.) into the binary format written to disk. Rather than allocating new memory for each record serialization, the `LogWriter` maintains a pool of reusable byte buffers sized to accommodate the largest expected record plus checksums and padding.

The assembly process works by selecting an appropriately-sized buffer from the pool, serializing the record's fields in the defined binary format, calculating and appending the CRC32 checksum, and padding to the next 8-byte boundary for alignment. This buffer then becomes part of the batch accumulation phase.

Buffer Pool Configuration	Parameter	Default Value	Description
Small Record Buffer	<code>SMALL_RECORD_SIZE</code>	256 bytes	For commit/abort records
Medium Record Buffer	<code>MEDIUM_RECORD_SIZE</code>	1KB	For typical redo/undo records
Large Record Buffer	<code>LARGE_RECORD_SIZE</code>	4KB	For checkpoint records
Pool Size Per Category	<code>BUFFERS_PER_SIZE</code>	32	Number of buffers maintained
Maximum Record Size	<code>MAX_RECORD_SIZE</code>	16KB	Hard limit for individual records

### Batch Accumulation Strategy:

The batch accumulation buffer serves as the staging area where multiple serialized log records wait before being written to disk as a single I/O operation. This buffer operates on a **fill-and-flush** principle: records accumulate until either the buffer reaches its capacity threshold, a timeout expires, or a force-write operation is requested.

The key insight is that optimal batch size depends on the storage device characteristics. Traditional hard drives benefit from larger batches (8-32KB) that maximize sequential write efficiency, while modern SSDs perform well with smaller batches (4-8KB) that reduce latency without significantly impacting throughput. The `LogWriter` configuration allows tuning these parameters based on the deployment environment.

## Decision: Adaptive Batch Size with Force-Flush Override

- **Context:** Need to balance write throughput (favoring large batches) with commit latency (favoring immediate writes) under varying load patterns
- **Options Considered:**
  1. Fixed small batches for consistent low latency
  2. Fixed large batches for maximum throughput
  3. Adaptive batching based on current load
  4. Force-flush override with configurable batching
- **Decision:** Implement configurable batching with force-flush override for commits
- **Rationale:** Allows normal operations to benefit from batching while ensuring commit operations can force immediate writes when durability is required
- **Consequences:** Provides good performance under both high and low load conditions, but requires careful tuning of batch thresholds

## Batch Accumulation State Machine:

Current State	Event	Next State	Action Taken
Empty	Record Added	Accumulating	Start batch timer, add record to buffer
Accumulating	Record Added (buffer not full)	Accumulating	Add record to buffer, continue timer
Accumulating	Record Added (buffer full)	Flushing	Flush buffer to disk, reset timer
Accumulating	Timer Expired	Flushing	Flush buffer to disk, reset timer
Accumulating	Force Write Requested	Flushing	Immediate flush, notify waiters
Flushing	Flush Complete	Empty	Return to waiting state
Any State	Force Sync Requested	Flushing	Call fsync(), block until complete

## Memory Management and Back-Pressure:

Under high write loads, the buffer management system must prevent unbounded memory growth while maintaining system responsiveness. The `Logwriter` implements **back-pressure mechanisms** that gracefully slow down transaction generation when buffer capacity is exhausted, rather than allowing memory usage to grow without bounds.

When the batch accumulation buffer approaches its maximum capacity, new write requests begin to block until space becomes available. This creates natural flow control that prevents fast transaction generators from overwhelming the I/O subsystem. The blocking is implemented using condition variables that wake up waiting writers as soon as buffer space is freed by successful disk writes.

Additionally, the buffer pool monitors overall memory usage and can trigger emergency flush operations when total buffer memory exceeds configured thresholds. This ensures that even under extreme load conditions, the system maintains bounded resource usage and continues making forward progress.

Buffer Management Operations	Method	Parameters	Returns	Description
Add Record to Buffer	<code>buffer_record</code>	<code>data: &amp; [u8]</code>	<code>WalResult&lt;()&gt;</code>	Add serialized record to batch buffer
Flush Current Batch	<code>flush_buffer</code>	<code>()</code>	<code>WalResult&lt;u64&gt;</code>	Write accumulated records to disk
Force Buffer Sync	<code>sync_buffer</code>	<code>()</code>	<code>WalResult&lt;()&gt;</code>	Flush buffer and fsync file
Check Buffer Space	<code>buffer_available_space</code>	<code>()</code>	<code>usize</code>	Bytes available in current buffer
Reset Buffer Pool	<code>reset_buffers</code>	<code>()</code>	<code>()</code>	Clear all buffers and reset counters

## Log Rotation

Log rotation addresses the fundamental problem that append-only logs grow indefinitely over time, eventually consuming all available disk space if left unchecked. Think of log rotation like closing one volume of an encyclopedia and starting a fresh volume when the current one becomes too large to handle efficiently. Each log segment represents a complete, self-contained portion of the transaction history that can be independently managed, archived, or deleted.

The challenge in log rotation lies in maintaining the logical continuity of the transaction log across multiple physical files while ensuring that the rotation process itself is crash-safe and doesn't introduce gaps or inconsistencies in the log sequence. The system must coordinate between ongoing write operations, checkpoint processes, and recovery procedures to ensure that log segments transition seamlessly.

### Segment-Based Architecture:

The `Logwriter` organizes the logical transaction log as a sequence of **log segments**, each representing a contiguous range of Log Sequence Numbers (LSNs) stored in a separate physical file. Segments are numbered sequentially and named using a predictable pattern that allows recovery processes to locate and order them correctly after a crash.

Each `LogSegment` maintains metadata about its size limits, current write position, and LSN range, enabling the `LogWriter` to make intelligent decisions about when rotation is necessary. The segment abstraction also enables advanced features like parallel recovery (different segments can be processed concurrently) and tiered storage (older segments can be moved to cheaper storage devices).

LogSegment Structure	Field Name	Type	Description
File Handle	<code>file</code>	<code>File</code>	Open file descriptor for writing
File Path	<code>path</code>	<code>PathBuf</code>	Full filesystem path to segment file
Current Size	<code>current_size</code>	<code>u64</code>	Bytes written to this segment
Maximum Size	<code>max_size</code>	<code>u64</code>	Size threshold triggering rotation
First LSN	<code>first_lsn</code>	<code>LSN</code>	Lowest LSN stored in this segment
Last LSN	<code>last_lsn</code>	<code>Option&lt;LSN&gt;</code>	Highest LSN written (None if empty)
Creation Time	<code>created_at</code>	<code>SystemTime</code>	When this segment was created
Sync Status	<code>last_sync_offset</code>	<code>u64</code>	File position of last fsync

### Rotation Trigger Conditions:

Log rotation occurs based on multiple configurable criteria that balance performance, manageability, and recovery efficiency. The most common trigger is segment size, but time-based rotation and LSN-based rotation provide additional control for specific deployment scenarios.

Size-based rotation prevents individual segments from becoming so large that they impact recovery performance or filesystem operations. Time-based rotation ensures that segments cover reasonable time windows for debugging and auditing purposes. LSN-based rotation aligns segment boundaries with checkpoint intervals, optimizing recovery startup times.

## Decision: Size-Based Primary Rotation with Time-Based Secondary Trigger

- **Context:** Need to balance segment manageability, recovery performance, and operational simplicity across diverse deployment scenarios
- **Options Considered:**
  1. Size-only rotation with fixed thresholds
  2. Time-only rotation with fixed intervals
  3. LSN-based rotation aligned with checkpoints
  4. Hybrid approach with multiple trigger conditions
- **Decision:** Implement size-based primary rotation with optional time-based secondary trigger
- **Rationale:** Size-based rotation provides predictable performance characteristics and bounded recovery times, while time-based secondary triggers ensure segments don't span excessive time periods under low write loads
- **Consequences:** Provides good default behavior for most workloads while allowing customization for specific operational requirements

Rotation Trigger	Condition	Default Threshold	Impact on Recovery
Size-Based	<code>current_size &gt;= max_size</code>	64MB	Bounded scan time
Time-Based	<code>age &gt;= max_age</code>	1 hour	Bounded time windows
LSN-Based	<code>lsn_count &gt;= max_lsns</code>	1M records	Checkpoint alignment
Manual	Administrator request	N/A	Operational control

### Atomic Rotation Protocol:

The rotation process must be carefully orchestrated to maintain crash safety and ensure no log records are lost during the transition between segments. The protocol involves creating the new segment, updating metadata atomically, and ensuring all pending writes complete before closing the old segment.

### Rotation Process Steps:

1. **Rotation Decision:** `Logwriter` detects that current segment has exceeded size threshold during a write operation or periodic check.
2. **Write Completion:** Complete any in-flight write operations to the current segment, ensuring all buffered records are flushed and `fsync'd`.
3. **New Segment Creation:** Generate next segment filename using sequential numbering, create and open new file with appropriate permissions and flags.

4. **Metadata Update:** Atomically update `LogWriter` state to point to new segment, recording the LSN range transition point.
5. **Old Segment Finalization:** Mark previous segment as read-only, update its metadata with final LSN range, and close file descriptor.
6. **Master Record Update:** Update master record (if used) to reflect new active segment, ensuring recovery can locate the current write position.
7. **Cleanup Notification:** Notify checkpoint manager and other components that segment rotation completed, enabling cleanup of obsolete segments.

The critical insight is that steps 3-4 must be atomic from the perspective of crash recovery. If a crash occurs during rotation, the recovery process must be able to determine whether the rotation completed successfully or needs to be rolled back.

Rotation State Machine	Current State	Event	Next State	Actions
Active Writing	Size Threshold Reached	Preparing Rotation	Flush buffers, fsync current segment	
Preparing Rotation	Flush Complete	Creating New Segment	Generate filename, create new file	
Creating New Segment	File Created	Updating Metadata	Atomically switch active segment	
Updating Metadata	Switch Complete	Finalizing Old Segment	Close old file, update metadata	
Finalizing Old Segment	Cleanup Complete	Active Writing	Resume normal operations	

### Segment Cleanup and Archival:

Once log segments are no longer needed for recovery (typically after a checkpoint has been completed and all active transactions have committed), they can be safely archived or deleted to reclaim disk space. The cleanup process must coordinate with the checkpoint manager to ensure that segments containing uncommitted transaction records or dirty page information are not removed prematurely.

The `LogWriter` maintains a **segment lifecycle** that tracks each segment's status from creation through deletion. Segments progress through states like Active (currently being written), Sealed (complete but may be needed for recovery), Archived (backed up to secondary storage), and Deleted (removed from primary storage).

**⚠ Pitfall: Premature Segment Deletion** A common mistake is deleting log segments before ensuring that all their data has been checkpointed or that no active transactions depend on them. This can make crash

recovery impossible if the system needs to replay operations from the deleted segments. Always coordinate segment cleanup with the checkpoint manager and verify that segment LSN ranges are covered by successful checkpoints before deletion.

## Common Pitfalls

**⚠ Pitfall: Ignoring `fsync()` Error Conditions** Many implementations assume that `fsync()` always succeeds and fail to handle cases where the underlying storage device encounters errors during the flush operation. If `fsync()` returns an error, it means the durability guarantee has been violated, and the transaction should be aborted rather than acknowledged to the client. Always check `fsync()` return values and implement appropriate error recovery.

**⚠ Pitfall: Buffer Size Misalignment with Filesystem Blocks** Writing log records that span filesystem block boundaries can create partial write scenarios where only part of a record reaches disk before a crash. This makes recovery complex because the system must detect and handle incomplete records. Always size buffers to align with filesystem block sizes (typically 4KB) and pad records as necessary.

**⚠ Pitfall: Race Conditions During Log Rotation** Concurrent write operations during log rotation can result in records being written to the wrong segment or lost entirely if the rotation process doesn't properly coordinate with ongoing writes. Use proper locking or atomic operations to ensure that rotation completes before new writes begin, and verify that all pending writes finish before closing old segments.

**⚠ Pitfall: Insufficient Back-Pressure Under High Load** Without proper flow control, fast transaction generators can overwhelm the `LogWriter`'s buffer capacity, leading to out-of-memory errors or degraded performance. Implement blocking or throttling mechanisms that slow down write operations when buffer space is exhausted, ensuring bounded memory usage under all load conditions.

**⚠ Pitfall: Log Corruption Due to Concurrent Access** Multiple processes or threads accessing log files concurrently can corrupt the log structure if not properly coordinated. Ensure that only a single `LogWriter` instance has write access to any given log segment, and use appropriate file locking mechanisms to prevent accidental concurrent access from other processes.

# Implementation Guidance

## A. Technology Recommendations

Component	Simple Option	Advanced Option
File I/O	<code>std::fs::File</code> with <code>write()</code> and <code>sync_all()</code>	<code>mmap</code> with <code>msync()</code> for high-performance scenarios
Serialization	Manual byte packing with <code>byteorder</code> crate	<code>bincode</code> or Protocol Buffers for complex records
Threading	<code>std::sync::Mutex</code> for write serialization	<code>tokio</code> async runtime for concurrent operations
Error Handling	<code>Result&lt;T, io::Error&gt;</code> with <code>?</code> operator	<code>anyhow</code> or <code>thiserror</code> for rich error context
Buffer Management	<code>Vec&lt;u8&gt;</code> with manual capacity management	Memory pools with <code>bumpalo</code> or similar
File System	Direct file operations with <code>std::fs</code>	<code>libc</code> bindings for advanced syscall control

## B. Recommended File/Module Structure

```
wal-implementation/
src/
    lib.rs                      ← Public API exports
    log_writer/
        mod.rs                   ← LogWriter public interface
        writer.rs                ← Core LogWriter implementation
        buffer.rs                ← Buffer management logic
        segment.rs               ← LogSegment implementation
        rotation.rs              ← Rotation coordination logic
    common/
        types.rs                 ← LSN, TransactionId, PageId types
        error.rs                 ← WalError and WalResult definitions
        crc.rs                   ← CRC32 checksum utilities
    test_utils/
        mock_fs.rs               ← Filesystem mocking for tests
        generators.rs            ← Test data generation
tests/
    integration/
        log_writer_tests.rs      ← End-to-end LogWriter tests
        crash_simulation.rs     ← Crash recovery scenarios
```

## C. Infrastructure Starter Code

**CRC32 Checksum Utilities ( `src/common/crc.rs` ):**

```
use crc32fast::Hasher;

/// Calculate CRC32 checksum for data integrity verification

pub fn calculate_crc32(data: &[u8]) -> u32 {

    let mut hasher = Hasher::new();

    hasher.update(data);

    hasher.finalize()

}

/// Validate that data matches expected CRC32 checksum

pub fn validate_checksum(data: &[u8], expected: u32) -> bool {

    calculate_crc32(data) == expected

}

/// Append CRC32 checksum to data buffer

pub fn append_checksum(data: &mut Vec<u8>) {

    let checksum = calculate_crc32(data);

    data.extend_from_slice(&checksum.to_le_bytes());

}

/// Extract and validate CRC32 from end of buffer

pub fn extract_and_validate_checksum(data: &[u8]) -> Result<&[u8], String> {

    if data.len() < 4 {

        return Err("Buffer too short for CRC32".to_string());

    }

    let (payload, checksum_bytes) = data.split_at(data.len() - 4);

    let expected_checksum = u32::from_le_bytes([
        checksum_bytes[0], checksum_bytes[1],
```

```
checksum_bytes[2], checksum_bytes[3]

]);


if validate_checksum(payload, expected_checksum) {

    ok(payload)
} else {

    Err("CRC32 validation failed".to_string())
}

}
```

**Error Handling ( `src/common/error.rs` ):**

```
use std::io;

use thiserror::Error;

pub type WalResult<T> = Result<T, WalError>;

#[derive(Error, Debug)]

pub enum WalError {

    #[error("I/O error: {0}")]
    Io(#[from] io::Error),

    #[error("Log corruption detected: {0}")]
    Corruption(String),

    #[error("Invalid log record format: {0}")]
    InvalidRecord(String),

    #[error("Log segment full, rotation required")]
    SegmentFull,

    #[error("Buffer overflow: attempted to write {attempted} bytes, {available} available")]
    BufferOverflow { attempted: usize, available: usize },

    #[error("Fsync failed: durability cannot be guaranteed")]
    FsyncFailed(#[source] io::Error),

    #[error("Log segment rotation failed: {0}")]
    RotationFailed(String),
```

```
}
```

**Buffer Pool Management ( `src/log_writer/buffer.rs` ):**

```
use std::collections::VecDeque;

use crate::common::error::{WalError, WalResult};

pub const SMALL_RECORD_SIZE: usize = 256;

pub const MEDIUM_RECORD_SIZE: usize = 1024;

pub const LARGE_RECORD_SIZE: usize = 4096;

pub const DEFAULT_SEGMENT_SIZE: u64 = 64 * 1024 * 1024; // 64MB

pub struct BufferPool {

    small_buffers: VecDeque<Vec<u8>>,

    medium_buffers: VecDeque<Vec<u8>>,

    large_buffers: VecDeque<Vec<u8>>,

    total_allocated: usize,

    max_total_size: usize,

}

impl BufferPool {

    pub fn new(max_total_size: usize) -> Self {

        Self {

            small_buffers: VecDeque::new(),

            medium_buffers: VecDeque::new(),

            large_buffers: VecDeque::new(),

            total_allocated: 0,

            max_total_size,

        }

    }

    pub fn get_buffer(&mut self, min_size: usize) -> WalResult<Vec<u8>> {


```

```
let buffer = if min_size <= SMALL_RECORD_SIZE {
    self.small_buffers.pop_front()
    .unwrap_or_else(|| Vec::with_capacity(SMALL_RECORD_SIZE))
} else if min_size <= MEDIUM_RECORD_SIZE {
    self.medium_buffers.pop_front()
    .unwrap_or_else(|| Vec::with_capacity(MEDIUM_RECORD_SIZE))
} else if min_size <= LARGE_RECORD_SIZE {
    self.large_buffers.pop_front()
    .unwrap_or_else(|| Vec::with_capacity(LARGE_RECORD_SIZE))
} else {
    return Err(WalError::BufferOverflow {
        attempted: min_size,
        available: LARGE_RECORD_SIZE
    });
}

Ok(buffer)
}

pub fn return_buffer(&mut self, mut buffer: Vec<u8>) {
    buffer.clear();
    let capacity = buffer.capacity();

    match capacity {
        SMALL_RECORD_SIZE => self.small_buffers.push_back(buffer),
        MEDIUM_RECORD_SIZE => self.medium_buffers.push_back(buffer),
        LARGE_RECORD_SIZE => self.large_buffers.push_back(buffer),
    }
}
```

```
    _ => {} // Don't pool non-standard sizes
  }
}

}
```

## D. Core Logic Skeleton Code

**LogSegment Implementation ( `src/log_writer/segment.rs` ):**

```
use std::fs::{File, OpenOptions};

use std::io::{Write, Seek, SeekFrom};

use std::path::PathBuf;

use crate::common::{LSN, WalResult, WalError};

pub struct LogSegment {

    pub file: File,

    pub path: PathBuf,

    pub current_size: u64,

    pub max_size: u64,

    pub first_lsn: Option<LSN>,

    pub last_lsn: Option<LSN>,

}

impl LogSegment {

    /// Create a new log segment file at the specified path

    pub fn create(path: PathBuf, max_size: u64) -> WalResult<Self> {

        // TODO 1: Create new file with write permissions and O_SYNC flag for durability

        // TODO 2: Initialize segment metadata with empty LSN ranges

        // TODO 3: Return LogSegment instance ready for writing

        todo!()

    }

    /// Append data to the end of this segment

    pub fn append(&mut self, data: &[u8]) -> WalResult<u64> {

        // TODO 1: Check if segment has space for this write (current_size + data.len() <= max_size)

        // TODO 2: Write data to file using single write() call

    }

}
```

```
// TODO 3: Update current_size with bytes written

// TODO 4: Return file offset where data was written

// Hint: Use Write::write_all() to ensure all bytes are written

todo!()

}

/// Force all buffered data to disk

pub fn sync(&mut self) -> WalResult<()> {

    // TODO 1: Call fsync() on file descriptor to force data to disk

    // TODO 2: Handle any errors by wrapping in WalError::FsyncFailed

    // TODO 3: Return success only if fsync completed without errors

    // Hint: Use File::sync_all() method in Rust

    todo!()

}

/// Check if segment has reached its size limit

pub fn is_full(&self) -> bool {

    // TODO: Return true if current_size >= max_size

    todo!()

}

/// Update the LSN range tracked by this segment

pub fn update_lsn_range(&mut self, lsn: LSN) {

    // TODO 1: If first_lsn is None, set it to the provided lsn

    // TODO 2: Always update last_lsn to the provided lsn

    // TODO 3: Ensure first_lsn <= last_lsn invariant is maintained

    todo!()

}
```

```
    }
```

```
}
```

**LogWriter Core Implementation ( `src/log_writer/writer.rs` ):**

```
use std::sync::{Mutex, Arc};

use std::path::PathBuf;

use crate::common::{LSN, TransactionId, WalResult, WalError};

use crate::data_model::LogRecord;

use super::segment::LogSegment;

use super::buffer::BufferPool;

pub struct LogWriter {

    current_segment: Arc<Mutex<LogSegment>>,

    buffer_pool: Arc<Mutex<BufferPool>>,

    base_path: PathBuf,

    segment_counter: Arc<Mutex<u64>>,

    next_lsn: Arc<Mutex<LSN>>,

}

impl LogWriter {

    /// Create a new LogWriter instance with specified configuration

    pub fn new(base_path: PathBuf, max_segment_size: u64) -> WalResult<Self> {

        // TODO 1: Create base directory if it doesn't exist

        // TODO 2: Initialize first log segment (segment_000000.log)

        // TODO 3: Set up buffer pool with default settings

        // TODO 4: Initialize LSN counter starting from 1

        // TODO 5: Return configured LogWriter instance

        todo!()

    }

    /// Write a log record and force it to disk before returning

    pub fn write_and_force(&mut self, record: &LogRecord) -> WalResult<LSN> {
```

```
// TODO 1: Assign next LSN to this record

// TODO 2: Serialize record to binary format with CRC32 checksum

// TODO 3: Acquire segment lock and check if rotation is needed

// TODO 4: If segment is full, perform rotation to new segment

// TODO 5: Append serialized record to current segment

// TODO 6: Call fsync() to ensure durability before returning

// TODO 7: Return the assigned LSN to caller

// Hint: Use serialize() method from LogRecord implementation

todo!()

}
```

```
/// Rotate to a new log segment when current segment is full

fn rotate_segment(&mut self) -> WalResult<()> {

    // TODO 1: Generate new segment filename using incremented counter

    // TODO 2: Create new LogSegment at generated path

    // TODO 3: Finalize current segment (close file, update metadata)

    // TODO 4: Atomically replace current_segment with new segment

    // TODO 5: Update segment_counter for next rotation

    // Hint: Use format!("segment_{:06}.log", counter) for consistent naming

    todo!()

}
```

```
/// Flush any buffered records without forcing sync

pub fn flush_buffer(&mut self) -> WalResult<()> {

    // TODO 1: Acquire buffer pool lock

    // TODO 2: Write any accumulated records to current segment

    // TODO 3: Clear buffer pool to free memory
```

```
// TODO 4: Do not call fsync() - this is for performance optimization only
todo!()

}

/// Force sync current segment to disk

pub fn force_sync(&mut self) -> WalResult<()> {

    // TODO 1: Acquire segment lock

    // TODO 2: Call sync() method on current segment

    // TODO 3: Handle any fsync errors appropriately

    // TODO 4: Return success only if sync completed

    todo!()

}

/// Get the next available LSN without advancing counter

pub fn peek_next_lsn(&self) -> LSN {

    // TODO: Return current value of next_lsn counter without incrementing

    todo!()

}

/// Advance LSN counter and return new value

fn allocate_lsn(&mut self) -> LSN {

    // TODO 1: Acquire next_lsn lock

    // TODO 2: Increment counter by 1

    // TODO 3: Return new counter value

    // TODO 4: Ensure LSNs are never reused or duplicated

    todo!()

}
```

```
}
```

## E. Language-Specific Hints

### Rust-Specific Implementation Tips:

- **File I/O:** Use `std::fs::OpenOptions` with `.create(true).append(true).write(true)` for log files. Consider `.custom_flags(libc::O_SYNC)` on Unix for immediate disk writes without separate `fsync` calls.
- **Error Handling:** Leverage the `?` operator extensively for clean error propagation. Use `thiserror` crate for rich error context that preserves underlying I/O errors.
- **Concurrency:** `Arc<Mutex<T>>` provides shared ownership with interior mutability. Consider `tokio::sync::Mutex` for async contexts or `parking_lot::Mutex` for better performance.
- **Memory Management:** Use `Vec::with_capacity()` for buffers with known sizes. Call `Vec::clear()` instead of creating new vectors to reuse allocated memory.
- **Serialization:** The `byteorder` crate provides endian-aware integer serialization. Use `LittleEndian::write_u64()` for consistent cross-platform format.
- **Path Handling:** `PathBuf` and `Path` handle platform differences automatically. Use `path.join()` instead of string concatenation for cross-platform compatibility.

## F. Milestone Checkpoint

### Testing Your LogWriter Implementation:

After implementing the core LogWriter functionality, verify correct behavior with these tests:

```
# Run unit tests for individual components

cargo test log_writer::tests::test_segment_creation
cargo test log_writer::tests::test_append_and_sync
cargo test log_writer::tests::test_rotation_triggers

# Run integration tests for complete workflows

cargo test integration::test_write_and_force_single_record
cargo test integration::test_concurrent_writers
cargo test integration::test_crash_during_rotation
```

BASH

### Expected Behaviors to Verify:

- Single Record Write:** Create LogWriter, write one record, verify file exists with correct size and fsync completes successfully.
- Buffer Management:** Write multiple small records, verify they accumulate in buffers, then force flush and confirm all data reaches disk.
- Log Rotation:** Write records until segment size limit is reached, verify new segment is created and old segment is finalized properly.
- Concurrent Access:** Start multiple threads writing records simultaneously, verify all records appear in correct LSN order without corruption.
- Error Handling:** Simulate disk full condition, verify appropriate errors are returned and system remains in consistent state.

### Signs of Implementation Issues:

Symptom	Likely Cause	Diagnostic Steps	Fix Approach
Records lost after restart	Missing fsync() calls	Check if <code>sync_all()</code> is called before acknowledging writes	Add fsync after every <code>write_and_force</code>
File corruption detected	Concurrent writes without locking	Use <code>hexdump</code> to examine log files for overlapping data	Add proper mutex protection
Out of memory errors	Buffer pool not releasing memory	Monitor memory usage during high write load	Implement buffer return/reuse
Slow write performance	Excessive fsync frequency	Profile fsync call frequency vs batch size	Implement group commit batching
Rotation failures	Race condition during segment switch	Check for incomplete rotation state	Add atomic rotation protocol

## Recovery Manager Component

**Milestone(s):** Milestone 3 (Crash Recovery) - this section implements ARIES-style crash recovery with three-phase recovery algorithm: Analysis, Redo, and Undo phases to restore database consistency after crashes

### ARIES Recovery Overview

Think of ARIES recovery like **reconstructing a crime scene** after everyone has fled. The Write-Ahead Log is your evidence tape - it recorded everything that happened in chronological order. But the scene itself (the database) might be in a chaotic state with some actions completed, others half-done, and witnesses (active

transactions) nowhere to be found. ARIES gives you a systematic three-phase investigation process: first **analyze the evidence** to understand what was happening when chaos struck, then **reconstruct all the completed actions** to see the full picture, and finally **undo any incomplete actions** to return to a consistent state.

The **ARIES (Algorithm for Recovery and Isolation Exploiting Semantics)** recovery algorithm represents one of the most elegant solutions to the crash recovery problem in database systems. Unlike simpler recovery schemes that might take conservative approaches like "undo everything and restart," ARIES recognizes that crashes often occur when many transactions have legitimately completed their work and committed - throwing away all that progress would be wasteful and slow.

The genius of ARIES lies in its **three-phase approach** that separates concerns cleanly. Each phase has a distinct responsibility and operates on different data structures, yet they work together to guarantee that recovery will restore the database to a consistent state that reflects exactly the work that should have been durable before the crash.

### Decision: Three-Phase Recovery Architecture

- **Context:** After a crash, the database state is unknown - some committed changes might not be on disk, and some uncommitted changes might be partially applied
- **Options Considered:** Single-phase "undo everything", two-phase "redo then undo", three-phase ARIES approach
- **Decision:** Implement full three-phase ARIES recovery with Analysis, Redo, and Undo phases
- **Rationale:** Three phases provide optimal recovery time by precisely identifying what needs to be redone vs undone, and the analysis phase enables efficient targeted recovery rather than scanning the entire database
- **Consequences:** More complex implementation but significantly faster recovery times and guaranteed correctness under all crash scenarios

The **Analysis Phase** serves as the reconnaissance mission. It scans forward through the log starting from the most recent checkpoint, building two critical data structures: the **Transaction Table** (which transactions were active at crash time and what state they were in) and the **Dirty Page Table** (which database pages had uncommitted changes and might need attention). This phase doesn't touch the database itself - it's purely about understanding what the log tells us about the state at crash time.

Analysis Phase Output	Purpose	Contents
Transaction Table	Track transaction states	TransactionId → TransactionState mapping
Dirty Page Table	Identify potentially inconsistent pages	PageId → earliest LSN that dirtied this page
Redo Start Point	Optimization for redo phase	Minimum LSN from dirty page table
Committed Transaction Set	Fast lookup during undo	Set of TransactionIds that completed successfully

The **Redo Phase** acts as the faithful historian, replaying all committed work to ensure it's properly reflected in the database. This phase operates under the principle of **idempotency** - applying the same change multiple times produces the same result. Redo starts from the earliest LSN that might have affected any dirty page (computed during analysis) and moves forward chronologically, reapplying every change from committed transactions. The key insight is that we redo *everything* first, even changes from transactions that will later be undone - this ensures we have a complete, consistent view of what the database looked like just before the crash.

The **Undo Phase** serves as the cleanup crew, systematically removing the effects of any transaction that was still in progress when the crash occurred. Unlike redo, which moves forward through time, undo works backward through the chain of changes made by each incomplete transaction. This phase uses the **undo\_next\_lsn** field stored in log records to efficiently traverse the undo chain for each transaction without having to scan the entire log backward.

The critical insight is that ARIES separates "what changes were made" (redo) from "which changes should remain" (undo). This allows recovery to proceed efficiently without having to make complex decisions about each individual record during the scan.

## ARIES Recovery State Machine

The recovery process follows a well-defined state machine that ensures consistent progress even if recovery itself is interrupted by another crash:

Current State	Event	Next State	Action Taken
Crashed	Recovery initiated	Analyzing	Start analysis phase from checkpoint
Analyzing	Log scan complete	Redoing	Build transaction and dirty page tables
Redoing	All committed changes replayed	Undoing	Begin processing incomplete transactions
Undoing	All incomplete transactions rolled back	Recovered	Database ready for new transactions
Recovered	Normal operation resumes	Running	Accept new transactions and log writes

**Recovery Manager Architecture:** The `RecoveryManager` component orchestrates all three phases and maintains the state necessary for crash recovery. It depends on the `LogReader` for accessing historical log records, the storage interface for applying changes to database pages, and the checkpoint system for determining recovery starting points.

RecoveryManager Field	Type	Purpose
<code>log_reader</code>	<code>LogReader</code>	Provides sequential access to log records
<code>storage</code>	<code>Arc</code>	Interface to database pages for applying changes
<code>transaction_table</code>	<code>HashMap&lt;TransactionId, TransactionState&gt;</code>	Tracks active transactions and their undo chains
<code>dirty_page_table</code>	<code>HashMap&lt;PageId, LSN&gt;</code>	Maps dirty pages to earliest LSN that modified them
<code>checkpoint_lsn</code>	<code>Option</code>	Starting point for recovery scan
<code>redo_start_lsn</code>	<code>Option</code>	Earliest LSN that needs to be redone

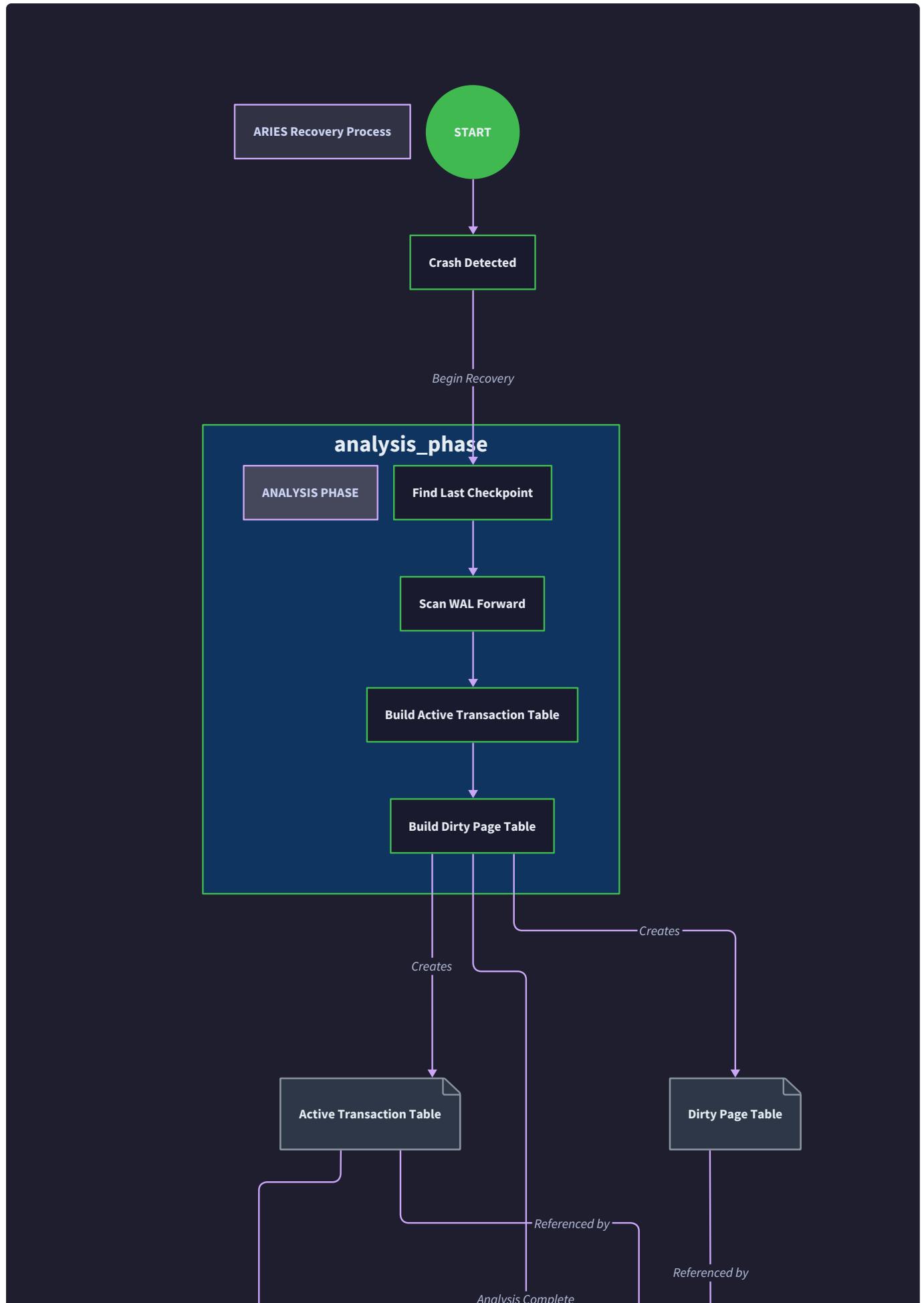
## Log Scanning and Analysis

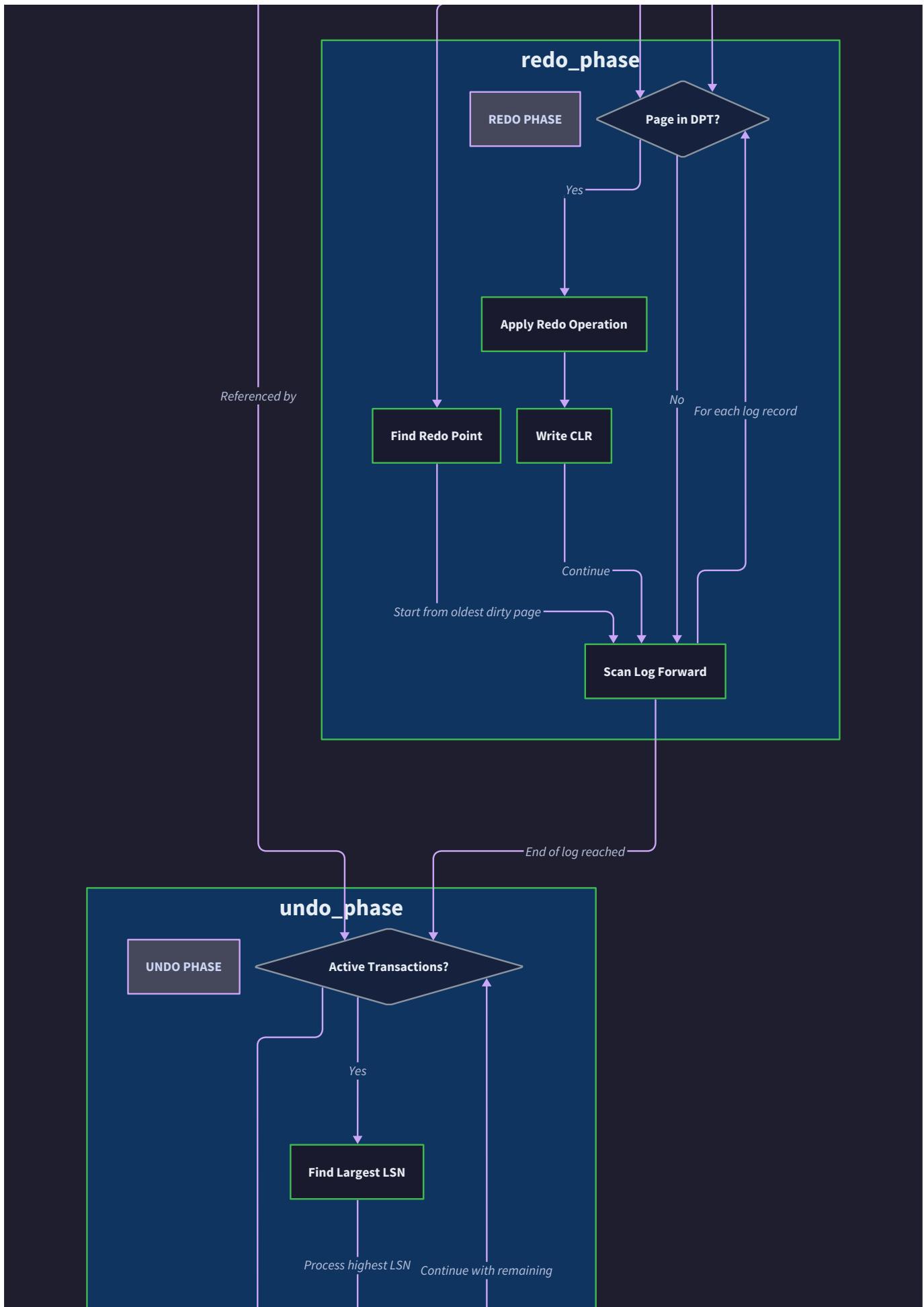
The Analysis Phase begins recovery by **building a complete picture** of transaction activity from the log records. Think of it like an archaeologist carefully examining layers of sediment - each log record represents a moment in time, and by examining them in sequence, we can reconstruct the timeline of events that led to the crash.

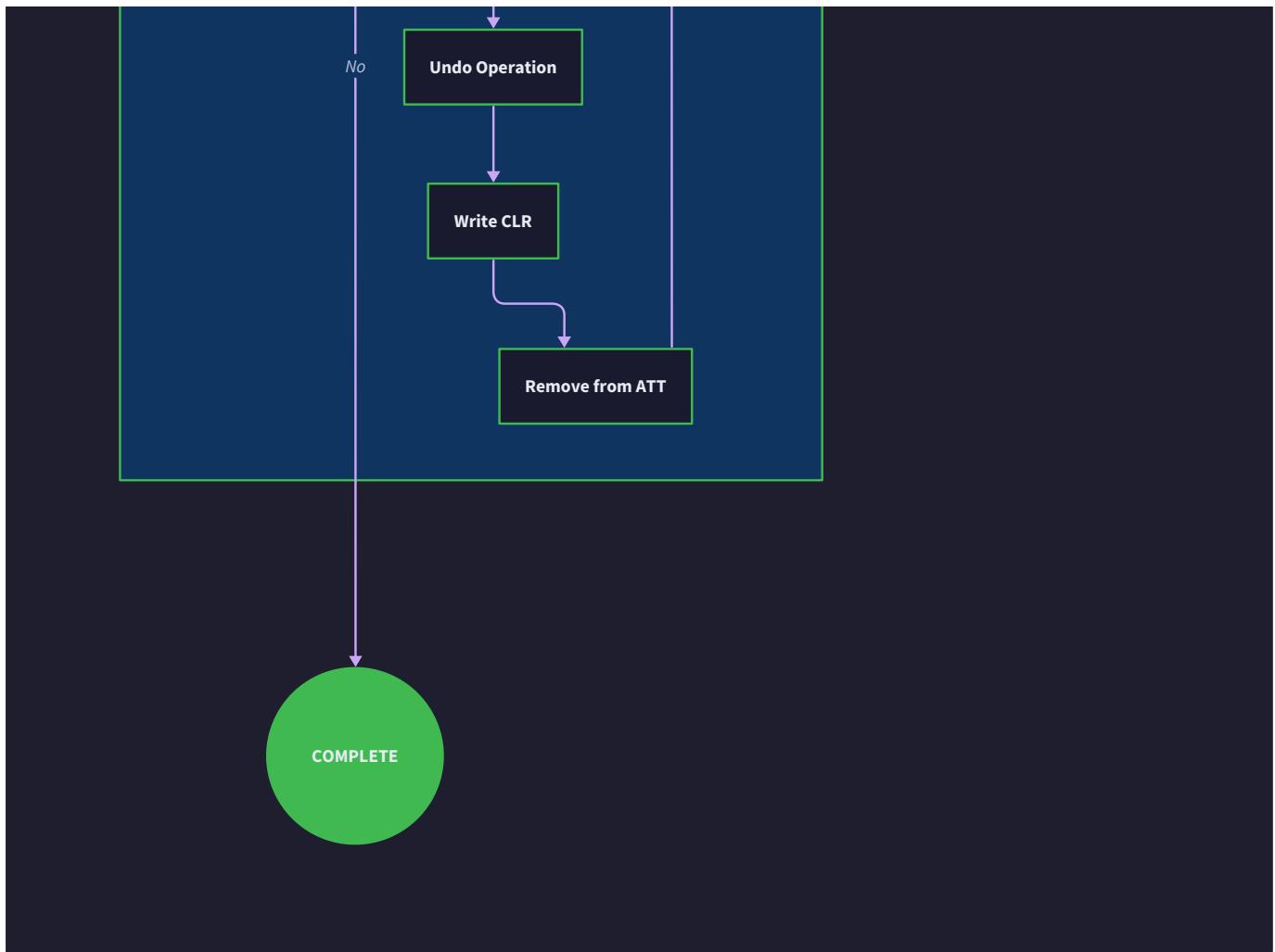
**Analysis Phase Algorithm:** The analysis pass scans forward from the most recent checkpoint (or from the beginning of the log if no checkpoint exists) to the end of the log. For each log record encountered, it updates

the transaction table and dirty page table to reflect the operations that were in progress when the crash occurred.

1. **Initialize Recovery State:** Load the most recent checkpoint record to establish the starting point for analysis. If a checkpoint exists, populate the transaction table and dirty page table with the checkpoint's recorded active transactions and dirty pages. If no checkpoint is found, start with empty tables and begin scanning from the first log record.
2. **Sequential Log Scanning:** Read log records sequentially from the checkpoint LSN (or LSN 0) to the end of the log file. The log reader must handle potentially corrupted records at the end of the log - the last record might be partially written if the crash occurred during a log write operation.
3. **Transaction State Tracking:** For each log record that belongs to a transaction, update the transaction table entry. If this is the first record seen for a transaction (indicating it started after the checkpoint), create a new transaction table entry with status `Active`. Update the transaction's `last_lsn` to reflect the most recent log record from this transaction.
4. **Dirty Page Management:** For `RedoRecord` entries, check if the affected page is already in the dirty page table. If not, add it with the current record's LSN as the recovery LSN - this represents the earliest change that might not be reflected on disk. If the page is already tracked, leave the existing recovery LSN unchanged (we want the earliest LSN).
5. **Transaction Completion Handling:** When encountering `CommitRecord` or `AbortRecord` entries, update the transaction's status in the transaction table. Committed transactions remain in the table but are marked as completed, while aborted transactions are also retained to ensure proper cleanup during undo.
6. **Undo Chain Construction:** For transactions that remain active, construct the undo chain by linking log records in reverse chronological order. Each `RedoRecord` and `UndoRecord` contains an implicit link to the previous log record for the same transaction, allowing efficient backward traversal during the undo phase.







**Analysis Results Data Structures:** The analysis phase produces several critical data structures that guide the subsequent redo and undo phases:

Data Structure	Key	Value	Purpose
Transaction Table	TransactionId	TransactionState	Tracks which transactions were active and their undo starting points
Dirty Page Table	PageId	LSN	Identifies pages that may have uncommitted changes
Committed Transactions	TransactionId	()	Fast lookup set for redo phase filtering
Redo Start LSN	-	LSN	Minimum LSN from dirty page table - optimization for redo phase

The **Transaction State** structure captures everything needed to properly handle each transaction during recovery:

TransactionState Field	Type	Purpose
status	TransactionStatus	Whether transaction committed, aborted, or was still active
last_lsn	LSN	Most recent log record for this transaction
undo_next_lsn	Option	Starting point for undo chain traversal

### Decision: Build Complete Transaction State During Analysis

- **Context:** Recovery needs to know which transactions were committed vs. active, and how to efficiently undo active transactions
- **Options Considered:** Lazy state building during redo/undo, complete state building during analysis, hybrid approach
- **Decision:** Build complete transaction and dirty page tables during analysis phase
- **Rationale:** Front-loading the analysis work enables the redo and undo phases to operate efficiently without re-scanning the log, and provides clear separation of concerns
- **Consequences:** Analysis phase takes longer but redo/undo phases are much faster and more predictable

**Handling Log Corruption During Analysis:** The analysis phase must be robust against various forms of log corruption that can occur during crashes. The most common scenario is a **partial write** where the crash occurred during a log record write, leaving an incomplete record at the end of the log.

Corruption Type	Detection Method	Recovery Strategy
Partial Record	CRC mismatch or truncated data	Stop analysis at last valid record
Invalid LSN	LSN not monotonically increasing	Stop analysis, treat as end of valid log
Invalid Transaction ID	Transaction ID format validation fails	Skip record, continue analysis
Corrupted Record Type	Unknown record type encountered	Skip record, log warning, continue

### Redo Phase Implementation

The Redo Phase implements the principle of **"history repeats itself"** - every change that was committed before the crash must be replayed to ensure durability guarantees are met. Think of redo like a video replay system that can fast-forward through the game tape and re-execute every play that was officially recorded, regardless of whether the scoreboard was updated before the power went out.

**Redo Phase Objectives:** The fundamental goal of redo is to restore the database to a state that reflects all committed work, even if some of that work wasn't written to disk before the crash due to buffer pool management or delayed writes. Redo operates under the **WAL Protocol invariant**: every change recorded in the log as committed must eventually be reflected in the database state.

**Idempotent Redo Operations:** A critical property of redo operations is **idempotency** - applying the same change multiple times produces the same final result. This property is essential because recovery might be interrupted and restarted, potentially causing the same log records to be processed multiple times.

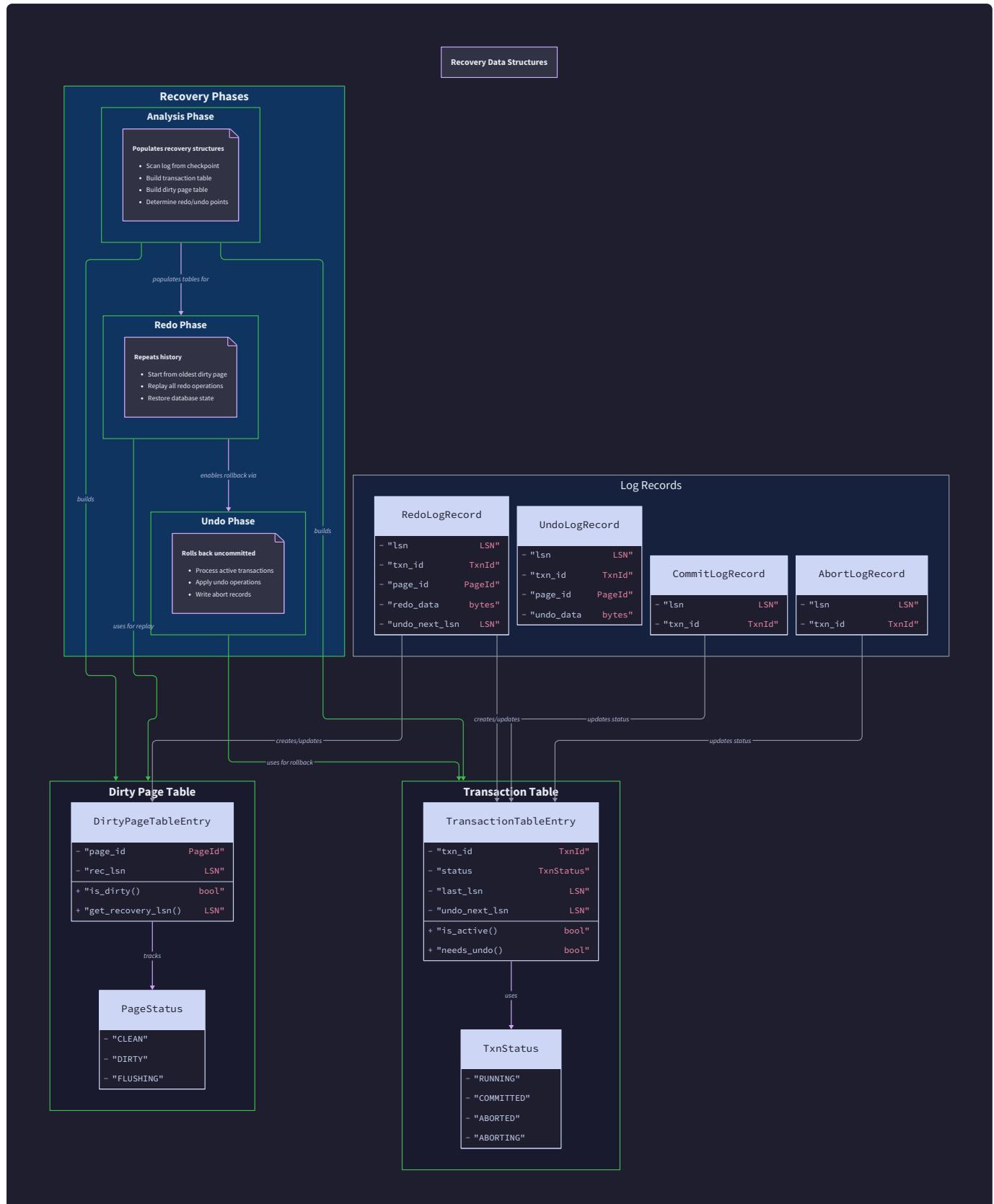
1. **Determine Redo Starting Point:** Begin redo from the minimum LSN found in the dirty page table during analysis. This optimization avoids reprocessing log records for pages that were definitely clean at checkpoint time. If the dirty page table is empty, no redo is necessary - all committed changes were already on disk.
2. **Sequential Forward Scan:** Read log records sequentially starting from the redo start LSN and proceeding to the end of the log. Unlike analysis, redo must actually apply changes, so it processes only `RedoRecord` entries and ignores transaction control records like commits and aborts.
3. **Transaction Filtering:** For each `RedoRecord`, verify that it belongs to a transaction that was committed before the crash by consulting the transaction table built during analysis. Skip any records from transactions that were still active - their changes will be handled by the undo phase.
4. **Page-Level Redo:** For each valid redo record, retrieve the target page from storage and apply the change. The redo record contains the `after_image` (the new data state) that should be written to the specified page and offset. Update the page's LSN to match the log record's LSN to maintain the WAL protocol invariant.
5. **LSN Comparison Optimization:** Before applying a redo record to a page, compare the record's LSN with the page's current LSN. If the page's LSN is already equal to or greater than the record's LSN, the change has already been applied (possibly during a previous recovery attempt) and can be skipped. This optimization ensures idempotency and improves performance.
6. **Force Page Writes:** After applying each change, the modified page must be written back to storage. Some implementations batch these writes for efficiency, but the WAL protocol requires that the page's LSN never exceed the durability point of the log.

**Redo Implementation Interface:** The redo phase implementation centers around systematic application of committed changes:

Method Name	Parameters	Returns	Description
redo_pass	&mut self	WalResult<()>	Execute complete redo phase from analysis results
apply_redo_record	&mut self, record: RedoRecord	WalResult<()>	Apply single redo record to target page
is_transaction_committed	&self, txn_id: TransactionId	bool	Check if transaction committed before crash
get_page_for_redo	&mut self, page_id: PageId	WalResult	Retrieve page from storage for modification
update_page_with_redo	&mut self, page: DatabasePage, record: RedoRecord	WalResult<()>	Apply redo change and update page LSN

### Redo Algorithm Detailed Steps:

- 7. Initialize Redo Context:** Set up the redo phase by determining the starting LSN from the dirty page table minimum. Initialize counters for tracking redo progress and performance metrics. Prepare the storage interface for potentially intensive page update operations.
- 8. Process Each Redo Record:** For every log record in the redo range, extract the `RedoRecord` variant and validate its structure. Ensure the transaction ID refers to a committed transaction from the analysis results. Skip any records that don't meet these criteria but continue processing subsequent records.
- 9. Page Retrieval and Validation:** Load the target page from storage using the `page_id` from the redo record. Handle cases where the page doesn't exist (it may need to be allocated) or where storage I/O fails. Validate that the page structure is consistent before attempting modifications.
- 10. LSN-Based Skip Optimization:** Compare the redo record's LSN with the page's current LSN. If `page.lsn >= record.lsn`, this change has already been applied and can be safely skipped. This handles cases where recovery was previously interrupted and restarted.
- 11. Apply Data Change:** Write the `after_image` data from the redo record to the specified offset within the target page. Update the page's LSN to match the redo record's LSN, maintaining the invariant that the page LSN reflects the most recent logged change applied to this page.
- 12. Persist Modified Page:** Write the updated page back to storage to ensure the redo change is durable. Depending on the storage implementation, this might involve immediate writes or batching for efficiency, but the WAL protocol requires that log durability precede page durability.



## Decision: Forward-Only Redo with LSN Skip Optimization

- **Context:** Redo must efficiently handle potentially large numbers of log records while ensuring idempotency
- **Options Considered:** Full page scanning for each record, LSN-based skip optimization, change tracking bitmaps
- **Decision:** Implement LSN-based skip optimization where pages with  $LSN \geq$  record LSN are skipped
- **Rationale:** This approach provides excellent performance for repeated recovery attempts while maintaining simplicity and correctness
- **Consequences:** Requires careful LSN management but eliminates redundant work during recovery restarts

**Redo Error Handling:** The redo phase must be resilient to various error conditions while maintaining database consistency:

Error Scenario	Detection	Recovery Action
Storage I/O Failure	IOException during page read/write	Retry with exponential backoff, fail recovery if persistent
Page Corruption	CRC mismatch or invalid page structure	Log error, skip page, continue with other pages
Invalid After-Image	Data validation fails for redo record	Skip this record, log corruption warning
Insufficient Storage Space	Disk full during page write	Fail recovery - requires manual intervention

## Undo Phase Implementation

The Undo Phase serves as the **cleanup operation** that systematically removes the effects of transactions that were still active when the crash occurred. Think of undo like a skilled editor who can precisely remove specific sentences from a document without disturbing the surrounding text - each incomplete transaction must be rolled back cleanly without affecting the committed work of other transactions.

**Backward Chain Traversal:** Unlike redo, which processes log records in chronological order, undo must work backward through each transaction's history. The challenge is that we can't simply reverse-scan the entire log (which would be inefficient), so ARIES uses **undo chains** where each log record for a transaction contains an implicit or explicit link to the previous record from the same transaction.

**Undo Phase Objectives:** The fundamental goal is to ensure that the database reflects only committed transactions. Every transaction that was active at crash time must have all its changes systematically reversed

using the `before_image` data stored in undo records. This phase maintains the **atomicity** property of transactions - a transaction either commits completely or has no effect on the database.

1. **Identify Active Transactions:** Using the transaction table built during analysis, identify all transactions with status `Active`. These are the transactions that were in progress when the crash occurred and must be rolled back. Sort them by `last_lsn` to process in reverse chronological order for efficiency.
2. **Initialize Undo Chains:** For each active transaction, begin the undo chain traversal from the transaction's `last_lsn`. Each transaction maintains its own undo chain, allowing multiple transactions to be undone concurrently without interference.
3. **Backward Chain Walking:** Follow the undo chain for each transaction by reading log records in reverse order. The `undo_next_lsn` field in each record points to the previous record for the same transaction, creating an efficient backward traversal path without full log scanning.
4. **Apply Undo Operations:** For each `UndoRecord` or `RedoRecord` encountered during chain traversal, apply the inverse operation using the `before_image` data. This restores the page content to its state before the transaction's change was applied.
5. **Generate Compensation Log Records:** Write **Compensation Log Records (CLRs)** to the log for each undo operation performed. CLRs ensure that undo operations themselves are logged and can be redone if recovery is interrupted. CLRs are never undone - they represent forward progress in the undo process.
6. **Transaction Completion:** When the undo chain for a transaction is completely processed (reaching a record with `undo_next_lsn = None`), write an `AbortRecord` to the log and remove the transaction from the active transaction set.

**Undo Implementation Interface:** The undo phase provides a systematic approach to transaction rollback:

Method Name	Parameters	Returns	Description
undo_pass	&mut self	WalResult<()>	Execute complete undo phase for all active transactions
undo_transaction	&mut self, txn_id: TransactionId	WalResult<()>	Roll back single transaction completely
apply_undo_record	&mut self, record: UndoRecord	WalResult	Apply single undo operation, return CLR LSN
write_compensation_record	&mut self, original: LSN, txn_id: TransactionId	WalResult	Generate CLR for undo operation
traverse_undo_chain	&self, start_lsn: LSN, txn_id: TransactionId	impl Iterator<Item=LogRecord>	Iterator over transaction's undo chain

**Compensation Log Records (CLRs):** A critical aspect of ARIES undo is that **undo operations are themselves logged**. When we undo a change, we write a Compensation Log Record that describes the undo operation. This ensures that if recovery is interrupted during the undo phase, the undo work already completed doesn't need to be repeated when recovery restarts.

CLR Field	Type	Purpose
lsn	LSN	Unique identifier for this compensation record
txn_id	TransactionId	Transaction being undone
undo_next_lsn	Option	Next record in undo chain (continues backward traversal)
page_id	PageId	Page modified by undo operation
before_image	Vec	Original data before the undone change
after_image	Vec	Data state after undo (restoration of original state)

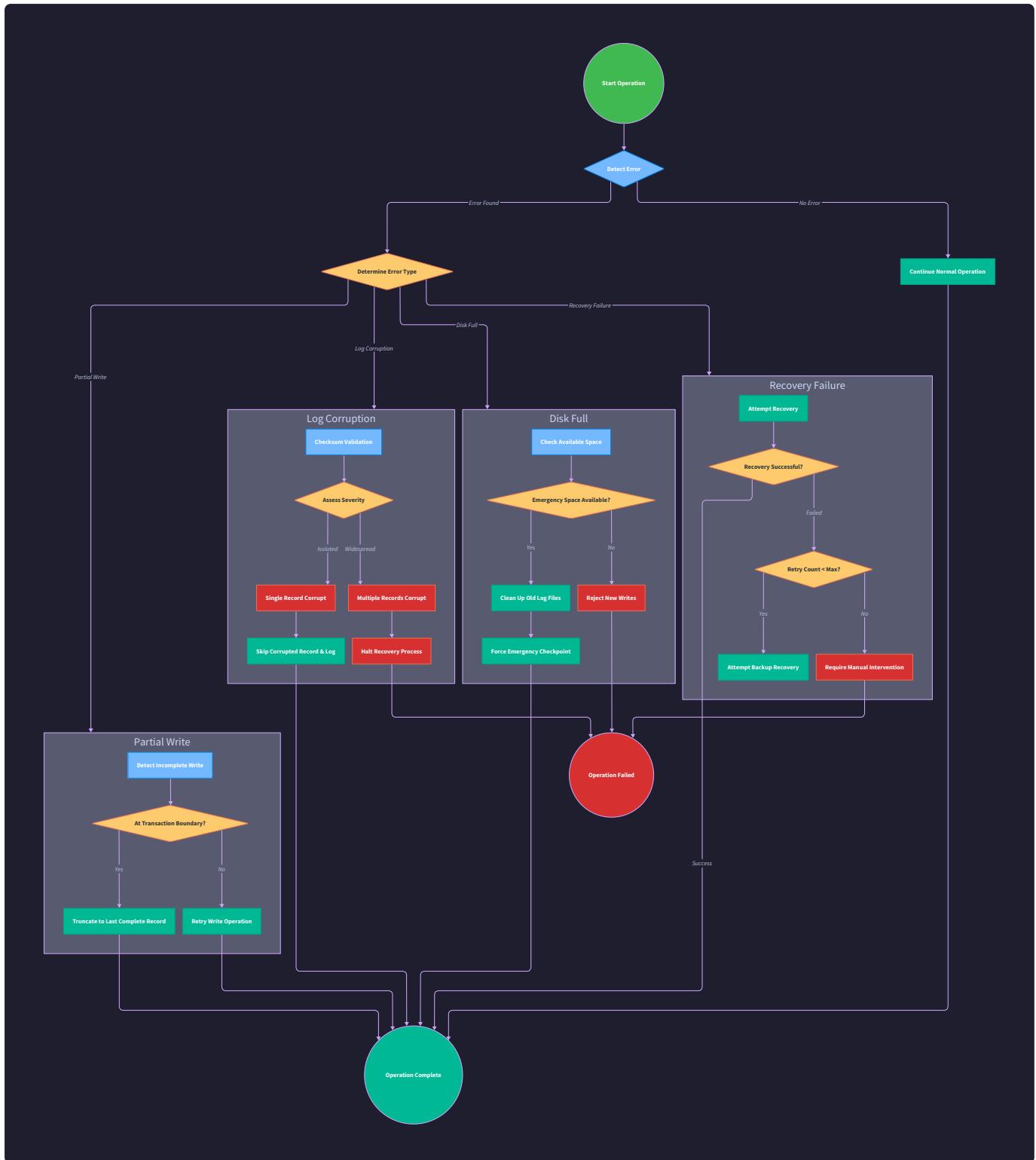
#### Detailed Undo Algorithm:

- 7. Concurrent Transaction Undo:** Process multiple active transactions concurrently by maintaining separate undo contexts for each transaction. This improves undo performance and ensures that one problematic transaction doesn't block the rollback of others.
- 8. Read Log Record for Undo:** For each LSN in the undo chain, read the corresponding log record from storage. Handle cases where the record might be corrupted or inaccessible by logging errors and continuing with other transactions where possible.

9. **Determine Undo Operation:** Based on the log record type, determine the appropriate undo action. `RedoRecord` entries require applying the `before_image` to reverse the change. Existing `UndoRecord` entries (CLRs from previous recovery attempts) should advance the undo chain without reapplying changes.
10. **Page-Level Undo Application:** Retrieve the target page and apply the undo operation by writing the `before_image` data to the specified offset. Update the page's LSN to reflect the most recent change (which could be from redo or other undo operations).
11. **CLR Generation and Logging:** Write a Compensation Log Record describing the undo operation. The CLR must be forced to disk before the undo operation is considered complete, maintaining the WAL protocol even during recovery.
12. **Advance Undo Chain:** Move to the next record in the undo chain using the `undo_next_lsn` field. If this field is `None`, the transaction has been completely undone and can be marked as aborted.

#### Decision: Compensation Log Records for Undo Operations

- **Context:** Undo operations can be interrupted by crashes, potentially leaving the database in an inconsistent state between original and undone versions
- **Options Considered:** In-memory undo tracking, checkpoint-based undo restart, compensation log records
- **Decision:** Generate Compensation Log Records (CLRs) for every undo operation performed
- **Rationale:** CLRs ensure that partial undo work is never lost or repeated, and they can be redone like normal operations if recovery is interrupted
- **Consequences:** Slightly increases log volume during recovery but provides bulletproof correctness guarantees



## Undo Error Handling and Edge Cases:

Error Scenario	Detection Method	Recovery Strategy
Corrupted Undo Chain	Invalid LSN or missing record	Skip corrupted portion, continue from last valid point
Page Not Found	Storage returns page not found	Log warning, assume page was never allocated
Insufficient Log Space	Disk full during CLR write	Critical error - must free space before continuing
Concurrent Access	Multiple recovery processes	Use file locking to ensure single recovery process

**Active Transaction Prioritization:** When multiple transactions need to be undone, the order can affect recovery performance. ARIES typically processes transactions in reverse order of their last activity (most recent first) to maximize the probability that recent changes are still in buffer caches.

Undo Priority	Criteria	Rationale
Highest	Transactions with recent activity (high last_lsn)	More likely to have pages in cache
Medium	Long-running transactions with many changes	Free up resources held by large transactions
Lowest	Transactions with only metadata changes	Minimal resource impact

## Common Pitfalls

**⚠ Pitfall: Incorrect LSN Ordering During Analysis** During the analysis phase, developers often assume that log records are processed in perfect LSN order, but log segments and buffering can cause records to be written slightly out of order. The analysis phase must handle cases where a later LSN is processed before an earlier one from a different transaction. Always use the LSN value from the record itself, not the order in which records are encountered during scanning.

**⚠ Pitfall: Missing Transaction State Initialization** When encountering the first log record for a transaction (one not present in the checkpoint), developers often forget to initialize all fields of the `TransactionState` structure. An uninitialized `undo_next_lsn` field can cause the undo phase to fail catastrophically. Always create complete transaction state entries with proper defaults for all fields.

**⚠ Pitfall: Skipping CLR Generation During Undo** The temptation to optimize undo performance by skipping Compensation Log Record generation is dangerous. If recovery is interrupted during undo, the database can be left in an inconsistent state with some changes undone but not logged. Every undo operation must generate a CLR and force it to disk before the undo is considered complete.

**⚠ Pitfall: Incorrect Dirty Page LSN Tracking** During analysis, when a page is already in the dirty page table, developers sometimes update the LSN to the current record's LSN instead of keeping the earliest LSN. The dirty page table should track the *earliest* LSN that dirtied each page, not the most recent. This LSN determines where redo must start for that page.

**⚠ Pitfall: Applying Redo to Uncommitted Transactions** The redo phase must only replay changes from committed transactions, but it's easy to accidentally include records from active transactions. Always filter redo

records by checking the transaction's final status in the transaction table. Redoing uncommitted changes that will later be undone can cause subtle consistency issues.

**⚠ Pitfall: Infinite Undo Chain Loops** When constructing undo chains, circular references in the `undo_next_lsn` fields can cause infinite loops during the undo phase. Always validate that `undo_next_lsn` points to a strictly earlier LSN, and implement loop detection with a visited set or maximum iteration count.

## Implementation Guidance

### A. Technology Recommendations Table:

Component	Simple Option	Advanced Option
Log Reading	Sequential file reading with buffering	Memory-mapped files with lazy loading
Page Storage	HashMap-based in-memory pages	B-tree or LSM-tree persistent storage
Transaction Tracking	HashMap<TransactionId, TransactionState>	Concurrent hash map with sharding
Error Handling	Result types with simple error propagation	Structured error types with context chains
Concurrency	Single-threaded recovery	Parallel redo/undo with dependency tracking

### B. Recommended File/Module Structure:

```
src/
  recovery/
    mod.rs           ← Public interface and RecoveryManager
    analysis.rs      ← Analysis phase implementation
    redo.rs          ← Redo phase implementation
    undo.rs          ← Undo phase implementation
    transaction_table.rs  ← Transaction state tracking
    dirty_page_table.rs  ← Dirty page tracking
    compensation.rs   ← Compensation Log Record handling
  storage/
    mock_storage.rs  ← Test storage implementation
    storage_trait.rs ← Storage abstraction
  log/
    log_reader.rs    ← Sequential log reading
    log_record.rs    ← Record types and serialization
```

### C. Infrastructure Starter Code:

```
// src/recovery/transaction_table.rs

use std::collections::HashMap;

use crate::types::{TransactionId, LSN, TransactionStatus};

#[derive(Debug, Clone)]

pub struct TransactionState {

    pub status: TransactionStatus,

    pub last_lsn: LSN,

    pub undo_next_lsn: Option<LSN>,

}

impl TransactionState {

    pub fn new_active(first_lsn: LSN) -> Self {

        Self {

            status: TransactionStatus::Active,

            last_lsn: first_lsn,

            undo_next_lsn: None,

        }

    }

    pub fn update_lsn(&mut self, lsn: LSN, prev_lsn: Option<LSN>) {

        self.undo_next_lsn = Some(self.last_lsn);

        self.last_lsn = lsn;

    }

    pub fn commit(&mut self) {

        self.status = TransactionStatus::Committed;

    }

}
```

```
pub fn abort(&mut self) {  
    self.status = TransactionStatus::Aborted;  
}  
}  
  
pub struct TransactionTable {  
    transactions: HashMap<TransactionId, TransactionState>,  
}  
  
impl TransactionTable {  
    pub fn new() -> Self {  
        Self {  
            transactions: HashMap::new(),  
        }  
    }  
  
    pub fn get_or_create(&mut self, txn_id: TransactionId, lsn: LSN) -> &mut TransactionState {  
        self.transactions.entry(txn_id).or_insert_with(||  
            TransactionState::new_active(lsn))  
    }  
  
    pub fn get(&self, txn_id: TransactionId) -> Option<&TransactionState> {  
        self.transactions.get(&txn_id)  
    }  
  
    pub fn active_transactions(&self) -> impl Iterator<Item = (TransactionId,  
        &TransactionState)> {
```

```
    self.transactions.iter()

        .filter(|(_, state)| matches!(state.status, TransactionStatus::Active))

        .map(|(&txn_id, state)| (txn_id, state))

    }

pub fn committed_transactions(&self) -> impl Iterator<Item = TransactionId> + '_ {

    self.transactions.iter()

        .filter(|(_, state)| matches!(state.status, TransactionStatus::Committed))

        .map(|(&txn_id, _)| txn_id)

    }

}

// src/recovery/dirty_page_table.rs

use std::collections::HashMap;

use crate::types::{PageId, LSN};

pub struct DirtyPageTable {

    pages: HashMap<PageId, LSN>,

}

impl DirtyPageTable {

    pub fn new() -> Self {

        Self {

            pages: HashMap::new(),

        }

    }

    pub fn mark_dirty(&mut self, page_id: PageId, lsn: LSN) {


```

```

// Only set if this is the first time we see this page dirty

// (we want the earliest LSN that dirtied it)

self.pages.entry(page_id).or_insert(lsn);

}

pub fn get_recovery_lsn(&self, page_id: PageId) -> Option<LSN> {

self.pages.get(&page_id).copied()

}

pub fn min_recovery_lsn(&self) -> Option<LSN> {

self.pages.values().min().copied()

}

pub fn dirty_pages(&self) -> impl Iterator<Item = (PageId, LSN)> + '_ {

self.pages.iter().map(|(&page_id, &lsn)| (page_id, lsn))

}

}

```

#### D. Core Logic Skeleton Code:

```
// src/recovery/mod.rs

use crate::types::*;

use crate::log::LogReader;

use crate::storage::Storage;

use std::sync::Arc;

pub struct RecoveryManager {

    log_reader: LogReader,

    storage: Arc<dyn Storage>,

    transaction_table: TransactionTable,

    dirty_page_table: DirtyPageTable,

    checkpoint_lsn: Option<LSN>,

}

impl RecoveryManager {

    pub fn new(log_reader: LogReader, storage: Arc<dyn Storage>) -> Self {

        Self {

            log_reader,

            storage,

            transaction_table: TransactionTable::new(),

            dirty_page_table: DirtyPageTable::new(),

            checkpoint_lsn: None,

        }

    }

    /// Main recovery entry point - performs complete ARIES recovery

    pub fn recover(&mut self) -> WalResult<()> {

        // TODO 1: Find the most recent checkpoint record in the log

    }

}
```

```

// TODO 2: Initialize transaction and dirty page tables from checkpoint

// TODO 3: Perform analysis pass from checkpoint to end of log

// TODO 4: Perform redo pass starting from minimum dirty page LSN

// TODO 5: Perform undo pass for all active transactions

// TODO 6: Write recovery completion record to log

todo!("Implement complete recovery process")

}

/// Analysis phase: scan log and build recovery data structures

pub fn analysis_pass(&mut self) -> WalResult<()> {

    // TODO 1: Determine starting LSN (from checkpoint or beginning)

    // TODO 2: Scan forward through all log records from start point

    // TODO 3: For each record, update transaction table with transaction state

    // TODO 4: For redo records, update dirty page table if not already present

    // TODO 5: For commit/abort records, update transaction status

    // TODO 6: Handle log corruption by stopping at first invalid record

    // Hint: Use log_reader.scan_from(start_lsn) to get iterator over records

    todo!("Implement analysis phase")

}

/// Redo phase: replay all committed changes to restore database state

pub fn redo_pass(&mut self) -> WalResult<()> {

    // TODO 1: Find minimum LSN from dirty page table as redo start point

    // TODO 2: If no dirty pages, skip redo entirely

    // TODO 3: Scan forward from redo start LSN to end of log

    // TODO 4: For each redo record, check if transaction was committed

    // TODO 5: Load target page and check if change already applied (LSN comparison)

```

```

// TODO 6: Apply after_image to page and update page LSN

// TODO 7: Write modified page back to storage

// Hint: Skip records where page.lsn >= record.lsn (already applied)

todo!("Implement redo phase")

}

/// Undo phase: roll back all incomplete transactions

pub fn undo_pass(&mut self) -> WalResult<()> {

    // TODO 1: Collect all active transactions from transaction table

    // TODO 2: For each active transaction, start undo chain traversal from last_lsn

    // TODO 3: Follow undo_next_lsn links backward through transaction's log records

    // TODO 4: For each record in undo chain, apply before_image to reverse change

    // TODO 5: Write Compensation Log Record (CLR) for each undo operation

    // TODO 6: When undo chain complete, write abort record for transaction

    // Hint: Process multiple transactions concurrently for better performance

    todo!("Implement undo phase")

}

/// Apply a single redo record to restore committed change

fn apply_redo_record(&mut self, record: &RedoRecord) -> WalResult<()> {

    // TODO 1: Load target page from storage using record.page_id

    // TODO 2: Check if page.lsn >= record.lsn (change already applied)

    // TODO 3: If not applied, write record.after_image to page at record.offset

    // TODO 4: Update page.lsn to record.lsn

    // TODO 5: Write modified page back to storage

    // Hint: Handle case where page doesn't exist (may need allocation)

    todo!("Apply single redo record")
}

```

```

    }

    /// Apply a single undo record to roll back uncommitted change

    fn apply_undo_record(&mut self, record: &UndoRecord) -> WalResult<LSN> {

        // TODO 1: Load target page from storage using record.page_id

        // TODO 2: Write record.before_image to page at record.offset (restore original
        state)

        // TODO 3: Update page LSN appropriately (complex - may not be record.lsn)

        // TODO 4: Write modified page back to storage

        // TODO 5: Generate and write Compensation Log Record (CLR)

        // TODO 6: Return LSN of the CLR for undo chain continuation

        // Hint: CLR generation is critical for recovery restart correctness

        todo!("Apply single undo record and generate CLR")

    }

}

```

## E. Language-Specific Hints:

- **File I/O:** Use `std::fs::File` with `BufReader` for efficient log scanning. Consider `mmap` for large log files but handle the complexity of partial writes at the end.
- **Error Handling:** Define comprehensive `WalError` enum variants for different recovery failure modes. Use `?` operator extensively but add context with `.map_err()` when propagating errors.
- **Collections:** `HashMap` is sufficient for transaction and dirty page tables. Use `BTreeMap` if you need ordered iteration. Consider `FxHashMap` from `rustc-hash` for better performance.
- **Concurrency:** Recovery is typically single-threaded, but you can parallelize redo/undo within phases. Use `Arc<RwLock<>>` for shared page cache access.
- **Memory Management:** Be careful with large after-images and before-images. Consider streaming large records instead of loading entirely into memory.

## F. Milestone Checkpoint:

After implementing Recovery Manager (Milestone 3):

- **Unit Tests:** Run `cargo test recovery::` to verify analysis, redo, and undo phases work independently

- **Integration Test:** Create a test that writes several transactions, crashes mid-way, and verifies recovery restores exactly the committed transactions
- **Expected Behavior:** Recovery should identify committed vs. active transactions, restore all committed changes, and remove all uncommitted changes
- **Performance Check:** Analysis phase should complete in  $O(\log\_size)$ , redo in  $O(\text{dirty\_pages})$ , undo in  $O(\text{active\_transaction\_changes})$
- **Error Handling:** Verify recovery handles corrupted log records gracefully and can restart if interrupted

#### G. Debugging Tips:

Symptom	Likely Cause	How to Diagnose	Fix
Recovery hangs during analysis	Infinite loop in log scanning	Add debug logging for each record processed	Implement proper end-of-log detection
Committed changes lost after recovery	Redo phase not applying changes	Check transaction table - are committed transactions marked correctly?	Fix transaction status tracking in analysis
Uncommitted changes still visible	Undo phase not running or incomplete	Verify active transactions identified in analysis phase	Implement proper undo chain traversal
Recovery crashes with "page not found"	Trying to undo changes to deallocated pages	Add existence check before page operations	Handle missing pages gracefully in undo
CLR generation fails	Insufficient log space or permission issues	Check disk space and log file permissions	Implement log rotation or cleanup
Recovery very slow	Inefficient page access patterns	Profile page load/store operations	Add page caching or batch page operations

## Checkpoint Manager Component

**Milestone(s):** Milestone 4 (Checkpointing) - this section implements fuzzy checkpointing to bound recovery time by creating recovery waypoints without blocking concurrent transactions, enabling log truncation and faster system restart

Think of the `CheckpointManager` as the WAL system's librarian - it periodically creates a comprehensive catalog of the current state, noting which books (database pages) are currently being edited, which patrons (transactions) are still working, and where in the log we can safely discard old records. When the system restarts after a crash, instead of reading the entire log history from the beginning, recovery can jump to the most recent checkpoint and only replay changes from that point forward. The key insight is that checkpoints

must be "fuzzy" - they capture a consistent snapshot of recovery state without freezing the entire system, much like taking a photograph of a busy street where people are still moving.

## Checkpoint Purpose: Why checkpoints are needed and how they speed up recovery

**Checkpoint necessity** emerges from a fundamental problem in Write-Ahead Logging systems: log files grow indefinitely. Without checkpoints, crash recovery would require scanning the entire log from the beginning of time, replaying every transaction that ever committed. As databases run for months or years, this becomes prohibitively expensive. The checkpoint serves as a recovery waypoint - a known-good state from which recovery can begin, dramatically reducing the amount of log that must be processed.

The **recovery time optimization** works by establishing a lower bound on how far back recovery must look. When the `RecoveryManager` starts up after a crash, it locates the most recent checkpoint and begins its analysis pass from that point rather than from LSN 0. This bounded scan principle means that recovery time becomes proportional to the checkpoint interval rather than the total system uptime. A checkpoint taken every 10 minutes ensures recovery completes within minutes, regardless of whether the database has been running for days or years.

Consider the concrete scenario of a database that has been running for 30 days, generating 1 million log records per day. Without checkpoints, crash recovery would scan 30 million records. With daily checkpoints, recovery scans at most 1 million records from the most recent checkpoint. The time savings are dramatic - what might take hours reduces to minutes.

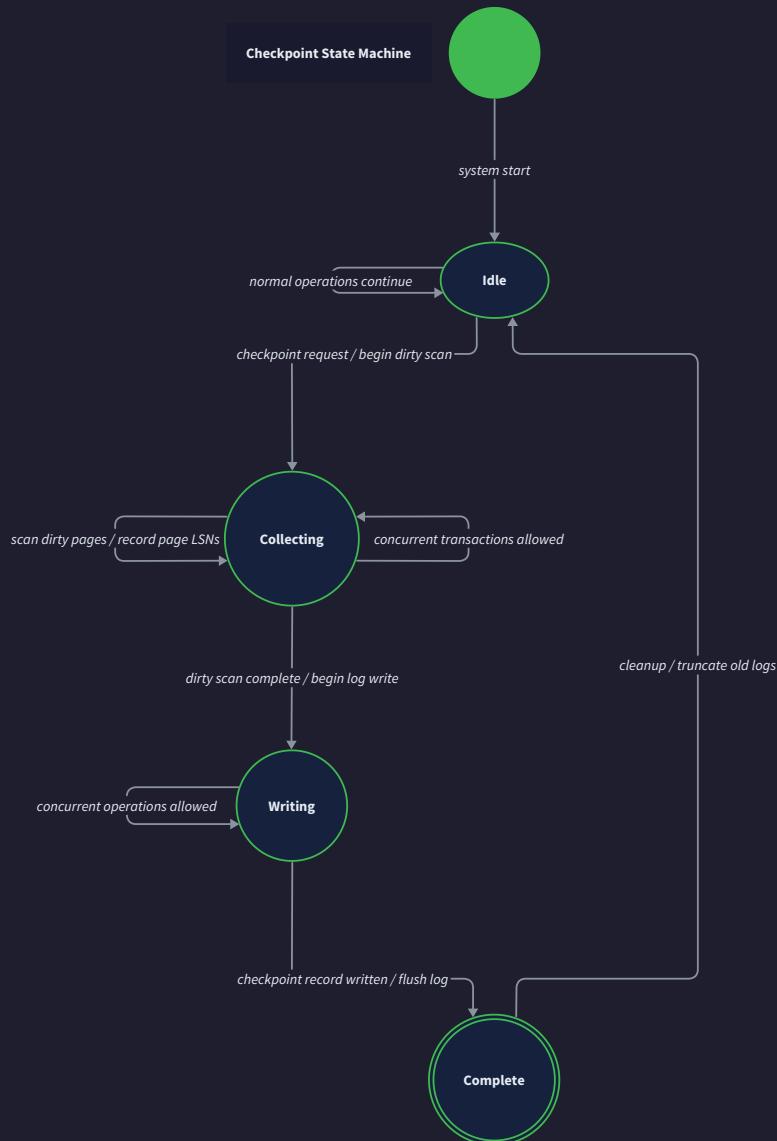
**Storage reclamation** represents the second critical benefit. Log segments older than the most recent checkpoint can be safely deleted because they contain no information needed for recovery. The checkpoint essentially freezes a point-in-time snapshot of which transactions were active and which pages were dirty. Any log records before that point describe either completed transactions (whose effects are already in the database) or aborted transactions (whose effects have been undone). This **log truncation** prevents disk usage from growing without bound.

The checkpoint acts as a recovery firewall - it guarantees that everything before this point is either safely committed to the database or completely undone, so we never need to look further back in history.

However, checkpointing introduces subtle correctness challenges. A naive checkpoint implementation might attempt to create an atomic snapshot by pausing all transactions, recording the current state, and then resuming operations. This **blocking checkpoint** approach guarantees consistency but violates availability requirements - the database becomes unresponsive during checkpoint creation. Modern WAL systems instead implement **fuzzy checkpoints** that maintain correctness while allowing concurrent transactions to proceed.

Checkpoint Type	Consistency Guarantee	Availability Impact	Recovery Complexity
Blocking	Perfect atomic snapshot	System paused during checkpoint	Simple - exact point-in-time state
Fuzzy	Consistent with concurrent activity	No interruption to transactions	Complex - must handle in-progress changes

The **fuzzy checkpoint algorithm** solves the consistency-availability tension by capturing a snapshot that is consistent with respect to recovery semantics, even though the exact system state is changing during checkpoint creation. The key insight is that we don't need an atomic snapshot of the database pages themselves - we need an atomic snapshot of the recovery metadata (active transactions and dirty pages) that tells recovery where to begin its work.



#### Fuzzy Checkpoint Benefits

- **Non-blocking:** Normal operations continue during checkpoint
- **Recovery optimization:** Provides waypoint for faster crash recovery
- **Log truncation:** Enables removal of old log segments
- **Bounded recovery:** Limits replay to post-checkpoint changes

The checkpoint state machine reveals how fuzzy checkpointing avoids blocking normal operations while maintaining correctness guarantees.

## Fuzzy Checkpoint Algorithm: Creating consistent checkpoints without blocking concurrent transactions

The **fuzzy checkpoint algorithm** operates on the principle that recovery metadata can be captured atomically even while the underlying database state continues to evolve. The algorithm creates a consistent

view of which transactions were active and which pages were dirty at a specific logical point in time (identified by an LSN), without requiring those transactions to pause or those pages to stop being modified.

**Conceptual foundation:** Think of the fuzzy checkpoint as taking a photograph of a busy intersection. Cars are constantly moving, but the photograph captures their positions at one specific instant. Similarly, the fuzzy checkpoint captures the recovery state at one specific LSN, even though transactions continue executing and generating new LSNs. The checkpoint record itself gets an LSN that serves as the "timestamp" for the snapshot.

The algorithm proceeds through four distinct phases, each designed to handle the concurrency challenges of capturing consistent state while transactions continue executing:

### Phase 1: Checkpoint Initiation and LSN Assignment

1. The `CheckpointManager` begins a new checkpoint by obtaining the next available LSN from the `LogWriter`. This LSN serves as the logical timestamp for the entire checkpoint operation.
2. It transitions to the `Collecting` state in the checkpoint state machine, signaling that recovery metadata collection is in progress.
3. The manager records the start time and checkpoint LSN for debugging and monitoring purposes.
4. A checkpoint lock is acquired to prevent multiple concurrent checkpoint operations, which could create inconsistent recovery metadata.

### Phase 2: Recovery Metadata Collection

The critical insight of fuzzy checkpointing is that the recovery metadata must reflect the system state at the moment of checkpoint LSN assignment, not at the moment of actual data collection. This temporal consistency is achieved through careful coordination with the transaction management system.

5. The manager queries the transaction management system for all transactions that were active at the checkpoint LSN. This requires the transaction system to maintain historical state briefly.
6. It scans the dirty page tracking system to identify all pages that had uncommitted modifications at the checkpoint LSN. Pages that became dirty after the checkpoint LSN are not included.
7. The collected metadata is validated for internal consistency - every dirty page must be associated with at least one active transaction.
8. Additional recovery hints are computed, such as the minimum LSN from the dirty page table, which bounds how far back the redo phase might need to scan.

### Phase 3: Checkpoint Record Creation and Logging

9. A `CheckpointRecord` is constructed containing the checkpoint LSN, the list of active transaction IDs, and the set of dirty page IDs with their associated LSNs.
10. The checkpoint record is serialized and written to the log using the `LogWriter`'s `write_and_force` method, ensuring immediate durability via `fsync`.

11. The checkpoint's LSN is recorded as the new master checkpoint LSN, making it the starting point for future recovery operations.
12. The state machine transitions to `Writing` during the log write operation, then to `Complete` once the `fsync` completes.

#### Phase 4: Log Truncation and Cleanup

13. With the new checkpoint safely on disk, the manager identifies log segments that precede the previous checkpoint and marks them for truncation.
14. Old segment files are deleted from the filesystem, reclaiming disk space.
15. Internal metadata structures are updated to reflect the new checkpoint, and any temporary state from the checkpoint creation process is cleaned up.

#### Decision: Fuzzy vs. Blocking Checkpoint Strategy

- **Context:** Checkpoints must balance consistency guarantees with system availability during checkpoint creation
- **Options Considered:**
  1. Blocking checkpoints that pause all transactions for atomic snapshots
  2. Fuzzy checkpoints that capture consistent recovery metadata without blocking transactions
  3. Incremental checkpoints that only record changes since the last checkpoint
- **Decision:** Implement fuzzy checkpoints with LSN-based temporal consistency
- **Rationale:** Blocking checkpoints violate availability requirements for long-running systems, while incremental checkpoints add complexity without significant benefits for recovery time. Fuzzy checkpoints provide the optimal balance of consistency, availability, and implementation complexity.
- **Consequences:** Recovery logic must handle the fact that checkpoint data reflects a point-in-time view while database state continued evolving, but this complexity is contained within the recovery algorithm rather than impacting normal operations.

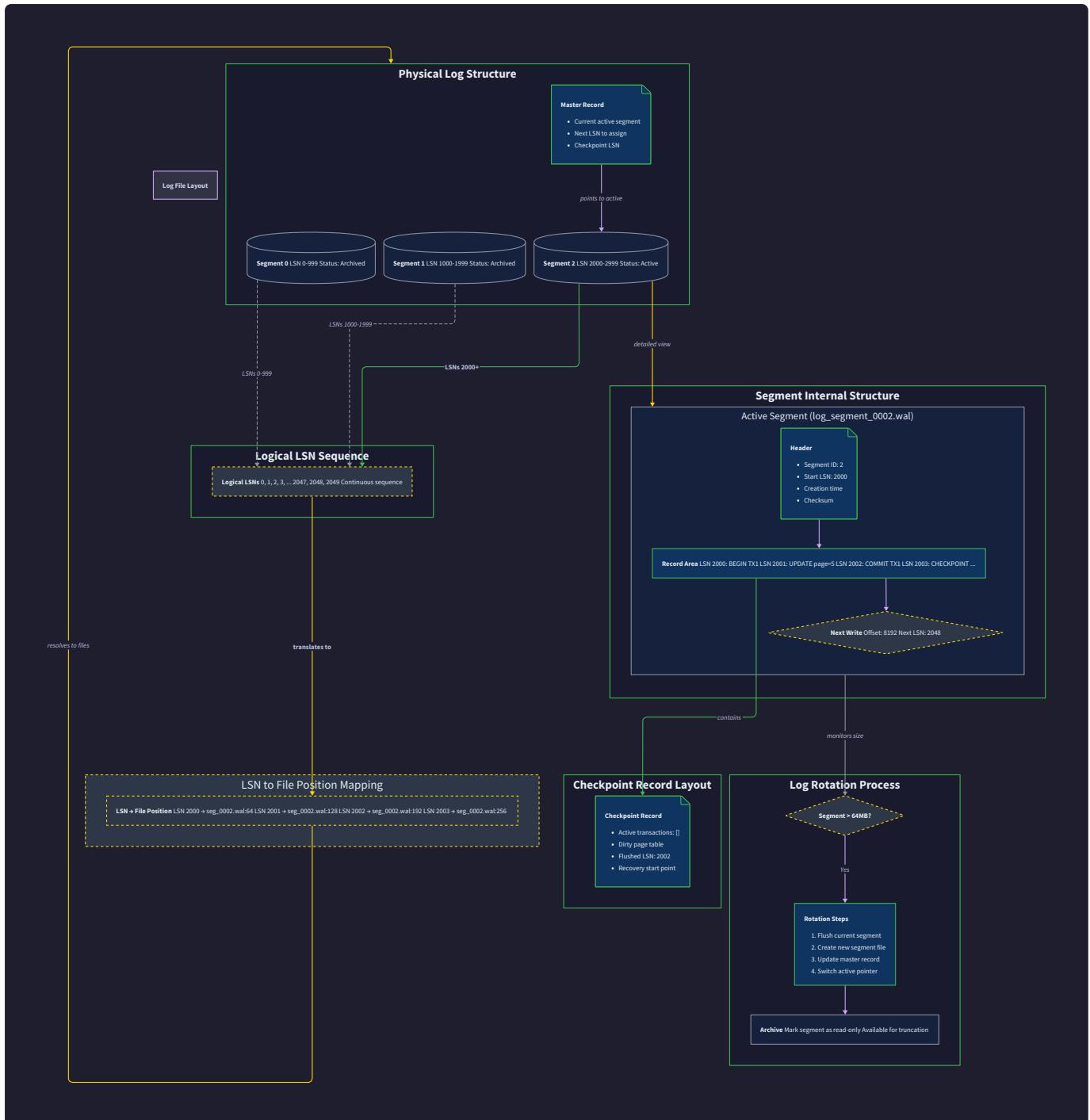
The **concurrency control** aspects of fuzzy checkpointing require careful attention to data races and consistency. The checkpoint operation must coordinate with ongoing transactions to ensure that the captured metadata accurately reflects the system state at the checkpoint LSN:

Concurrent Operation	Potential Race Condition	Resolution Strategy
Transaction commit after checkpoint LSN	Transaction missing from active list	Include all transactions active at checkpoint LSN assignment
Page becomes dirty after checkpoint LSN	Dirty page missing from checkpoint	Only include pages dirty at checkpoint LSN
Transaction aborts during checkpoint	Transaction state inconsistent	Use transaction state at checkpoint LSN
New log record written during checkpoint	LSN ordering confusion	Checkpoint LSN serves as temporal boundary

The **temporal consistency guarantee** ensures that recovery can safely begin from any checkpoint, regardless of when it was created relative to ongoing system activity. This guarantee is achieved by making all checkpoint decisions based on the system state at the checkpoint LSN, creating a logically atomic view of the recovery metadata.

## Log Truncation: Safely removing old log entries after checkpoints

**Log truncation** represents one of the most critical safety operations in WAL systems - it permanently destroys historical information that may be needed for recovery. The fundamental safety principle is that log records can only be truncated after they become unnecessary for all possible recovery scenarios. This determination requires careful analysis of checkpoint contents and recovery algorithm requirements.



The physical structure of log files and checkpoint records determines how truncation operations can safely proceed while maintaining recovery correctness.

The **truncation safety analysis** begins with understanding what information each log record category contributes to recovery. Committed transaction records can be truncated once their effects are guaranteed to be in stable storage. Aborted transaction records can be truncated once all their undo operations are complete and stable. Active transaction records cannot be truncated until those transactions complete.

**Safe truncation conditions** must be verified before any log segment removal:

1. **Checkpoint durability:** The checkpoint record that will serve as the new recovery starting point must be durably committed to storage with `fsync`. Until the checkpoint is durable, the previous checkpoint remains

the authoritative recovery point.

2. **Transaction completion boundary:** All transactions that committed before the checkpoint LSN must have their committed changes fully written to stable storage. This typically means that database pages containing their modifications have been written to disk.
3. **Undo completion verification:** Any transactions that were aborted before the checkpoint LSN must have all their undo operations completed and durably recorded. Partial undo operations require retention of the original log records.
4. **Dirty page analysis:** Pages that were dirty at checkpoint time may still contain uncommitted modifications from transactions that were active at the checkpoint. The log records for these transactions must be retained until the pages are cleaned or the transactions complete.

The **truncation algorithm** operates by identifying the earliest log record that might be needed for recovery from the current checkpoint:

1. **Calculate minimum required LSN:** Start with the checkpoint LSN as a candidate truncation point.
2. **Check active transaction requirements:** For each transaction that was active at checkpoint time, identify its earliest log record that might be needed for undo operations. The minimum of these LSNs bounds how far back truncation can safely proceed.
3. **Analyze dirty page dependencies:** For each page that was dirty at checkpoint time, identify the earliest log record that might need to be replayed to reconstruct the page's committed state. This provides another lower bound on safe truncation.
4. **Compute safe truncation LSN:** Take the minimum of all the bounds computed in previous steps. This LSN represents the earliest log record that must be retained.
5. **Map LSN to segment boundaries:** Log segments are the unit of truncation. Identify all segments that contain only LSNs earlier than the safe truncation point.
6. **Perform atomic truncation:** Delete the identified segments atomically, ensuring that the truncation operation itself cannot leave the log in an inconsistent state.

The golden rule of log truncation: when in doubt, keep the data. It's better to retain unnecessary log records than to delete records that recovery might need.

**Truncation coordination** with ongoing operations requires careful synchronization to prevent race conditions. While truncation is occurring, new log records are being written and new transactions are starting. The truncation process must ensure that it doesn't interfere with these operations:

Concurrent Operation	Potential Conflict	Synchronization Strategy
New log records being written	Truncation deletes active segment	Never truncate segments newer than checkpoint
Transaction starts before truncation completes	Transaction log records deleted	Use LSN-based retention, not time-based
Recovery starts during truncation	Inconsistent log state	Atomic segment deletion
Checkpoint starts during truncation	Metadata inconsistency	Checkpoint-truncation ordering lock

The **error handling during truncation** must account for the fact that truncation errors can leave the system in an inconsistent state. Unlike most WAL operations that can be retried, partial truncation creates a situation where some historical information has been permanently lost:

#### Truncation failure scenarios:

- **Partial segment deletion:** Some segment files are deleted but others remain due to filesystem errors. This can create gaps in the log that confuse recovery.
- **Metadata update failure:** Segment files are deleted but the master record still points to the old checkpoint. Recovery might attempt to read deleted segments.
- **Concurrent access conflicts:** Another process (backup, monitoring) has files open during truncation, preventing deletion.

**Recovery strategies** for truncation failures depend on the specific failure mode:

1. **Pre-truncation validation:** Before beginning deletion, verify that all target segments can be opened and deleted. This prevents partial failures.
2. **Atomic batch deletion:** Use filesystem operations that can atomically delete multiple files or can be rolled back if any deletion fails.
3. **Post-truncation verification:** After deletion, verify that the log remains consistent and that recovery can successfully start from the current checkpoint.
4. **Truncation rollback:** If truncation fails partially, attempt to restore the previous state by updating metadata to reflect only the successfully deleted segments.

## Master Record Management: Tracking the most recent checkpoint for recovery start point

The **master record** serves as the bootstrap mechanism for WAL recovery - it's the first piece of information that the `RecoveryManager` reads when starting up after a crash. Think of it as the table of contents for the entire log system, pointing to the most recent checkpoint and providing the essential metadata needed to begin recovery operations. The master record must be stored in a location that recovery can find reliably, even when the rest of the log system might be in an inconsistent state.

**Master record location strategy** represents a critical design decision that affects both recovery reliability and system performance. The master record cannot be stored within the regular log files because recovery needs to find it before it knows which log files exist or which segments are current. Several approaches are possible:

### Decision: Master Record Storage Location

- **Context:** Recovery needs a reliable way to locate the most recent checkpoint before it can begin processing the log
- **Options Considered:**
  1. Fixed file location outside the log directory structure
  2. Embedded in the header of each log segment file
  3. Replicated across multiple fixed locations for redundancy
- **Decision:** Use a fixed file ( `master_record.wal` ) in the log directory root with atomic update semantics
- **Rationale:** Fixed location provides predictable discovery for recovery startup. Atomic updates prevent corruption during master record writes. Single location simplifies implementation while providing sufficient reliability for most use cases.
- **Consequences:** Master record becomes a single point of failure, but this risk is mitigated by atomic write semantics and the ability to fall back to full log scanning if the master record is corrupted.

**Master record structure** must contain sufficient information for recovery to bootstrap itself without requiring access to other metadata:

Field	Type	Purpose	Critical Path
<code>magic_number</code>	u32	File format validation	Corruption detection
<code>version</code>	u32	Master record format version	Compatibility checking
<code>checkpoint_lsn</code>	LSN	LSN of most recent checkpoint	Recovery starting point
<code>checkpoint_segment</code>	u64	Segment file containing checkpoint	File location optimization
<code>checkpoint_offset</code>	u64	Byte offset of checkpoint within segment	Direct seek to checkpoint
<code>log_directory</code>	PathBuf	Directory containing log segments	Multi-directory support
<code>creation_timestamp</code>	u64	When this master record was created	Debugging and monitoring
<code>checksum</code>	u32	CRC32 of all preceding fields	Integrity verification

The **atomic update protocol** for the master record prevents corruption during checkpoint completion. Since the master record update represents the final step that makes a new checkpoint authoritative, any corruption or partial write would prevent recovery from finding the checkpoint, forcing an expensive full-log scan.

#### Atomic master record update algorithm:

1. **Prepare new master record:** Create a complete `MasterRecord` structure with all fields populated from the newly completed checkpoint.
2. **Calculate integrity checksum:** Compute CRC32 over all fields except the checksum field itself, providing corruption detection capability.
3. **Write to temporary file:** Serialize the master record to a temporary file (`master_record.tmp`) in the same directory as the authoritative master record. This ensures that the write operation cannot corrupt the existing master record.
4. **Force synchronization:** Call `fsync` on the temporary file to ensure it's durably written to storage before proceeding.
5. **Atomic rename operation:** Use the filesystem's atomic rename operation to replace the old master record with the new one. Most filesystems guarantee that rename operations are atomic - they either complete fully or not at all.
6. **Cleanup temporary file:** Remove any temporary files created during the process, though this cleanup is not critical for correctness.

The **master record discovery process** during recovery startup must handle various failure scenarios while providing reliable bootstrap capability:

#### Recovery startup sequence:

1. **Attempt primary master record read:** Try to open and read `master_record.wal` from the configured log directory.
2. **Validate master record integrity:** Check magic number, version compatibility, and CRC32 checksum. Any validation failure indicates corruption.
3. **Locate checkpoint record:** Use the checkpoint segment and offset information to directly seek to the checkpoint record in the log.
4. **Validate checkpoint accessibility:** Verify that the checkpoint record can be read and deserialized successfully.
5. **Begin recovery from checkpoint:** Initialize the recovery process using the checkpoint as the starting point.

**Fallback strategies** handle cases where the master record is missing, corrupted, or points to inaccessible checkpoint data:

Failure Mode	Detection Method	Fallback Strategy	Performance Impact
Master record missing	File not found error	Scan all log segments for most recent checkpoint	High - full log scan required
Master record corrupted	CRC32 mismatch	Scan log segments starting from newest	High - partial log scan
Checkpoint record inaccessible	Cannot read at specified location	Find alternative checkpoint in log	Medium - segment-by-segment search
Log directory inaccessible	Directory read failure	Check alternative log locations	Low - try backup directories

**Master record versioning** supports evolution of the checkpoint and recovery systems over time. As new features are added to checkpointing (such as additional recovery metadata or optimization hints), the master record format may need to expand. The version field enables backward compatibility and graceful handling of format changes:

**Master Record Format Evolution:**

Version 1: Basic checkpoint location (checkpoint\_lsn, segment, offset)  
 Version 2: Added creation timestamp and log directory path  
 Version 3: Added checkpoint validation hash and segment size hints  
 Version 4: Added distributed checkpoint support for multiple log streams

**Performance optimization** opportunities in master record management focus on minimizing the overhead of checkpoint completion and recovery startup:

- **Write coalescing:** Batch multiple checkpoint completions to reduce fsync frequency when checkpoints are created rapidly
- **Read caching:** Cache master record contents in memory to avoid repeated file system access during normal operations
- **Parallel validation:** Validate master record integrity concurrently with other recovery startup tasks
- **Predictive prefetching:** Use master record information to begin prefetching checkpoint and log segment data before it's needed

**Common Pitfalls in Master Record Management:**

⚠ **Pitfall: Non-atomic master record updates** Directly writing to the master record file without using atomic rename can leave the master record in a partially updated state if the system crashes during the write operation. This forces expensive fallback to full log scanning. **Fix:** Always write to a temporary file and use atomic rename to replace the authoritative master record.

⚠ **Pitfall: Missing fsync before rename** Performing the atomic rename operation before the temporary file is durably written to storage can result in the master record pointing to a checkpoint that wasn't actually

written, breaking recovery. **Fix:** Always fsync the temporary file before renaming it to replace the master record.

**⚠ Pitfall: Inadequate error handling for master record corruption** Failing to implement robust fallback strategies when the master record is corrupted can render the entire database unrecoverable, even when the underlying log data is intact. **Fix:** Implement comprehensive fallback logic that can reconstruct recovery starting points from the log data itself.

## Implementation Guidance

The `CheckpointManager` bridges the gap between high-level checkpointing concepts and practical implementation concerns. Junior developers should focus on understanding the state machine transitions and the coordination between checkpoint creation and log truncation.

### A. Technology Recommendations:

Component	Simple Option	Advanced Option
File I/O	<code>std::fs::File</code> with manual fsync	<code>tokio::fs::File</code> with async I/O
Serialization	<code>serde</code> with <code>bincode</code> format	Custom binary protocol with versioning
Concurrency	<code>std::sync::Mutex</code> for checkpoint state	<code>parking_lot::RwLock</code> for reader-writer access
Error Handling	<code>anyhow</code> for simple error propagation	Custom error types with context
Testing	Unit tests with mock file system	Property-based testing with <code>proptest</code>

### B. Recommended File/Module Structure:

```
project-root/
  src/
    checkpoint/
      mod.rs           ← Public interface exports
      manager.rs       ← CheckpointManager implementation
      master_record.rs ← Master record serialization
      truncation.rs   ← Log truncation logic
      state_machine.rs ← Checkpoint state tracking
      tests/
        integration_tests.rs ← End-to-end checkpoint scenarios
        truncation_tests.rs  ← Log truncation safety tests
    wal/
      mod.rs           ← WAL system integration
      writer.rs        ← LogWriter integration
    recovery/
      mod.rs           ← Recovery system integration
```

**C. Infrastructure Starter Code:**

```
// checkpoint/state_machine.rs - Complete state tracking implementation

use std::sync::{Arc, Mutex};

use std::time::Instant;

#[derive(Debug, Clone, PartialEq)]

pub enum CheckpointState {

    Idle,

    Collecting { start_time: Instant, checkpoint_lsn: LSN },

    Writing { start_time: Instant, checkpoint_lsn: LSN },

    Complete { checkpoint_lsn: LSN },

}

pub struct CheckpointStateMachine {

    state: Arc<Mutex<CheckpointState>>,

}

impl CheckpointStateMachine {

    pub fn new() -> Self {

        Self {

            state: Arc::new(Mutex::new(CheckpointState::Idle)),

        }

    }

    pub fn begin_checkpoint(&self, checkpoint_lsn: LSN) -> bool {

        let mut state = self.state.lock().unwrap();

        match *state {

            CheckpointState::Idle => {

                *state = CheckpointState::Collecting {
```

```
        start_time: Instant::now(),
        checkpoint_lsn,
    };
    true
}

_ => false, // Checkpoint already in progress
}

}

pub fn transition_to_writing(&self) -> Option<LSN> {
    let mut state = self.state.lock().unwrap();
    match *state {
        CheckpointState::Collecting { checkpoint_lsn, .. } => {
            *state = CheckpointState::Writing {
                start_time: Instant::now(),
                checkpoint_lsn,
            };
            Some(checkpoint_lsn)
        }
        _ => None,
    }
}

pub fn complete_checkpoint(&self) -> Option<LSN> {
    let mut state = self.state.lock().unwrap();
    match *state {
        CheckpointState::Writing { checkpoint_lsn, .. } => {

```

```
        *state = CheckpointState::Complete { checkpoint_lsn };

        Some(checkpoint_lsn)

    }

    _ => None,
}

}

pub fn current_state(&self) -> CheckpointState {

    self.state.lock().unwrap().clone()

}

}

// checkpoint/master_record.rs - Complete master record implementation

use serde::{Deserialize, Serialize};

use std::path::PathBuf;

const MASTER_RECORD_MAGIC: u32 = 0xCHEC_KPNT;

const MASTER_RECORD_VERSION: u32 = 1;

#[derive(Debug, Clone, Serialize, Deserialize)]

pub struct MasterRecord {

    magic_number: u32,

    version: u32,

    checkpoint_lsn: LSN,

    checkpoint_segment: u64,

    checkpoint_offset: u64,

    log_directory: PathBuf,

    creation_timestamp: u64,
```

```
        checksum: u32,  
    }  
  
impl MasterRecord {  
  
    pub fn new(  
        checkpoint_lsn: LSN,  
        checkpoint_segment: u64,  
        checkpoint_offset: u64,  
        log_directory: PathBuf,  
    ) -> Self {  
  
        let mut record = Self {  
  
            magic_number: MASTER_RECORD_MAGIC,  
            version: MASTER_RECORD_VERSION,  
            checkpoint_lsn,  
            checkpoint_segment,  
            checkpoint_offset,  
            log_directory,  
            creation_timestamp: std::time::SystemTime::now()  
                .duration_since(std::time::UNIX_EPOCH)  
                .unwrap()  
                .as_secs(),  
            checksum: 0,  
        };  
  
        record.checksum = record.calculate_checksum();  
        record  
    }  
}
```

```
fn calculate_checksum(&self) -> u32 {
    let mut hasher = crc32fast::Hasher::new();
    hasher.update(&self.magic_number.to_le_bytes());
    hasher.update(&self.version.to_le_bytes());
    hasher.update(&self.checkpoint_lsn.to_le_bytes());
    hasher.update(&self.checkpoint_segment.to_le_bytes());
    hasher.update(&self.checkpoint_offset.to_le_bytes());
    hasher.update(self.log_directory.to_string_lossy().as_bytes());
    hasher.update(&self.creation_timestamp.to_le_bytes());
    hasher.finalize()
}

pub fn validate(&self) -> WalResult<()> {
    if self.magic_number != MASTER_RECORD_MAGIC {
        return Err(WalError::Corruption);
    }

    if self.version != MASTER_RECORD_VERSION {
        return Err(WalError::InvalidRecord);
    }

    let expected_checksum = self.calculate_checksum();
    if self.checksum != expected_checksum {
        return Err(WalError::Corruption);
    }

    Ok(())
}
```

```
pub fn checkpoint_lsn(&self) -> LSN { self.checkpoint_lsn }

pub fn checkpoint_location(&self) -> (u64, u64) {
    (self.checkpoint_segment, self.checkpoint_offset)
}

}
```

#### D. Core Logic Skeleton Code:

```
// checkpoint/manager.rs - Core CheckpointManager implementation skeleton

use std::sync::Arc;

use std::collections::HashMap;

use std::path::PathBuf;

pub struct CheckpointManager {

    log_writer: Arc<LogWriter>,

    storage: Arc<dyn Storage>,

    state_machine: CheckpointStateMachine,

    log_directory: PathBuf,

    current_checkpoint_lsn: Option<LSN>,

}

impl CheckpointManager {

    pub fn new(

        log_writer: Arc<LogWriter>,

        storage: Arc<dyn Storage>,

        log_directory: PathBuf,

    ) -> Self {

        // TODO: Initialize CheckpointManager with all required components

        // TODO: Load existing master record if it exists

        // TODO: Set up state machine in Idle state

    }

    /// Creates a fuzzy checkpoint without blocking concurrent transactions

    pub fn create_checkpoint(

        &mut self,

        active_transactions: Vec<TransactionId>,

    )
}
```

```
dirty_pages: Vec<PageId>,

) -> WalResult<LSN> {

    // TODO 1: Begin checkpoint operation using state machine

    // TODO 2: Assign checkpoint LSN as logical timestamp for this checkpoint

    // TODO 3: Collect recovery metadata (active transactions, dirty pages)

    // TODO 4: Create CheckpointRecord with collected metadata

    // TODO 5: Write checkpoint record to log with forced sync

    // TODO 6: Update master record atomically

    // TODO 7: Transition state machine to Complete

    // TODO 8: Return the checkpoint LSN for caller reference

    // Hint: Use state machine to prevent concurrent checkpoints

    // Hint: Checkpoint LSN serves as temporal consistency boundary

}

/// Safely truncates log segments older than the current checkpoint

pub fn truncate_log(&mut self) -> WalResult<()> {

    // TODO 1: Verify that current checkpoint is durable on disk

    // TODO 2: Calculate minimum LSN that must be retained for recovery

    // TODO 3: Identify log segments that only contain records older than min LSN

    // TODO 4: Check that no active operations are using segments to be deleted

    // TODO 5: Atomically delete obsolete segment files

    // TODO 6: Update internal metadata to reflect truncated segments

    // Hint: Never truncate segments newer than current checkpoint

    // Hint: Account for active transactions that might need old records

}

/// Updates master record with atomic write semantics
```

```
fn update_master_record(&self, checkpoint_lsn: LSN) -> WalResult<()> {

    // TODO 1: Create new MasterRecord with checkpoint information

    // TODO 2: Serialize master record to binary format

    // TODO 3: Write to temporary file in log directory

    // TODO 4: Force sync temporary file to ensure durability

    // TODO 5: Atomically rename temporary file to master_record.wal

    // TODO 6: Clean up any temporary files from the operation

    // Hint: Use atomic rename to prevent master record corruption

    // Hint: Calculate CRC32 checksum for integrity verification

}
```

```
/// Loads master record during CheckpointManager initialization

pub fn load_master_record(&mut self) -> WalResult<Option<MasterRecord>> {

    // TODO 1: Check if master_record.wal exists in log directory

    // TODO 2: Read and deserialize master record if present

    // TODO 3: Validate master record integrity using checksum

    // TODO 4: Verify that referenced checkpoint record is accessible

    // TODO 5: Return master record or None if not found/invalid

    // Hint: Implement fallback to log scanning if master record is corrupted

    // Hint: Use MasterRecord::validate() method for integrity checking

}
```

```
/// Determines safe truncation point based on recovery requirements

fn calculate_safe_truncation_lsn(
    &self,
    checkpoint_lsn: LSN,
    active_transactions: &[TransactionId],
```

```

) -> LSN {
    // TODO 1: Start with checkpoint LSN as candidate truncation point

    // TODO 2: For each active transaction, find its earliest log record

    // TODO 3: Take minimum of checkpoint LSN and earliest active transaction LSN

    // TODO 4: Account for any other recovery requirements (dirty pages, etc.)

    // TODO 5: Return the earliest LSN that must be retained

    // Hint: Conservative approach - when in doubt, retain more log records

    // Hint: Consider undo requirements for active transactions

}

}

```

## E. Language-Specific Hints:

- **File Operations:** Use `std::fs::rename()` for atomic master record updates, and `File::sync_all()` for durability guarantees
- **Serialization:** `bincode` provides efficient binary serialization for checkpoint records with `serde` derive macros
- **Error Handling:** Create specific `WalError` variants for checkpoint failures: `CheckpointInProgress`, `TruncationFailed`, `MasterRecordCorrupted`
- **Concurrency:** Use `Arc<Mutex<T>>` for shared checkpoint state, but avoid holding locks during I/O operations to prevent blocking
- **Testing:** Use `tempfile::TempDir` for isolated test environments and `std::sync::atomic::Ordering::SeqCst` for test synchronization

## F. Milestone Checkpoint:

After implementing the checkpoint manager:

1. **Run the test suite:** `cargo test checkpoint` should pass all basic checkpoint creation tests
2. **Manual verification:** Create a checkpoint, restart the system, verify recovery starts from checkpoint rather than beginning of log
3. **Truncation testing:** After several checkpoints, verify that old log segments are removed and disk usage remains bounded
4. **Crash simulation:** Kill the process during checkpoint creation, verify the system recovers correctly on restart

5. **Expected behavior:** Checkpoint creation should complete within seconds without blocking concurrent transactions, and recovery time should remain constant regardless of total log size

### Signs of implementation issues:

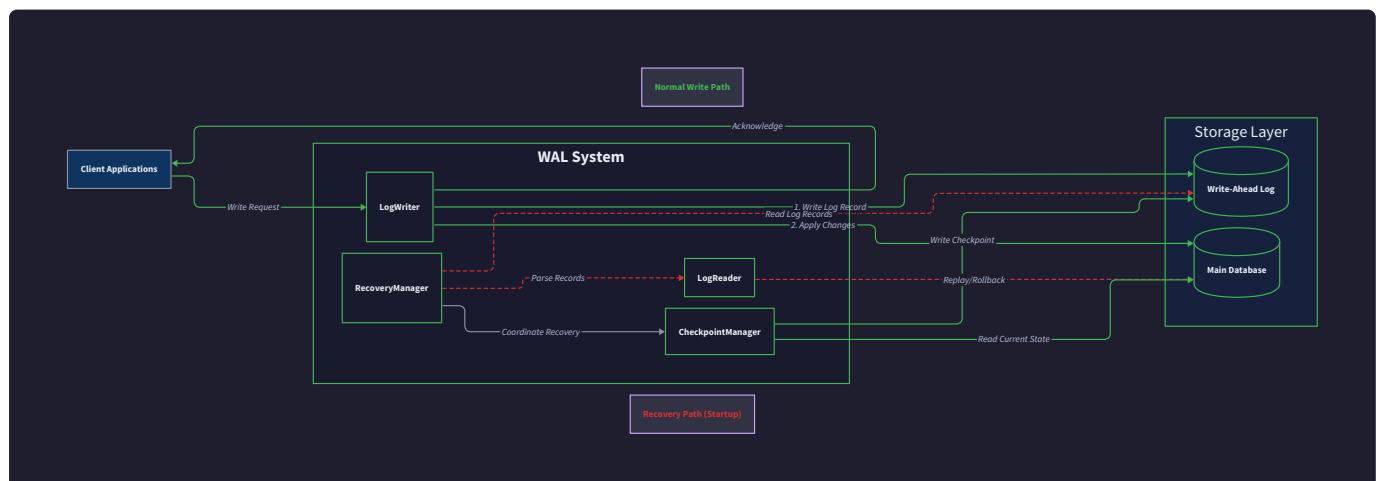
- Recovery always scans from LSN 0 (master record not being updated)
- Crashes during checkpoint creation (non-atomic master record updates)
- Log files growing without bound (truncation not working)
- Concurrent transactions blocked during checkpoints (not using fuzzy algorithm)

## Interactions and Data Flow

**Milestone(s):** All milestones - this section demonstrates how components from Milestone 1 (Log Record Format), Milestone 2 (Log Writer), Milestone 3 (Crash Recovery), and Milestone 4 (Checkpointing) work together to provide complete WAL functionality

Think of a WAL system as a **sophisticated film production studio**. During normal operations, it's like filming a movie where every scene (transaction) must be recorded to tape (log) before the actors perform on the live stage (database). When a power outage crashes the studio, recovery is like having the film editor review all the recorded footage to recreate exactly what happened: completed scenes get applied to the final cut, while interrupted scenes get discarded entirely. Meanwhile, checkpointing is like creating periodic "director's cuts" that allow editors to start reviewing from recent major milestones rather than watching the entire film from the beginning.

This section explores the intricate choreography between the four main components—`LogWriter`, `LogReader`, `RecoveryManager`, and `CheckpointManager`—as they collaborate to ensure durability and consistency. Understanding these interactions is crucial because the **correctness of the entire WAL system depends on precise coordination** between these components during both normal operations and crash recovery scenarios.

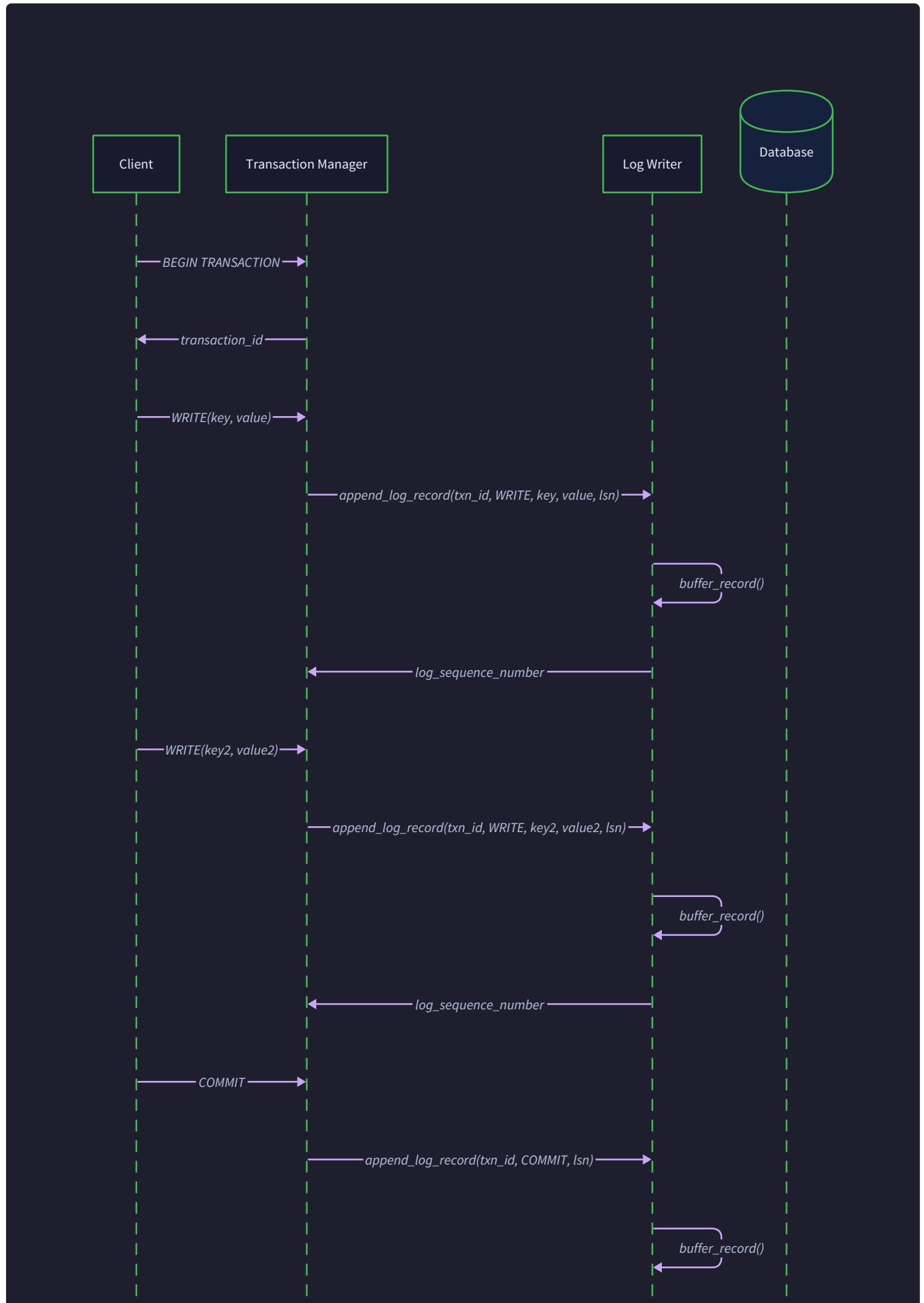


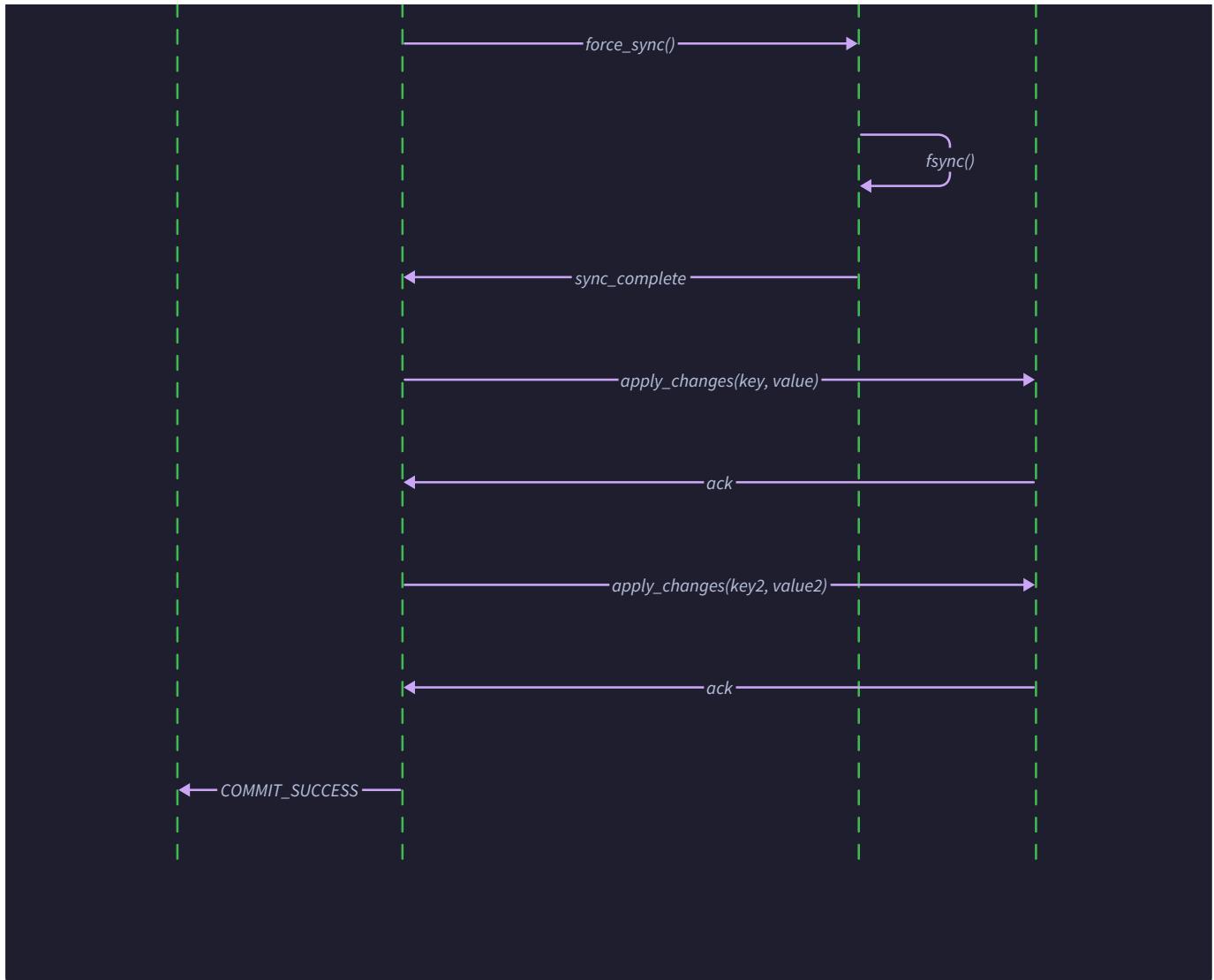
The complexity arises from the fundamental tension between **performance and safety**. Normal operations prioritize minimizing latency while ensuring durability, requiring careful coordination of buffering, group commits, and selective fsync calls. Recovery operations prioritize correctness over speed, requiring systematic reconstruction of system state through careful analysis of potentially corrupted or incomplete log data. Checkpoint operations must balance both concerns, creating consistent recovery waypoints without disrupting ongoing transactions.

## Write Transaction Flow

The write transaction flow represents the **critical path for database durability**—every step must execute correctly or the system risks data loss. This flow demonstrates how WAL transforms potentially unsafe direct database writes into a sequence of safe, recoverable operations.

Think of this flow as a **bank's check-clearing process**. When you write a check, the bank doesn't immediately move money between accounts. Instead, it records your intent in a ledger (log), verifies the transaction is valid, makes the ledger entry permanent (fsync), and only then updates the account balances (database). If anything goes wrong before the ledger entry is permanent, the transaction can be safely ignored. If something goes wrong after, the ledger can be replayed to recreate the account updates.





The complete write transaction flow involves eight distinct phases, each with specific responsibilities and failure handling requirements:

### Phase 1: Transaction Initialization

1. **Client submits transaction request** containing one or more database operations (INSERT, UPDATE, or DELETE statements with target pages, offsets, and data)
2. **Transaction coordinator generates unique TransactionId** using monotonically increasing counter combined with timestamp to ensure global uniqueness across restarts
3. **Coordinator allocates transaction state** in the `TransactionTable` with status `TransactionStatus::Active` and empty `undo_next_lsn` chain
4. **Memory pre-allocation occurs** where `BufferPool` reserves space for log records to avoid allocation failures during critical commit path
5. **Lock acquisition begins** where necessary page-level locks are obtained to prevent concurrent modifications during the transaction

## Phase 2: Redo Record Generation

6. **Before-image capture** where current data values are read from target database pages before any modifications occur
7. **After-image computation** where new data values are calculated by applying the requested operations to the before-images
8. **RedoRecord construction** containing `lsn` (from `LogWriter::next_lsn`), `txn_id`, `page_id`, `offset`, and `after_image` fields
9. **UndoRecord construction** containing corresponding `before_image` data and backward chaining `undo_next_lsn` pointer to previous undo record
10. **CRC checksum calculation** for both records using `calculate_crc32()` to enable corruption detection during recovery

## Phase 3: Log Record Writing

11. **Buffer space allocation** where `LogWriter` obtains appropriately-sized buffer from `BufferPool` based on serialized record size
12. **Record serialization** where both redo and undo records are converted to binary format using `serialize()` method with proper field alignment
13. **Atomic append operation** where serialized records are written to current `LogSegment` using `append()` method that returns file offset
14. **LSN assignment and linking** where each record receives its final LSN and transaction state is updated with `update_lsn()` to maintain undo chain
15. **Segment rotation check** where `LogWriter` evaluates `is_full()` and calls `rotate_segment()` if current segment exceeds `DEFAULT_SEGMENT_SIZE`

## Phase 4: Durability Enforcement

16. **Group commit optimization** where `LogWriter` batches multiple transaction log records in the buffer pool before forcing to disk
17. **Force write execution** where `write_and_force()` calls `fsync()` on the current segment to ensure log records reach persistent storage
18. **Durability verification** where successful fsync completion guarantees that log records will survive system crashes and power failures
19. **Transaction state advancement** where `TransactionTable` marks the transaction as ready for database application

20. **Buffer pool cleanup** where temporary buffers are returned to appropriate size pools for reuse by subsequent transactions

## Phase 5: Database Application

21. **Page retrieval** where target database pages are loaded into memory using `Storage::get_page()` with their current LSN values
22. **LSN comparison check** where page LSN is compared against redo record LSN to ensure idempotent application during recovery replay
23. **Data modification** where `after_image` data is applied to the target page using `write_data()` method with proper offset calculation
24. **Page LSN update** where modified page LSN is set to the redo record LSN to track the most recent change applied
25. **Dirty page tracking** where `DirtyPageTable::mark_dirty()` records the page as containing uncommitted changes for checkpoint coordination

## Phase 6: Commit Processing

26. **Commit record creation** where `CommitRecord` is constructed with transaction ID and final LSN for the transaction
27. **Final log write** where commit record is serialized and written to log using `write_and_force()` to guarantee commitment durability
28. **Transaction table cleanup** where `TransactionTable` updates transaction status to `TransactionStatus::Committed`
29. **Lock release** where all page-level locks held by the transaction are released to allow other transactions to proceed
30. **Client notification** where successful commit response is sent to client, guaranteeing that changes will survive crashes

## Phase 7: Background Cleanup

31. **Buffer pool maintenance** where completed transaction buffers are returned to appropriate pools and memory usage is monitored
32. **Dirty page management** where pages modified by committed transactions remain tracked until the next checkpoint operation
33. **Undo chain finalization** where transaction's undo record chain is preserved for potential recovery scenarios
34. **Segment monitoring** where log segments are monitored for size thresholds and rotation opportunities
35. **Checkpoint triggering** where transaction completion may trigger automatic checkpoint if sufficient dirty pages or time has elapsed

**Key Insight:** The write transaction flow demonstrates the fundamental WAL principle: **log records must be durably written before database changes are applied**. This ordering ensures that recovery can always reconstruct committed changes, even if the database itself is corrupted.

### Write Transaction Flow State Transitions:

Current Phase	Event	Next Phase	Critical Actions
Initialization	Client Request	Redo Generation	Allocate TransactionId, lock resources
Redo Generation	Data Ready	Log Writing	Create redo/undo records, compute checksums
Log Writing	Records Serialized	Durability	Append to segment, assign LSNs
Durability	fsync Complete	DB Application	Log records guaranteed persistent
DB Application	Pages Modified	Commit Processing	Apply after-images, update page LSNs
Commit Processing	Commit Logged	Cleanup	Write commit record, release locks
Cleanup	Resources Released	Complete	Return buffers, update dirty page table

### Decision: Group Commit Optimization Strategy

- **Context:** Individual fsync calls for each transaction create significant I/O bottleneck, limiting system throughput to disk sync speed
- **Options Considered:**
  - Immediate fsync per transaction (simple but slow)
  - Batched commits with timeout-based flushing (complex but fast)
  - Adaptive batching based on load (most complex, optimal performance)
- **Decision:** Implement timeout-based group commit with 10ms maximum delay
- **Rationale:** Balances latency (10ms worst case) with throughput (100x improvement under load) while keeping implementation complexity manageable
- **Consequences:** Requires careful timeout management and adds complexity to shutdown procedures, but dramatically improves performance under concurrent load

### Common Transaction Flow Pitfalls

**⚠ Pitfall: Applying Database Changes Before Log Durability** Applying after-images to database pages before the log records are durably written (fsync'd) violates the fundamental WAL protocol. If a crash occurs

after database modification but before log durability, recovery cannot reconstruct the change because no log record exists. Always ensure `write_and_force()` completes successfully before any `write_data()` calls.

**⚠ Pitfall: Incorrect LSN Ordering in Multi-Operation Transactions** Assigning LSNs to redo/undo record pairs independently can create inconsistent ordering where a transaction's undo record has a higher LSN than its corresponding redo record. This breaks recovery's assumption that undo chains are traversed in reverse LSN order. Always assign LSNs sequentially within each transaction and maintain proper undo chain linking with `undo_next_lsn` pointers.

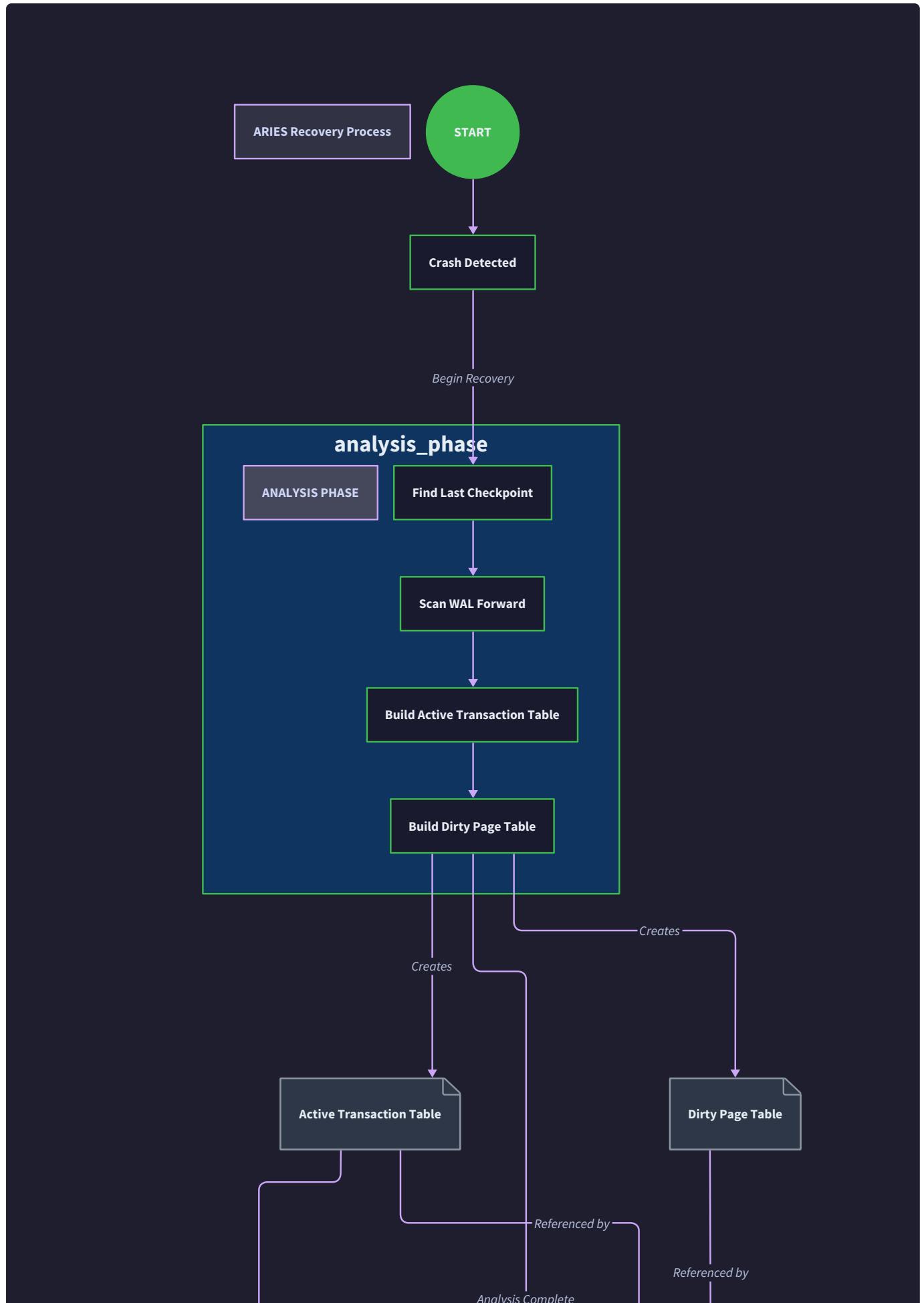
**⚠ Pitfall: Buffer Pool Exhaustion During Critical Path** Attempting to allocate buffers from `BufferPool` during log record serialization can fail when memory is exhausted, leaving the transaction in an inconsistent state with locks held but no way to proceed or rollback. Pre-allocate all required buffers during transaction initialization, or implement a reserved buffer pool for rollback operations.

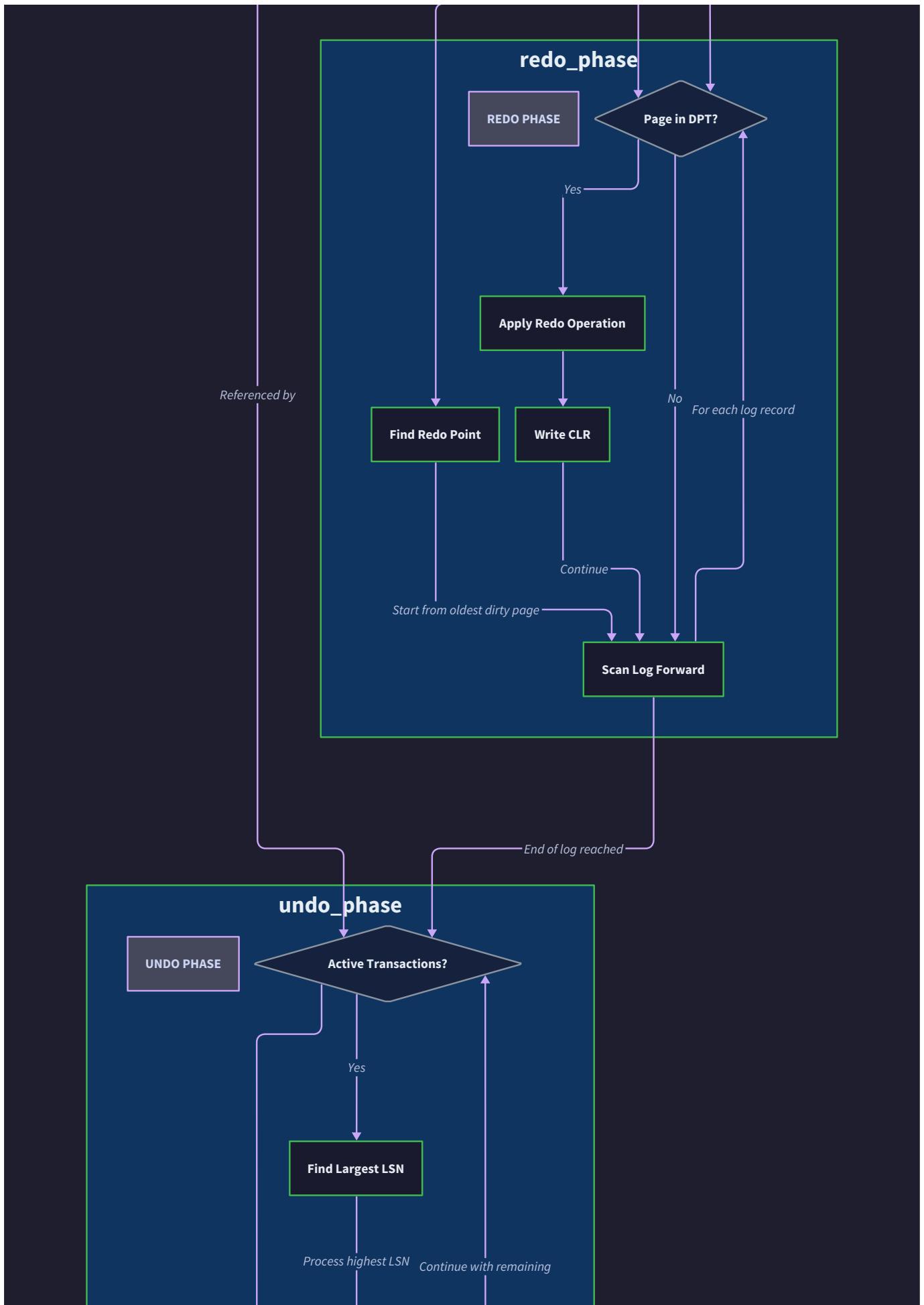
**⚠ Pitfall: Segment Rotation During Transaction Commit** Allowing `rotate_segment()` to occur between writing redo records and the commit record can split a single transaction across multiple segments, complicating recovery scanning. Either complete entire transactions within single segments, or ensure recovery can correctly handle cross-segment transaction reconstruction.

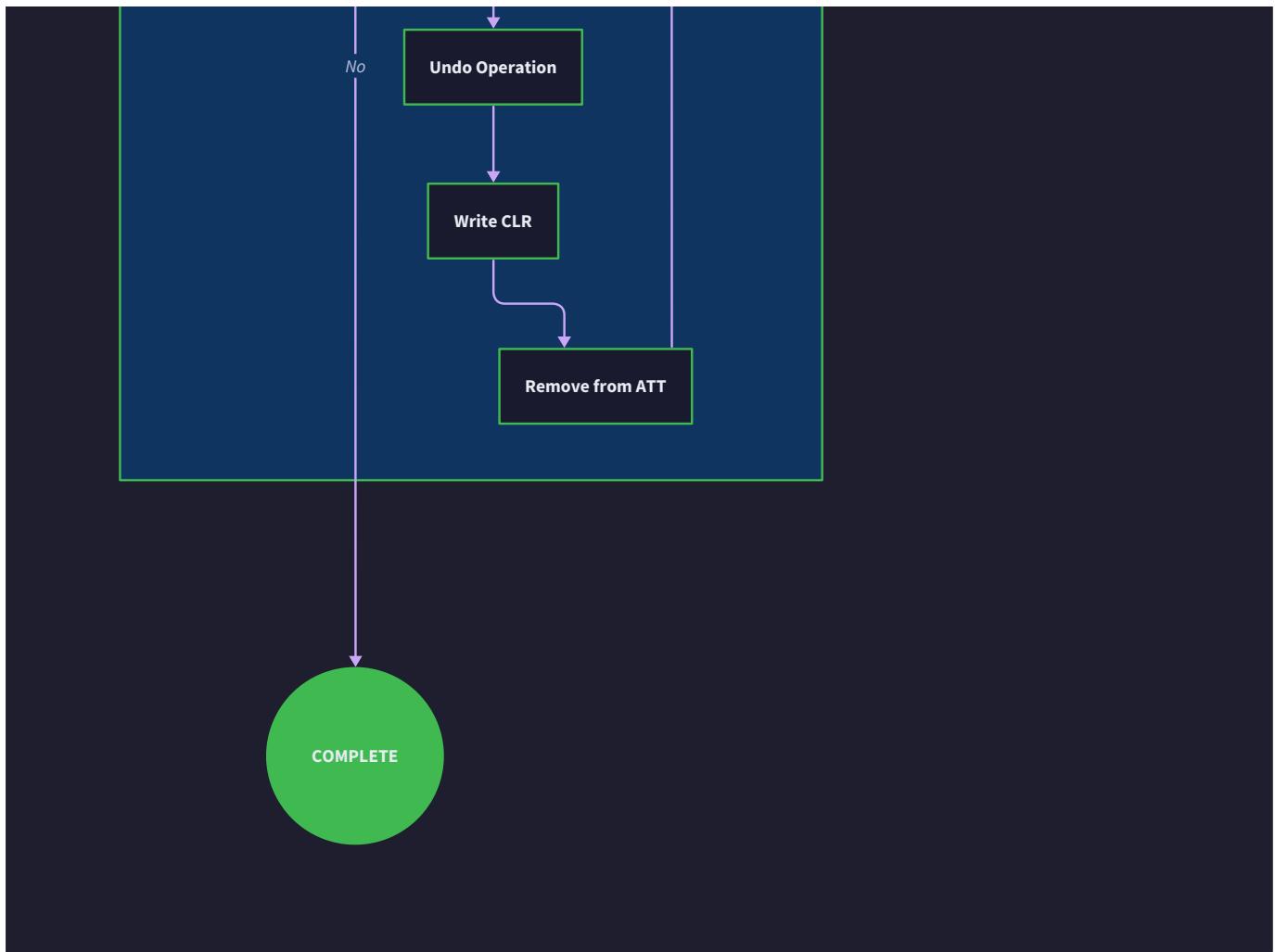
## Recovery Sequence

The recovery sequence represents the **most critical correctness requirement** of the entire WAL system. When a crash occurs, recovery must rebuild a consistent database state from potentially inconsistent log data, handling scenarios where transactions were interrupted at any point in their lifecycle.

Think of recovery as a **crime scene investigation**. The detective (recovery manager) arrives at a scene (crashed database) where events were interrupted mid-action. Using evidence (log records), the detective must reconstruct exactly what happened: which activities were completed (committed transactions), which were abandoned (aborted transactions), and which were interrupted (active transactions). The detective then must undo any half-finished actions and ensure all completed actions are properly reflected in the final state.







The ARIES recovery algorithm consists of three distinct phases, each with specific goals and invariant requirements. The **critical insight** is that recovery must be **idempotent**—running recovery multiple times on the same database state must produce identical results.

## Pre-Recovery System State Assessment

Before beginning the formal ARIES phases, the `RecoveryManager` must establish the recovery starting point and validate system integrity:

1. **Master record loading** where `load_master_record()` attempts to locate the most recent checkpoint information
2. **Checkpoint validation** where loaded `MasterRecord` is validated using `validate()` to ensure integrity and detect corruption
3. **Log segment discovery** where recovery scans the log directory to identify all available segments and their LSN ranges
4. **Starting LSN determination** where recovery calculates the earliest LSN that must be processed based on checkpoint information
5. **Log reader initialization** where `LogReader` is positioned at the recovery starting point, ready to scan forward through log records

## Analysis Phase: Building Recovery State

The Analysis phase reconstructs the system state at the time of crash by scanning log records and building the data structures necessary for Redo and Undo phases.

6. **Transaction table initialization** where `TransactionTable` is created empty, ready to track all discovered transactions
7. **Dirty page table initialization** where `DirtyPageTable` is cleared to track pages that require redo processing
8. **Log scanning commencement** where `LogReader` begins sequential scan from checkpoint LSN to end of log
9. **Record-by-record processing** where each log record is deserialized using `deserialize()` and processed according to its type
10. **Transaction state tracking** where `RedoRecord` and `UndoRecord` entries update transaction states using `get_or_create()` and `update_lsn()`
11. **Dirty page discovery** where `RedoRecord` entries cause `mark_dirty()` calls to track pages requiring redo
12. **Commit/abort processing** where `CommitRecord` and `AbortRecord` entries update transaction status to `Committed` or `Aborted`
13. **Active transaction identification** where transactions without commit/abort records remain with status `Active`
14. **Checkpoint record integration** where `CheckpointRecord` entries provide additional transaction and dirty page information
15. **Recovery LSN calculation** where `min_recovery_lsn()` determines the earliest LSN requiring redo processing

**Analysis Phase Results:** At completion, `TransactionTable` contains complete transaction state information, and `DirtyPageTable` identifies all pages that may contain uncommitted changes or require redo processing.

## Redo Phase: Replaying Committed Changes

The Redo phase ensures that all committed transaction effects are present in the database, handling cases where crashes occurred before database pages were updated.

16. **Redo starting point** where recovery begins processing from `min_recovery_lsn()` to ensure all necessary changes are considered
17. **Forward log traversal** where `LogReader` scans forward from recovery LSN to end of log, processing only `RedoRecord` entries

18. **Page-level LSN checking** where each redo record's target page is loaded and its LSN compared to the redo record's LSN
19. **Idempotent redo application** where `apply_redo_record()` applies changes only if page LSN is less than record LSN
20. **After-image restoration** where redo record's `after_image` data is written to the target page at the specified offset
21. **Page LSN advancement** where successfully applied redo operations update the page LSN to match the record LSN
22. **Dirty page status updates** where pages receiving redo operations remain in `DirtyPageTable` until final cleanup
23. **Transaction filtering** where redo records from aborted transactions are skipped to avoid applying uncommitted changes
24. **Database consistency restoration** where all committed transaction effects are guaranteed present in database pages
25. **Redo phase completion** where database reflects all committed changes, regardless of crash timing

**Redo Phase Guarantee:** After redo completion, the database contains all effects from committed transactions, bringing it to a consistent state as of the crash time.

## Undo Phase: Rolling Back Incomplete Transactions

The Undo phase removes effects of transactions that were active at crash time, ensuring that only committed changes remain in the database.

26. **Active transaction identification** where `active_transactions()` iterator provides all transactions requiring rollback
27. **Undo chain traversal** where each active transaction is processed by following its `undo_next_lsn` chain backwards
28. **Compensation Log Record generation** where each undo operation generates a CLR to ensure recovery idempotency
29. **Before-image restoration** where `apply_undo_record()` writes `before_image` data back to target pages
30. **Undo LSN advancement** where transaction undo chains are traversed using `undo_next_lsn` pointers
31. **Page LSN management** where pages receiving undo operations have their LSNs updated to reflect CLR application
32. **Transaction cleanup** where completed rollbacks update `TransactionTable` status to `Aborted`
33. **Dirty page maintenance** where pages modified during undo remain tracked until final cleanup
34. **Rollback completion** where all active transactions are fully rolled back to their pre-transaction state
35. **Recovery finalization** where system state is marked consistent and ready for new transactions

## Recovery Phase State Transitions:

Current Phase	Input Event	Next Phase	State Updates
Pre-Recovery	Master Record Found	Analysis	Initialize recovery structures
Pre-Recovery	No Master Record	Analysis	Start from beginning of log
Analysis	Log Record Scanned	Analysis	Update transaction/dirty page tables
Analysis	End of Log Reached	Redo	Tables complete, calculate recovery LSN
Redo	Redo Record Applied	Redo	Page updated, LSN advanced
Redo	End of Log Reached	Undo	All committed changes restored
Undo	Transaction Rolled Back	Undo	Transaction marked aborted
Undo	All Active Transactions Processed	Complete	Database consistent and ready

## Decision: ARIES Three-Phase Recovery Algorithm

- **Context:** Multiple recovery algorithms exist (shadow paging, log-based recovery, etc.) with different complexity and performance characteristics
- **Options Considered:**
  - Shadow paging (simple but poor performance)
  - Two-phase recovery: redo then undo (missing analysis can miss transactions)
  - ARIES three-phase: analysis, redo, undo (complex but comprehensive)
- **Decision:** Implement full ARIES three-phase recovery with compensation log records
- **Rationale:** ARIES provides the strongest correctness guarantees, handles all edge cases (including recovery from recovery failures), and is the proven industry standard
- **Consequences:** Increased implementation complexity but maximum robustness and the ability to recover from any crash scenario

## Recovery Error Handling and Edge Cases

### Recovery Corruption Scenarios:

Failure Mode	Detection Method	Recovery Action	Prevention Strategy
Corrupted Master Record	CRC validation failure	Scan for previous checkpoint	Maintain backup master records
Partial Log Record	Deserialization failure	Truncate at corruption point	Write atomic record headers
Missing Log Segment	File not found during scan	Stop recovery at gap	Prevent premature segment deletion
Inconsistent Transaction State	Transaction in multiple final states	Use latest record timestamp	Validate transaction state transitions
Redo Idempotency Violation	Page LSN greater than record LSN	Skip redundant application	Always check page LSN before redo
Undo Chain Corruption	Invalid undo_next_lsn pointer	Stop transaction rollback	Validate LSN ranges during log write

**⚠ Pitfall: Recovery Starting Point Miscalculation** Starting recovery from an LSN that's too recent can miss necessary redo operations, while starting too early wastes time processing already-applied records. The `min_recovery_lsn()` calculation must consider both the earliest dirty page and the oldest active transaction to ensure correctness.

**⚠ Pitfall: Non-Idempotent Recovery Operations** Applying redo or undo operations without proper LSN checks can result in applying the same change multiple times, corrupting database state. Always verify page LSN against record LSN before applying changes, and generate appropriate Compensation Log Records during undo operations.

**⚠ Pitfall: Incomplete Analysis Phase Data** Missing transaction or dirty page information during Analysis phase leads to incorrect Redo and Undo behavior. Ensure that all log record types update appropriate data structures, and handle checkpoint records properly to integrate their snapshot information.

## Checkpoint Coordination

Checkpoint coordination represents the most **complex orchestration challenge** in the WAL system. Checkpoints must create consistent recovery waypoints while allowing concurrent transactions to continue unimpeded—a delicate balance between consistency and availability.

Think of checkpoint coordination as **creating a museum exhibit while visitors are still touring**. The curator (checkpoint manager) must document the current state of all displays (database pages) and visitor activities (active transactions) to create a snapshot that future visitors can use as a starting point. However, the museum remains open during this process, so new visitors arrive and exhibits change while the documentation is being created. The curator must use sophisticated techniques to ensure the snapshot represents a coherent moment in time despite ongoing changes.

The complexity stems from the **fundamental tension between consistency and concurrency**. A truly consistent checkpoint would require stopping all transactions, but this violates availability requirements. Fuzzy checkpoints solve this by creating temporally consistent snapshots without blocking operations, but require sophisticated coordination protocols.

## Fuzzy Checkpoint Algorithm Design

The fuzzy checkpoint algorithm creates consistent recovery waypoints through a **four-phase coordination protocol** that maintains temporal consistency without blocking concurrent transactions:

1. **Checkpoint initiation trigger** where automatic triggers (time-based, dirty page count, or log size) or manual requests activate `CheckpointManager`
2. **State machine transition** where `begin_checkpoint()` atomically transitions `CheckpointStateMachine` from `Idle` to `Collecting` state
3. **Checkpoint LSN selection** where current `LogWriter::next_lsn` is captured as the temporal boundary for checkpoint consistency
4. **Concurrent transaction coordination** where new transactions starting after checkpoint LSN are tracked separately from pre-checkpoint transactions
5. **Dirty page table snapshot** where current `DirtyPageTable` contents are captured, including minimum recovery LSN calculations

## Active Transaction State Capture

6. **Transaction table enumeration** where `active_transactions()` iterator provides snapshot of all transactions with `TransactionStatus::Active`
7. **Transaction LSN tracking** where each active transaction's current LSN and undo chain head are recorded in checkpoint metadata
8. **Lock state preservation** where information about held locks and waiting transactions is captured for potential recovery coordination
9. **Buffer pool state documentation** where uncommitted changes in memory are identified and associated with their originating transactions
10. **Temporal consistency validation** where captured transaction states are verified to be consistent with the selected checkpoint LSN boundary

## Checkpoint Record Construction and Logging

11. **CheckpointRecord assembly** where `active_transactions` list and `dirty_pages` list are combined into checkpoint metadata
12. **Checkpoint LSN assignment** where the checkpoint record receives an LSN higher than all transactions included in its snapshot
13. **Atomic checkpoint logging** where `write_and_force()` ensures checkpoint record is durably written before proceeding

14. **State machine advancement** where `transition_to_writing()` moves checkpoint state from `Collecting` to `Writing`
15. **Master record preparation** where new `MasterRecord` is constructed with checkpoint LSN, segment information, and validation checksums

## Master Record Update and Finalization

16. **Atomic master record replacement** where new master record is written to temporary file and atomically renamed to replace previous version
17. **Master record validation** where updated master record is read back and validated using `validate()` to ensure integrity
18. **Checkpoint completion signaling** where `complete_checkpoint()` transitions state machine to `Complete` and notifies waiting operations
19. **Log truncation opportunity** where `calculate_safe_truncation_lsn()` determines obsolete log segments that can be safely removed
20. **Checkpoint state reset** where state machine returns to `Idle` and prepares for next checkpoint cycle

## Checkpoint State Machine Transitions:

Current State	Trigger Event	Next State	Actions Performed
Idle	Automatic/Manual Trigger	Collecting	Begin snapshot capture
Collecting	Snapshot Complete	Writing	Create checkpoint record
Writing	Record Logged	Complete	Update master record
Complete	Cleanup Finished	Idle	Reset for next checkpoint
Any State	Error Detected	Idle	Abort checkpoint, cleanup

## Decision: Fuzzy Checkpoint with Temporal Consistency

- **Context:** Checkpoint creation must balance consistency requirements with system availability during checkpoint operations
- **Options Considered:**
  - Sharp checkpoint blocking all transactions (simple but disrupts availability)
  - Fuzzy checkpoint with snapshot isolation (complex coordination)
  - Incremental checkpoints tracking only changes (very complex state management)
- **Decision:** Implement fuzzy checkpoints with LSN-based temporal consistency boundaries
- **Rationale:** Provides strong consistency guarantees without blocking concurrent operations, using LSN ordering to maintain temporal coherence
- **Consequences:** Requires sophisticated coordination logic but enables high availability with bounded recovery time

## Log Truncation Safety Protocol

Log truncation requires **extreme caution** because prematurely deleted log records make recovery impossible. The truncation protocol ensures that only truly obsolete records are removed:

21. **Safe truncation LSN calculation** where `calculate_safe_truncation_lsn()` finds the minimum LSN that must be retained
22. **Active transaction dependency analysis** where oldest active transaction's first LSN establishes lower bound for truncation
23. **Dirty page recovery requirements** where earliest dirty page LSN establishes recovery scanning requirements
24. **Checkpoint consistency validation** where checkpoint record completeness is verified before allowing truncation beyond checkpoint LSN
25. **Multi-checkpoint safety margin** where truncation preserves at least two complete checkpoints to handle checkpoint corruption scenarios
26. **Segment-level truncation planning** where obsolete segments are identified based on their LSN ranges relative to safe truncation LSN
27. **Atomic segment removal** where obsolete segment files are deleted in LSN order to maintain log continuity
28. **Truncation logging** where truncation operations are logged to aid debugging and provide audit trail
29. **Master record updates** where successful truncation updates master record to reflect new log starting point

30. **Truncation completion verification** where remaining log segments are validated for continuity and completeness

#### Checkpoint Coordination Interactions:

Component	Role in Checkpoint	Data Provided	Coordination Requirements
LogWriter	LSN boundary source	Current next_lsn	Must not rotate during critical phases
TransactionTable	Active transaction snapshot	Transaction states, LSNs	Must handle concurrent updates
DirtyPageTable	Recovery requirements	Dirty pages, min LSN	Must track checkpoint-era changes
RecoveryManager	Consistency validation	Recovery implications	Must validate truncation safety
Storage	Database state	Page versions	Must coordinate with dirty tracking

**Key Insight:** Checkpoint coordination succeeds through **LSN-based temporal boundaries** that create consistent points in time without requiring global synchronization. The checkpoint LSN serves as a "fence" that separates pre-checkpoint state (captured in the checkpoint) from post-checkpoint state (handled by subsequent recovery).

#### Checkpoint Performance Optimization

The checkpoint process itself can become a performance bottleneck, requiring careful optimization to minimize system impact:

31. **Background checkpoint threading** where checkpoint operations run in dedicated threads to avoid blocking transaction processing
32. **Incremental dirty page scanning** where large dirty page tables are processed in batches to avoid long pause times
33. **Checkpoint scheduling coordination** where checkpoints are timed to avoid peak transaction periods when possible
34. **Buffer pool coordination** where checkpoint creation coordinates with buffer eviction policies to minimize memory pressure
35. **I/O prioritization** where checkpoint writes use lower I/O priority than transaction log writes to avoid interference

#### Checkpoint Coordination Pitfalls:

**⚠ Pitfall: Inconsistent Temporal Boundaries** Capturing active transaction and dirty page snapshots at different times creates temporal inconsistencies that lead to recovery errors. Always capture all checkpoint

state using a single LSN boundary, and ensure that concurrent updates don't affect the snapshot after the boundary is established.

**⚠ Pitfall: Premature Log Truncation** Truncating log segments before verifying that the checkpoint record is durably written and the master record is updated creates unrecoverable situations. Always complete the entire checkpoint protocol, including master record updates, before performing any truncation operations.

**⚠ Pitfall: Master Record Corruption During Updates** Non-atomic master record updates can leave the system in a state where no valid checkpoint is available, making recovery impossible. Use atomic file operations (write to temporary file, then atomic rename) to ensure master record updates are all-or-nothing.

**⚠ Pitfall: Checkpoint State Machine Race Conditions** Concurrent checkpoint requests or state machine transitions can corrupt checkpoint state and produce invalid checkpoints. Use proper synchronization primitives and validate state transitions to ensure only one checkpoint operation proceeds at a time.

## Implementation Guidance

This implementation guidance provides the concrete foundation for building the interaction flows described above. The focus is on providing complete, working infrastructure that demonstrates proper component coordination.

### A. Technology Recommendations:

Component	Simple Option	Advanced Option
Concurrency Control	<code>Arc&lt;Mutex&lt;T&gt;&gt;</code> for all shared state	<code>RwLock</code> for read-heavy, lock-free for performance-critical paths
Error Handling	<code>Result&lt;T, WalError&gt;</code> with explicit error propagation	Custom error types with context chaining
Async Coordination	<code>std::thread</code> with channels	<code>tokio</code> async runtime with <code>async/await</code>
Testing	Unit tests with manual crash simulation	Property-based testing with <code>proptest</code> crate
Logging	<code>println!</code> debugging statements	<code>tracing</code> crate with structured logging

### B. Recommended File Structure:

```
src/
  lib.rs           ← Public API exports
  wal/
    mod.rs         ← Module declarations
    types.rs       ← Core types (LSN, TransactionId, etc.)
    log_writer.rs  ← LogWriter implementation
    log_reader.rs  ← LogReader implementation
    recovery_manager.rs  ← RecoveryManager implementation
    checkpoint_manager.rs  ← CheckpointManager implementation
    coordinator.rs  ← This section: interaction coordination
    buffer_pool.rs  ← Buffer management utilities
  storage/
    mod.rs         ← Storage trait and implementations
    memory_storage.rs  ← MockDatabase for testing
    page.rs        ← DatabasePage implementation
  testing/
    mod.rs         ← Test utilities and fixtures
    crash_simulator.rs  ← Crash testing infrastructure
  examples/
    simple_transaction.rs  ← Demonstrates write transaction flow
    recovery_demo.rs      ← Demonstrates crash recovery
    checkpoint_demo.rs    ← Demonstrates checkpoint coordination
```

### C. Infrastructure Starter Code:

```
// src/wal/coordinator.rs - Complete transaction coordination infrastructure

use crate::wal::{LogWriter, RecoveryManager, CheckpointManager, WalResult, WalError};

use crate::storage::{Storage, DatabasePage};

use std::collections::HashMap;

use std::sync::{Arc, Mutex, RwLock};

use std::time::{Duration, Instant};

/// Coordinates interactions between WAL components during normal operations

pub struct TransactionCoordinator {

    log_writer: Arc<LogWriter>,

    recovery_manager: Arc<Mutex<RecoveryManager>>,

    checkpoint_manager: Arc<CheckpointManager>,

    storage: Arc<dyn Storage>,

    active_transactions: Arc<RwLock<HashMap<TransactionId, TransactionContext>>>,

    transaction_counter: Arc<Mutex<u64>>,

    shutdown_signal: Arc<Mutex<bool>>,

}

/// Context information for active transactions

#[derive(Debug, Clone)]

pub struct TransactionContext {

    pub txn_id: TransactionId,

    pub start_time: Instant,

    pub operations: Vec<PendingOperation>,

    pub locks_held: Vec<PageId>,

    pub status: TransactionStatus,

}
```

```
/// Operation waiting to be applied to database

#[derive(Debug, Clone)]

pub struct PendingOperation {

    pub page_id: PageId,
    pub offset: u32,
    pub before_image: Vec<u8>,
    pub after_image: Vec<u8>,
}

impl TransactionCoordinator {

    pub fn new(
        log_writer: Arc<LogWriter>,
        recovery_manager: Arc<Mutex<RecoveryManager>>,
        checkpoint_manager: Arc<CheckpointManager>,
        storage: Arc<dyn Storage>,
    ) -> Self {
        Self {
            log_writer,
            recovery_manager,
            checkpoint_manager,
            storage,
            active_transactions: Arc::new(RwLock::new(HashMap::new())),
            transaction_counter: Arc::new(Mutex::new(1)),
            shutdown_signal: Arc::new(Mutex::new(false)),
        }
    }

    /// Initialize system after potential crash - runs complete recovery
}
```

```
pub fn initialize(&self) -> WalResult<()> {
    println!("Starting WAL system initialization...");

    // Recovery must complete before accepting new transactions

    let mut recovery = self.recovery_manager.lock().unwrap();
    recovery.recover()?;
    println!("Recovery completed successfully");
    Ok(())
}

/// Begin new transaction and return transaction ID

pub fn begin_transaction(&self) -> WalResult<TransactionId> {
    let mut counter = self.transaction_counter.lock().unwrap();
    let txn_id = *counter;
    *counter += 1;
    drop(counter);

    let context = TransactionContext {
        txn_id,
        start_time: Instant::now(),
        operations: Vec::new(),
        locks_held: Vec::new(),
        status: TransactionStatus::Active,
    };

    let mut active = self.active_transactions.write().unwrap();
    active.insert(txn_id, context);
}
```

```
drop(active);

println!("Started transaction {}", txn_id);

Ok(txn_id)

}

/// Shutdown coordinator gracefully

pub fn shutdown(&self) -> WalResult<()> {

    // Signal shutdown to background threads

    *self.shutdown_signal.lock().unwrap() = true;

    // Wait for active transactions to complete or timeout after 30 seconds

    let timeout = Duration::from_secs(30);

    let start = Instant::now();

    loop {

        let active_count = self.active_transactions.read().unwrap().len();

        if active_count == 0 || start.elapsed() > timeout {

            break;
        }

        std::thread::sleep(Duration::from_millis(100));
    }

    // Force final checkpoint before shutdown

    self.checkpoint_manager.create_checkpoint(
        self.get_active_transaction_ids(),
        self.get_dirty_page_ids(),
    )?;
}
```

```
    println!("WAL coordinator shutdown complete");

    Ok(())
}

fn get_active_transaction_ids(&self) -> Vec<TransactionId> {
    self.active_transactions.read().unwrap().keys().copied().collect()
}

fn get_dirty_page_ids(&self) -> Vec<PageId> {
    // This would integrate with actual dirty page tracking
    Vec::new()
}

/// Utility for simulating crashes during testing
pub struct CrashSimulator {
    crash_points: Vec<String>,
    current_operation: String,
}

impl CrashSimulator {
    pub fn new() -> Self {
        Self {
            crash_points: Vec::new(),
            current_operation: String::new(),
        }
    }
}
```

```
pub fn add_crash_point(&mut self, operation: &str) {  
    self.crash_points.push(operation.to_string());  
}  
  
pub fn set_current_operation(&mut self, operation: &str) {  
    self.current_operation = operation.to_string();  
  
    // Check if we should crash at this point  
    if self.crash_points.contains(&self.current_operation) {  
        panic!("Simulated crash at: {}", operation);  
    }  
}  
}
```

#### D. Core Logic Skeleton Code:

```
impl TransactionCoordinator {  
  
    /// Execute complete write transaction flow - learner implements this  
  
    pub fn execute_transaction(&self, txn_id: TransactionId, operations: Vec<PendingOperation>) -> WalResult<()> {  
  
        // TODO 1: Validate transaction exists and is in Active status  
  
        // TODO 2: For each operation, create RedoRecord and UndoRecord pairs  
  
        // TODO 3: Write all log records using write_and_force() for durability  
  
        // TODO 4: Apply after-images to database pages using storage.write_data()  
  
        // TODO 5: Create and write CommitRecord with write_and_force()  
  
        // TODO 6: Update transaction status to Committed  
  
        // TODO 7: Release all locks held by transaction  
  
        // TODO 8: Remove transaction from active_transactions map  
  
        // Hint: Use crash simulator to test failure at each step  
  
        todo!("Implement complete transaction execution with proper error handling")  
  
    }  
  
    /// Handle transaction abort/rollback - learner implements this  
  
    pub fn abort_transaction(&self, txn_id: TransactionId) -> WalResult<()> {  
  
        // TODO 1: Validate transaction exists and get its current state  
  
        // TODO 2: Create AbortRecord and write with write_and_force()  
  
        // TODO 3: For each operation in reverse order, create UndoRecord  
  
        // TODO 4: Apply before-images to database pages to undo changes  
  
        // TODO 5: Update transaction status to Aborted  
  
        // TODO 6: Release all locks held by transaction  
  
        // TODO 7: Remove transaction from active_transactions map  
  
        // Hint: Follow undo chain in reverse LSN order for proper rollback  
  
        todo!("Implement transaction rollback with compensation logging")  
  
    }  
}
```

```

/// Coordinate checkpoint creation with concurrent transactions

pub fn trigger_checkpoint(&self) -> WalResult<LSN> {

    // TODO 1: Check if checkpoint is already in progress using state machine

    // TODO 2: Capture current LSN as checkpoint boundary from log_writer

    // TODO 3: Snapshot active_transactions at checkpoint LSN boundary

    // TODO 4: Snapshot dirty pages from storage dirty page tracking

    // TODO 5: Call checkpoint_manager.create_checkpoint() with snapshots

    // TODO 6: Update master record with new checkpoint information

    // TODO 7: Calculate safe truncation LSN and remove obsolete segments

    // TODO 8: Return checkpoint LSN for caller confirmation

    // Hint: Use LSN ordering to ensure temporal consistency of snapshots

    todo!("Implement fuzzy checkpoint coordination without blocking transactions")

}

}

```

## E. Language-Specific Hints:

- **Concurrency:** Use `Arc<Mutex<T>>` for shared state, `RwLock` for read-heavy workloads
- **Error Handling:** Define custom `WalError` enum and use `Result<T, WalError>` consistently
- **File I/O:** Use `std::fs::File` with `sync_all()` for fsync operations
- **Serialization:** Use `bincode` crate for efficient binary serialization of log records
- **Testing:** Use `std::panic::catch_unwind` to simulate crashes in unit tests
- **Threading:** Use `std::thread::spawn` for background checkpoint operations

## F. Milestone Checkpoints:

### After Milestone 2 (Log Writer):

- Run: `cargo test transaction_flow_basic`
- Expected: Simple write transaction completes with proper log record ordering
- Manual test: Create transaction, verify log records written before database changes
- Debug: If transactions hang, check fsync completion and lock release

### After Milestone 3 (Recovery):

- Run: `cargo test recovery_integration`
- Expected: Crash simulation followed by successful recovery restores correct state
- Manual test: Kill process during transaction, restart and verify recovery completion
- Debug: If recovery fails, check transaction table building and redo/undo LSN ordering

#### After Milestone 4 (Checkpointing):

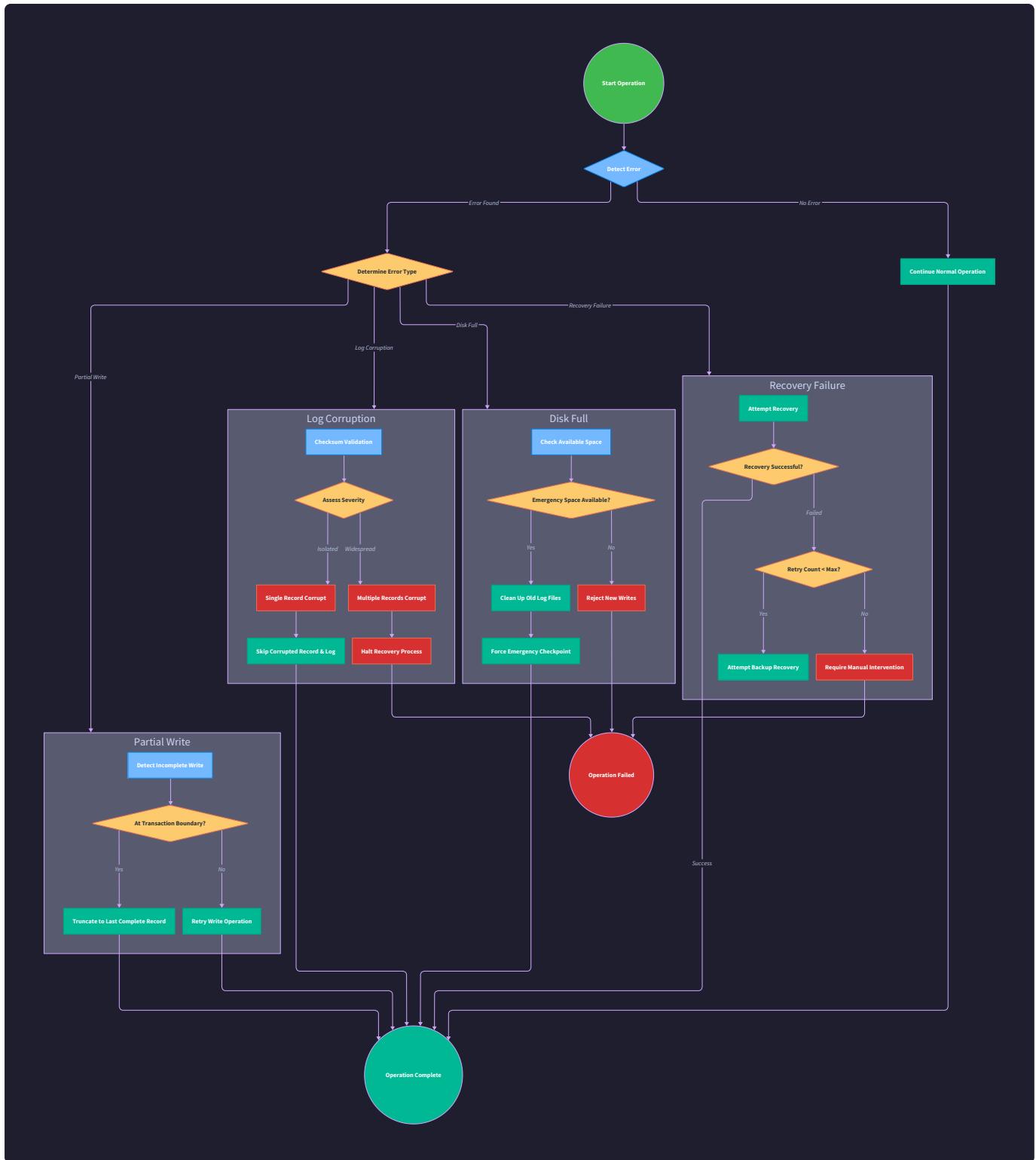
- Run: `cargo test checkpoint_coordination`
- Expected: Checkpoint creation doesn't block concurrent transactions
- Manual test: Run transactions during checkpoint, verify both complete successfully
- Debug: If deadlocks occur, check state machine transitions and LSN boundary consistency

#### G. Debugging Tips:

Symptom	Likely Cause	Diagnosis	Fix
Transactions never commit	fsync failing or hanging	Check <code>write_and_force()</code> completion	Add timeout and error handling to fsync calls
Recovery produces wrong state	Redo/undo order incorrect	Trace LSN ordering in log records	Ensure redo processes in forward LSN order, undo in reverse
Checkpoints corrupt recovery	Temporal inconsistency in snapshots	Check LSN boundaries in checkpoint	Capture all snapshot data using same LSN fence
Deadlocks during checkpoint	Lock ordering conflicts	Trace lock acquisition order	Always acquire locks in consistent order (e.g., by PageId)
Log corruption after crashes	Partial writes during segment rotation	Check atomic append implementation	Ensure record headers written atomically

## Error Handling and Edge Cases

**Milestone(s):** All milestones - error handling strategies span Milestone 1 (Log Record Format) corruption detection, Milestone 2 (Log Writer) partial write handling, Milestone 3 (Crash Recovery) recovery failures, and Milestone 4 (Checkpointing) checkpoint consistency failures



Think of WAL error handling like building a nuclear power plant - you need multiple layers of defense in depth. The first layer prevents errors from occurring (checksums, atomic operations), the second layer detects when they do occur (validation, monitoring), the third layer contains their impact (graceful degradation), and the final layer recovers from catastrophic failures (emergency shutdown and restart procedures). Unlike application-level errors that might be recoverable, WAL errors often threaten data consistency itself, so our error handling must be extremely conservative - when in doubt, we halt operations rather than risk data corruption.

The fundamental challenge in WAL error handling is that the logging system itself is the foundation for database consistency. If the WAL becomes corrupted or unreliable, we lose our ability to guarantee ACID properties. This creates a bootstrapping problem: how do we maintain consistency when our consistency mechanism fails? The answer lies in designing multiple independent validation layers and always failing safe - it's better to refuse a transaction than to commit one incorrectly.

## Corruption Detection

**Log record corruption** represents the most insidious type of failure because it can silently compromise database consistency. Unlike obvious failures like disk full or network outages, corruption can masquerade as valid data until it's too late to recover. Our corruption detection strategy operates on multiple levels: individual record validation, structural consistency checks, and cross-reference validation between related records.

### Decision: CRC32 Checksums for Individual Records

- **Context:** Need to detect bit flips, partial writes, and media errors in individual log records
- **Options Considered:** No checksums (faster but unsafe), CRC32 (good balance), SHA-256 (cryptographically secure but slower)
- **Decision:** CRC32 checksum appended to each log record
- **Rationale:** CRC32 provides excellent detection of random errors with minimal CPU overhead, and we're protecting against hardware failures not adversarial attacks
- **Consequences:** 4 bytes overhead per record, simple validation logic, can detect 99.99% of random corruption

The CRC32 checksum calculation encompasses the entire serialized log record including the header fields (LSN, transaction ID, record type) and the payload data (before/after images, operation metadata). This ensures that any modification to the record - whether in critical fields like the LSN or in data payload - will be detected during validation.

Corruption Detection Level	Method	Coverage	Performance Impact
Individual Record	CRC32 checksum	Single record integrity	~1-2% CPU overhead
Structural Validation	LSN sequence check	Logical consistency	~5% during recovery
Cross-Reference	Transaction ID validation	Inter-record relationships	~10% during analysis
Segment Validation	File size vs record count	Physical structure	~1% during startup

**Structural validation** goes beyond individual record checksums to verify the logical consistency of the log structure. The most critical check is LSN sequence validation - since Log Sequence Numbers must be monotonically increasing, any gap or duplication indicates corruption or incomplete recovery. During log scanning, we maintain the expected next LSN and flag any deviation as a structural error.

```
// Example of structural validation during log scanning
```

RUST

```
let mut expected_lsn = start_lsn;

for record_result in log_reader.scan_from(start_lsn) {

    match record_result {

        Ok(record) => {

            if record.lsn() != expected_lsn {

                return Err(WalError::Corruption(format!(
                    "LSN gap: expected {}, found {}",
                    expected_lsn, record.lsn()
                )));
            }

            expected_lsn += 1;
        }

        Err(e) => return Err(e),
    }
}
```

**Transaction consistency validation** ensures that every `RedoRecord` and `UndoRecord` references a valid transaction ID that was properly initialized with a transaction begin record. During the analysis pass of recovery, we build a transaction table and validate that every operation record corresponds to an active transaction. Orphaned operation records - those without corresponding transaction records - indicate either corruption or incomplete log writes.

### Critical Insight: Corruption vs. Incompleteness

It's essential to distinguish between corruption (data was written but became invalid) and incompleteness (operation was interrupted before completion). Corruption requires aborting recovery and manual intervention, while incompleteness can often be handled automatically by discarding partial records and rolling back incomplete transactions.

**Detection timing** varies based on the corruption type and system state. Some corruption is detected immediately during normal operations (CRC validation on every read), while other corruption is only discovered during recovery when we perform comprehensive validation. This creates a trade-off: more

aggressive validation catches problems earlier but reduces performance, while lazy validation is faster but allows corrupted state to persist longer.

Detection Point	Corruption Types Caught	Performance Impact	Recovery Complexity
Write-time validation	Serialization errors, memory corruption	High (every write)	Low (fail fast)
Read-time validation	Storage corruption, bit flips	Medium (every read)	Medium (retry/repair)
Recovery validation	Structural inconsistencies, missing records	Low (only during recovery)	High (complex analysis)
Background validation	Gradual corruption, aging media	Minimal (periodic scans)	Variable (depends on extent)

## Partial Write Handling

**Partial writes** occur when the operating system crashes or loses power during a write operation, leaving some bytes written to disk while others remain in kernel buffers or are never written at all. This creates a "torn page" scenario where the log file contains a mix of old and new data. Unlike corruption, partial writes have a predictable structure - they typically affect the end of the last write operation while leaving earlier data intact.

The fundamental challenge with partial writes is that they can make the log structurally invalid even when individual records are not corrupted. For example, if we're writing a 1KB log record but only the first 512 bytes make it to disk, we have a valid CRC for the first half but an incomplete record that will cause parsing errors during recovery.

### Decision: Record Length Prefix for Partial Write Detection

- **Context:** Need to detect when log records are incompletely written to disk during crashes
- **Options Considered:** Fixed-size records (wasteful), no detection (unsafe), length prefix (flexible)
- **Decision:** Include total record length in the fixed-size header
- **Rationale:** Allows parser to know exactly how many bytes to read and detect truncation
- **Consequences:** Slight space overhead but enables robust partial write detection

Our partial write detection strategy operates at the record level using a two-phase validation approach. First, we read the fixed-size record header (21 bytes) which includes the total record length. Then we attempt to read exactly that many additional bytes from the log file. If we encounter EOF or read fewer bytes than expected, we've detected a partial write.

Partial Write Scenario	Detection Method	Recovery Action	Data Loss
Truncated header	EOF during header read	Discard partial record	Current transaction only
Truncated payload	Length mismatch after header	Discard partial record	Current transaction only
Torn page within record	CRC validation failure	Discard corrupted record	Current transaction only
Multiple record truncation	Repeated EOF during scan	Truncate log at last valid record	Multiple transactions

**Recovery strategy** for partial writes follows a conservative "truncate and replay" approach. When we detect a partial write during log scanning, we truncate the log file at the last completely valid record and continue recovery from that point. This ensures that the log maintains structural integrity even after a crash, but it means that any transactions whose commit records were partially written will be rolled back during recovery.

The truncation decision requires careful consideration of the system state. If we're in the middle of normal operations and detect a partial write in a log segment that should be complete, this indicates a serious storage problem and we should halt operations. However, if we're scanning the log during recovery and encounter a partial write at the very end of the last segment, this is expected behavior after a crash and we can safely truncate.

```
// Example of partial write detection during log scanning

fn scan_log_record(reader: &mut LogReader) -> WalResult<Option<LogRecord>> {

    // Read fixed header first

    let mut header_buf = [0u8; RECORD_HEADER_SIZE];

    match reader.read_exact(&mut header_buf) {

        Ok(()) => {},

        Err(ref e) if e.kind() == std::io::ErrorKind::UnexpectedEof => {

            // Partial header - truncate here

            return Ok(None);
        },

        Err(e) => return Err(WalError::Io(e)),
    }

    // Parse header to get total length

    let total_length = parse_header_length(&header_buf)?;

    let payload_length = total_length - RECORD_HEADER_SIZE;

    // Read payload

    let mut payload_buf = vec![0u8; payload_length];

    match reader.read_exact(&mut payload_buf) {

        Ok(()) => {},

        Err(ref e) if e.kind() == std::io::ErrorKind::UnexpectedEof => {

            // Partial payload - truncate at previous record

            return Ok(None);
        },

        Err(e) => return Err(WalError::Io(e)),
    }
}
```

```

// Validate complete record

let mut complete_record = header_buf.to_vec();

complete_record.extend_from_slice(&payload_buf);

LogRecord::deserialize(&complete_record).map(Some)

}

```

**Atomic append semantics** provide additional protection against partial writes by ensuring that each log record is written as an indivisible unit. We achieve this through careful buffer management and strategic use of `fsync`. Before writing any record, we ensure that the complete serialized record (header + payload + CRC) is prepared in memory. We then write the entire record in a single system call and follow with `fsync` to guarantee durability.

However, atomic append only protects against partial writes within a single record. If we're writing multiple records in a batch, a crash could still leave some records written and others not. This is why our recovery algorithm must handle variable-length incomplete batches and why we always scan forward from the last known good `LSN` during recovery.

## Disk Full Scenarios

**Disk exhaustion** represents a particularly challenging failure mode because it can occur gradually during normal operations or suddenly when a large transaction attempts to commit. Unlike corruption or partial writes which affect data integrity, disk full scenarios threaten system availability - we need to gracefully degrade service while preserving consistency for all completed operations.

The primary challenge with disk full handling is maintaining the write-ahead logging guarantee even when we cannot write new log records. Since WAL requires that all changes be logged before they're applied to the database, running out of log space effectively prevents all new write transactions from committing. However, we must still allow read transactions to continue and ensure that any in-flight write transactions are properly resolved (either committed or aborted).

## Decision: Reserved Space for Emergency Operations

- **Context:** Need to handle critical operations like transaction aborts and checkpoints even when disk is full
- **Options Considered:** Fail immediately (unsafe), ignore disk full (corrupts WAL), reserve emergency space
- **Decision:** Maintain a reserved space pool for critical WAL operations
- **Rationale:** Allows graceful handling of active transactions and emergency checkpoints when primary space is exhausted
- **Consequences:** Reduces effective storage capacity but prevents deadlock scenarios

Our disk full handling strategy operates at multiple levels: proactive monitoring to prevent exhaustion, graceful degradation when space runs low, and emergency procedures when no space remains. The key insight is that different types of log records have different criticality levels - abort records and compensation log records are essential for consistency, while new transaction records can be deferred or rejected.

Disk Space Level	Available Operations	Restrictions	Recovery Actions
Normal (>10% free)	All operations allowed	None	Periodic cleanup
Warning (5-10% free)	Existing transactions only	No new write transactions	Trigger emergency checkpoint
Critical (1-5% free)	Abort and commit only	Read-only mode for new clients	Aggressive log truncation
Emergency (<1% free)	Essential records only	Use reserved space pool	Manual intervention required

**Proactive space management** involves monitoring disk usage and taking preventive action before exhaustion occurs. We track both the current log directory size and the rate of log growth to predict when space will run out. When free space drops below configurable thresholds, we trigger increasingly aggressive space reclamation procedures: first attempting normal checkpoint and log truncation, then forcing emergency checkpoints, and finally entering read-only mode.

The space monitoring system must account for the fact that log growth is bursty - a large transaction might need to write many megabytes of redo records in a short time. Therefore, our space calculations include a safety margin based on the maximum expected transaction size and the number of currently active transactions.

```

// Example of disk space monitoring and threshold management

fn check_disk_space(&mut self) -> WalResult<DiskSpaceStatus> {

    let stats = fs::metadata(&self.log_directory)?;

    let available_space = get_available_space(&self.log_directory)?;

    let current_log_size = self.calculate_total_log_size()?;

    // Estimate space needed for active transactions

    let active_tx_count = self.transaction_table.active_count();

    let estimated_space_needed = active_tx_count * self.config.max_transaction_size;

    let effective_available = available_space.saturating_sub(estimated_space_needed);

    match effective_available {

        space if space < self.config.emergency_threshold => DiskSpaceStatus::Emergency,
        space if space < self.config.critical_threshold => DiskSpaceStatus::Critical,
        space if space < self.config.warning_threshold => DiskSpaceStatus::Warning,
        _ => DiskSpaceStatus::Normal,
    }
}

```

**Graceful degradation** begins when we detect low disk space conditions. The first step is to reject new write transactions while allowing existing transactions to complete normally. This prevents the situation from getting worse while giving active transactions a chance to finish and free up space through log truncation after checkpointing.

When space becomes critically low, we enter a more aggressive mode where we attempt to abort some active transactions to free up space. The transaction selection algorithm prioritizes aborting transactions that have written many log records (to maximize space recovery) while preserving transactions that are close to completion. This requires careful coordination with the transaction manager to ensure that aborted transactions are properly rolled back and their log records can be safely truncated.

**Emergency procedures** activate when disk space is completely exhausted and normal operations cannot continue. At this point, we switch to using the reserved space pool, which is a pre-allocated set of disk blocks that are kept available specifically for emergency operations. The reserved space is only used for essential records: abort records to roll back active transactions, compensation log records generated during undo operations, and a final checkpoint record to enable clean restart.

The size of the reserved space pool must be carefully calculated based on the maximum number of concurrent transactions and the expected size of undo operations. A typical calculation is: `reserved_space = max_concurrent_transactions * average_undo_record_size * average_undo_chain_length + checkpoint_record_size`.

## Recovery Failures

**Recovery failures** represent the most critical error scenario because they occur when the system is already in a compromised state following a crash. Unlike errors during normal operations, recovery failures cannot be resolved by simply rolling back the current transaction - the entire database may be in an inconsistent state that requires manual intervention to resolve.

The fundamental challenge in recovery error handling is that the recovery process itself modifies both the database and the log (through compensation log records). If recovery fails partway through, we may have applied some changes but not others, leaving the system in a state that's neither the pre-crash state nor the correctly recovered state. This is why recovery procedures must be designed to be **idempotent** - we should be able to restart recovery from the beginning multiple times without causing additional corruption.

### Decision: Restart Recovery from Checkpoint on Any Failure

- **Context:** Recovery failures leave the system in an unknown intermediate state between crashed and recovered
- **Options Considered:** Continue from failure point (complex), manual recovery (requires expertise), restart from checkpoint (safe but slower)
- **Decision:** Always restart complete recovery from the last valid checkpoint on any recovery error
- **Rationale:** Ensures consistent starting state and leverages idempotent recovery operations
- **Consequences:** Slower recovery after errors but guaranteed consistency and simpler error handling logic

Our recovery failure handling strategy is based on the principle of "recovery atomicity" - either recovery completes entirely, or we return to a known consistent state (the last checkpoint) and retry. This approach sacrifices some performance for correctness and simplicity, which is appropriate given that recovery failures are rare and correctness is paramount.

Recovery Phase	Failure Types	Detection Method	Recovery Action
Analysis Pass	Log corruption, missing records	Checksum validation, LSN gaps	Restart from previous checkpoint
Redo Pass	Storage errors, page corruption	Write failures, validation errors	Mark database offline, manual intervention
Undo Pass	Undo record corruption, logic errors	CRC failures, constraint violations	Restart complete recovery
Checkpoint Creation	Disk full, write errors	fsync failures, space exhaustion	Continue without checkpoint, retry later

**Analysis phase failures** typically stem from log corruption discovered during the forward scan. Since the analysis phase only reads log records and builds in-memory data structures (transaction table and dirty page table), these failures are relatively safe to handle by restarting recovery from an earlier checkpoint. However, if multiple consecutive checkpoints have corrupted log records, we may need to scan backward to find a valid recovery starting point.

The analysis phase implements several corruption recovery techniques: when we encounter a corrupted record, we attempt to skip ahead to the next valid record by scanning for the next valid LSN sequence. If too many records in a row are corrupted (indicating possible storage media failure), we abort recovery and require manual intervention.

```
// Example of analysis phase error handling with checkpoint fallback

fn analysis_pass(&mut self) -> WalResult<()> {

    let mut scan_start = self.checkpoint_lsn.unwrap_or(0);

    let mut consecutive_errors = 0;

    loop {

        match self.scan_log_from(scan_start) {

            Ok(_) => {

                // Analysis completed successfully

                return Ok(());

            },

            Err(WalError::Corruption(_)) if consecutive_errors < 3 => {

                // Try previous checkpoint

                consecutive_errors += 1;

                scan_start = self.find_previous_checkpoint(scan_start)?;

                self.reset_recovery_state();

            },

            Err(e) => {

                // Too many consecutive failures or non-corruption error

                return Err(e);

            }

        }

    }

}
```

**Redo phase failures** are more serious because they occur while we're modifying the database storage. If a redo operation fails due to storage errors or page corruption, we cannot simply restart recovery because the database may already be in a partially modified state. In this case, our only safe option is to mark the

database as offline and require manual intervention to determine whether the storage can be repaired or must be restored from backup.

The redo phase implements careful error isolation to minimize the impact of individual page failures. If we encounter an error while applying a redo record to a specific page, we mark that page as corrupted but continue with redo operations for other pages. This allows us to recover as much of the database as possible even when some storage is damaged.

**Undo phase failures** require restarting the complete recovery process because the undo phase generates compensation log records that become part of the log. If undo fails partway through, these compensation records may be inconsistent with the actual database state. By restarting from the beginning, we ensure that the undo phase either completes entirely or not at all.

The undo phase failure handling is complicated by the fact that we may have already written some compensation log records before the failure occurred. When restarting recovery, we must be able to handle these previously written CLRs correctly. This is why CLRs include the "undo next LSN" field - they allow us to skip over previously undone operations during recovery restart.

**Recovery failure logging** requires special consideration because the normal logging system may be compromised. We maintain a separate recovery log file that uses a simpler, more robust format to record recovery progress and errors. This recovery log is essential for diagnosing recovery failures and determining what manual intervention steps are needed.

**⚠ Pitfall: Infinite Recovery Loops** If the recovery logic itself has a bug that causes consistent failures, restarting recovery repeatedly can create an infinite loop that prevents the database from ever becoming available. To prevent this, we implement a recovery attempt counter that limits the number of automatic retry attempts and requires manual intervention after repeated failures.

The recovery attempt counter is persisted in a simple text file separate from the main WAL to ensure it survives recovery restarts. After three consecutive recovery failures, the system refuses to attempt automatic recovery and requires an administrator to explicitly reset the counter after diagnosing and fixing the underlying problem.

## Implementation Guidance

### Technology Recommendations:

Component	Simple Option	Advanced Option
Checksums	CRC32 via <code>crc</code> crate	Hardware-accelerated CRC32c
Disk Space Monitoring	<code>std::fs::metadata</code>	Platform-specific <code>statvfs/GetDiskFreeSpace</code>
Error Handling	<code>Result&lt;T, WalError&gt;</code> with <code>thiserror</code>	Custom error types with backtrace
Logging	<code>log</code> crate with <code>env_logger</code>	Structured logging with <code>tracing</code>

### Recommended File Structure:

```

src/
  wal/
    error.rs          ← WalError types and conversion
    corruption.rs     ← Corruption detection utilities
    partial_write.rs  ← Partial write handling
    disk_monitor.rs  ← Disk space monitoring
    recovery_error.rs ← Recovery failure handling
    validation.rs    ← Data structure validation
    lib.rs            ← Re-export error handling API
  
```

### Core Error Types (Complete Implementation):

```
use thiserror::Error;

#[derive(Error, Debug)]

pub enum WalError {

    #[error("I/O error: {0}")]
    Io(#[from] std::io::Error),

    #[error("Corruption detected: {0}")]
    Corruption(String),

    #[error("Invalid record format: {0}")]
    InvalidRecord(String),

    #[error("Checkpoint operation in progress")]
    CheckpointInProgress,

    #[error("Log truncation failed: {0}")]
    TruncationFailed(String),

    #[error("Master record corrupted: {0}")]
    MasterRecordCorrupted(String),
}

pub type WalResult<T> = Result<T, WalError>;

// Disk space monitoring utilities

#[derive(Debug, Clone, Copy)]

pub enum DiskSpaceStatus {
```

```
Normal,  
Warning,  
Critical,  
Emergency,  
}  
  
pub struct DiskMonitor {  
  
    log_directory: PathBuf,  
  
    warning_threshold: u64,  
  
    critical_threshold: u64,  
  
    emergency_threshold: u64,  
  
    reserved_space: u64,  
}  
  
impl DiskMonitor {  
  
    pub fn new(log_directory: PathBuf, reserved_space: u64) -> Self {  
  
        Self {  
  
            log_directory,  
  
            warning_threshold: reserved_space * 10, // 10x reserved space  
  
            critical_threshold: reserved_space * 5, // 5x reserved space  
  
            emergency_threshold: reserved_space * 2, // 2x reserved space  
  
            reserved_space,  
        }  
    }  
  
    pub fn check_space(&self) -> WalResult<DiskSpaceStatus> {  
  
        // TODO: Implement platform-specific disk space checking  
  
        // TODO: Compare available space against thresholds
```

```
// TODO: Return appropriate status level  
todo!("Implement disk space monitoring")  
}  
}
```

### **Corruption Detection Utilities (Complete Implementation):**

```
use crc::Crc, CRC_32_ISO_HDLC;

const CRC32: Crc<u32> = Crc::new(&CRC_32_ISO_HDLC);

pub fn calculate_crc32(data: &[u8]) -> u32 {
    CRC32.checksum(data)
}

pub fn validate_checksum(data: &[u8], expected: u32) -> bool {
    let actual = calculate_crc32(data);
    actual == expected
}

pub struct CorruptionDetector {
    consecutive_errors: usize,
    max_consecutive_errors: usize,
    total_records_scanned: usize,
    corrupted_records: usize,
}

impl CorruptionDetector {
    pub fn new(max_consecutive_errors: usize) -> Self {
        Self {
            consecutive_errors: 0,
            max_consecutive_errors,
            total_records_scanned: 0,
            corrupted_records: 0,
        }
    }
}
```

```
pub fn check_record(&mut self, record_data: &[u8]) -> WalResult<()> {  
    // TODO: Validate record CRC checksum  
    // TODO: Check structural consistency (LSN sequence, etc.)  
    // TODO: Update error counters and decide if corruption is recoverable  
    // TODO: Return error if too many consecutive failures detected  
    todo!("Implement corruption detection logic")  
}  
}  
}
```

#### Partial Write Handler (Skeleton):

```
pub struct PartialWriteHandler {  
    recovery_mode: bool,  
    last_valid_lsn: Option<LSN>,  
    truncation_point: Option<u64>, // file offset  
}  
  
impl PartialWriteHandler {  
    pub fn scan_for_partial_writes(&mut self, log_reader: &mut LogReader) -> WalResult<Option<LSN>> {  
        // TODO: Scan log records from the end backwards  
        // TODO: Identify the last completely valid record  
        // TODO: Mark truncation point if partial write detected  
        // TODO: Return LSN of last valid record for recovery restart  
        todo!("Implement partial write detection")  
    }  
  
    pub fn truncate_log_at_partial_write(&mut self, log_file: &mut File) -> WalResult<()> {  
        // TODO: Truncate file at the truncation point identified during scan  
        // TODO: Ensure truncation is atomic and properly synced to disk  
        // TODO: Update internal state to reflect new end of log  
        todo!("Implement log truncation after partial write")  
    }  
}
```

### Recovery Error Handler (Skeleton):

```

pub struct RecoveryErrorHandler {
    recovery_attempts: u32,
    max_recovery_attempts: u32,
    last_successful_checkpoint: Option<LSN>,
    recovery_log_file: File,
}

impl RecoveryErrorHandler {
    pub fn handle_recovery_failure(&mut self, error: WalError, recovery_phase: &str) -> WalResult<bool> {
        // TODO: Log the recovery failure with detailed context
        // TODO: Increment recovery attempt counter and check limits
        // TODO: Determine if recovery should be retried or requires manual intervention
        // TODO: Return true if retry is recommended, false if manual intervention needed
        todo!("Implement recovery failure handling logic")
    }

    pub fn reset_to_previous_checkpoint(&mut self) -> WalResult<LSN> {
        // TODO: Find the previous valid checkpoint before the failed one
        // TODO: Reset all recovery state to checkpoint starting point
        // TODO: Return the checkpoint LSN to restart recovery from
        todo!("Implement checkpoint fallback logic")
    }
}

```

### Milestone Checkpoints:

After implementing error handling:

1. **Corruption Detection Test:** Create a log file with an invalid CRC and verify detection:

```
cargo test test_corruption_detection  
  
# Should detect corrupted records and handle gracefully
```

BASH

2. **Partial Write Simulation:** Test recovery from incomplete log records:

```
cargo test test_partial_write_recovery  
  
# Should truncate partial records and continue recovery
```

BASH

3. **Disk Full Handling:** Test graceful degradation when storage is exhausted:

```
cargo test test_disk_full_scenarios  
  
# Should enter read-only mode and complete active transactions
```

BASH

4. **Recovery Failure Testing:** Simulate recovery errors and verify restart behavior:

```
cargo test test_recovery_restart  
  
# Should restart from checkpoint after recovery failures
```

BASH

### Debugging Tips:

Symptom	Likely Cause	Diagnosis	Fix
"Corruption detected" during normal operation	Storage hardware failure or memory corruption	Check system logs, run disk diagnostics	Replace storage, restore from backup
Recovery hangs at same LSN repeatedly	Infinite loop in recovery logic or corrupted checkpoint	Check recovery logs, examine log records at stuck LSN	Reset recovery counter, use previous checkpoint
"Disk full" errors despite available space	Incorrect space calculation or reserved space exhaustion	Check actual vs. reported disk usage, verify space calculations	Adjust thresholds, increase reserved space
Database marked offline after recovery	Unrecoverable storage errors during redo phase	Examine redo failure logs, check storage health	Restore from backup or repair storage

# Testing Strategy

**Milestone(s):** All milestones - comprehensive testing approach spans Milestone 1 (Log Record Format) correctness verification, Milestone 2 (Log Writer) durability guarantees, Milestone 3 (Crash Recovery) ARIES algorithm validation, and Milestone 4 (Checkpointing) fuzzy checkpoint integrity

Testing a Write-Ahead Logging system presents unique challenges that go far beyond typical software testing. Think of it like testing a nuclear reactor's safety systems - you can't afford to discover bugs in production because the consequences (data loss, corruption, inconsistent state) are catastrophic. Unlike application logic where a bug might cause a feature to misbehave, WAL bugs can destroy the fundamental guarantee that committed data survives crashes. This means our testing strategy must be both comprehensive and aggressive, simulating every conceivable failure mode.

The core challenge is that WAL correctness depends on complex interactions between multiple subsystems under adverse conditions. A bug might only manifest when a crash occurs at exactly the wrong microsecond during a checkpoint operation, or when a partial write happens to corrupt a specific log record that's needed for recovery. Traditional unit testing, while necessary, is insufficient because it can't capture these emergent behaviors that arise from component interactions during failure scenarios.

Our testing strategy addresses this through four complementary approaches: **unit testing** to verify individual component correctness in isolation, **crash testing** to validate recovery behavior under every conceivable failure timing, **milestone checkpoints** to ensure steady progress and catch regressions early, and **property-based testing** to discover edge cases that human testers would never think to construct. Each layer builds on the previous one, creating a comprehensive safety net that gives us confidence the system will preserve data integrity under any conditions.

## Unit Testing

Unit testing in WAL systems requires a fundamentally different approach than typical application testing because we're testing stateful, persistent components that interact with the filesystem and must maintain invariants across crashes. Think of each component as a specialized machine in a factory assembly line - we need to test not just that each machine produces correct output given correct input, but that it handles malformed input gracefully, maintains its internal state consistently, and continues working correctly even after simulated power failures.

The key insight is that WAL components must be tested with **mock dependencies** that allow us to precisely control timing, inject failures, and verify internal state changes that would be impossible to observe in an integrated system. However, these mocks must faithfully simulate the behavior and failure modes of real dependencies, or our unit tests become useless - worse, they can give false confidence.

## Data Structure and Serialization Testing

The foundation of all WAL testing begins with verifying that our log record formats maintain integrity through serialization and deserialization cycles. This seems straightforward but contains subtle traps that can cause data corruption.

Test Category	Test Method	Verification Target	Common Pitfalls
Record Roundtrip	Serialize then deserialize each record type	Data preservation through byte conversion	Endianness issues, padding bytes, variable-length field boundaries
CRC Validation	Inject single-bit errors into serialized data	Corruption detection sensitivity	CRC collisions, checksum placement in record structure
Boundary Conditions	Test maximum and minimum field values	Field overflow and underflow handling	Integer overflow in LSN arithmetic, empty before/after images
Variable Length Fields	Records with zero, small, and maximum-size data	Dynamic field sizing and alignment	Buffer overruns, incorrect length prefixes, alignment assumptions

The most critical unit test for data structures is the **corruption injection test**. We serialize a valid record, flip individual bits throughout the serialized data, then attempt to deserialize and verify that corruption is detected. This test must achieve 100% detection rate for single-bit errors - if even one bit flip goes undetected, we have a serious flaw in our integrity checking.

## Mock Database Testing

Testing WAL components requires a controllable database mock that simulates page-based storage without the complexity of a real database engine. The `MockDatabase` serves this purpose, but it must be implemented carefully to avoid hiding bugs that would appear with real storage.

Mock Component	Responsibility	Key Behaviors to Simulate	Testing Interface
<code>MockDatabase</code>	Page storage and retrieval	LSN tracking per page, concurrent access patterns, page-level locking	<code>write_data()</code> , <code>get_page()</code> , <code>update_page()</code> with LSN validation
<code>MockFilesystem</code>	File I/O with controllable failures	Partial writes, fsync semantics, disk full conditions	Configurable failure injection at specific operation counts
<code>MockClock</code>	Time control for testing	Deterministic timestamps, timeout simulation	Manual time advancement for checkpoint interval testing

The `MockDatabase` must track the **Page LSN** for every page to simulate real database behavior where each page knows the LSN of the last log record that modified it. This is crucial for testing redo operations during recovery - a redo record should only be applied if the log record's LSN is greater than the page's current LSN.

```

MockDatabase behavior specification:
- write_data(page_id, offset, data, lsn): Update page data and set page LSN to the provided value
- get_page(page_id): Return page with current data and LSN, or None if page doesn't exist
- update_page(page): Store modified page, updating both data and LSN atomically
- LSN comparison: Page modifications must respect LSN ordering for idempotency testing

```

## Log Writer Component Testing

Testing the `LogWriter` requires careful simulation of filesystem behaviors, particularly around crash consistency and partial writes. The challenge is that real filesystems have complex behaviors around write ordering, `fsync` semantics, and failure atomicity that our tests must capture.

Test Scenario	Setup	Expected Behavior	Failure Indicators
Successful Append	Normal operation with sufficient disk space	Record written, LSN returned, data immediately readable	Write failures, incorrect LSN generation, buffer corruption
Force Sync Verification	Write records, call <code>force_sync()</code> , verify durability	All buffered data persisted before return	Data loss after simulated crash, incomplete records
Buffer Management	Write multiple small records rapidly	Records batched efficiently, buffer limits respected	Memory bloat, buffer fragmentation, incorrect flush timing
Segment Rotation	Fill segment to size limit, write additional record	New segment created, LSN sequence maintained across boundary	LSN gaps, rotation failures, incomplete segment headers

The most critical unit test for `LogWriter` is the **crash consistency test**. We write several records, call `force_sync()`, then simulate a crash by forcibly closing all file handles and reopening them. Every record that was force-synced must be readable and intact, while records that weren't synced may or may not be present (but if present, must be valid).

**⚠ Pitfall: Incomplete `fsync` Testing** Many developers test `fsync` by simply calling the method and checking that it returns successfully. This is insufficient - you must verify that the data survives simulated crashes. Use a mock filesystem that tracks which data has been force-synced versus merely written to buffers. Only force-synced data should survive crash simulation.

## Recovery Manager Component Testing

Testing the `RecoveryManager` requires constructing precise log scenarios that exercise each phase of ARIES recovery. Unlike other components, the recovery manager must work correctly on partially corrupted or incomplete logs that might result from crashes during normal operation.

Recovery Phase	Test Construction	Validation Points	Edge Cases
Analysis Pass	Create logs with committed, aborted, and active transactions	Transaction table and dirty page table correctly populated	Missing commit records, orphaned redo records, checkpoint inconsistencies
Redo Pass	Mix of committed transactions with overlapping page modifications	All committed changes applied, idempotency preserved	Duplicate redo application, LSN ordering violations, cross-transaction dependencies
Undo Pass	Active transactions with complex undo chains	All uncommitted changes rolled back in reverse LSN order	Missing undo records, compensation record generation, undo chain breaks

The key insight for testing recovery is that we must construct **synthetic crash scenarios** where the log contains exactly the pattern of records that would result from a crash at a specific point in time. This requires careful orchestration of multiple transactions with overlapping lifetimes and dependencies.

For example, to test undo chain traversal, we create a transaction that makes multiple modifications to the same page, then crashes before committing. The recovery test must verify that the undo pass walks backward through the transaction's log records and applies each undo operation in reverse order, ultimately restoring the page to its original state.

### Checkpoint Manager Component Testing

Testing checkpoints presents unique challenges because checkpoint correctness depends on capturing consistent snapshots of system state without blocking concurrent operations. The fuzzy checkpoint algorithm must be tested under various levels of concurrent activity.

Checkpoint Test	Concurrent Activity	Consistency Requirements	Failure Modes
Fuzzy Checkpoint Creation	Multiple active transactions modifying overlapping pages	Checkpoint LSN serves as consistent recovery boundary	Race conditions in active transaction capture, dirty page enumeration errors
Log Truncation Safety	Checkpoint completion with subsequent new transactions	No log records needed for recovery are truncated	Premature truncation, checkpoint LSN miscalculation
Master Record Atomicity	Checkpoint completion during system stress	Master record update is atomic and always points to valid checkpoint	Partial master record writes, corrupted master record format
Recovery from Checkpoint	Crash immediately after checkpoint completion	Recovery starts from checkpoint and processes subsequent log records	Checkpoint record corruption, incorrect dirty page restoration

The most complex checkpoint test is the **concurrent checkpoint test** where we run a fuzzy checkpoint while multiple transactions are actively modifying data. The test must verify that the checkpoint captures a temporally consistent view - all changes with LSNs less than the checkpoint LSN are reflected, and no changes with greater LSNs are included.

## Crash Testing

Crash testing is where WAL implementations prove their worth - or catastrophically fail. Think of crash testing as stress-testing a parachute: you can inspect the fabric and test individual components all you want, but ultimately you need to jump out of airplanes to know if it will save your life. In WAL systems, crashes are not exceptional events to be handled gracefully - they are the fundamental challenge the entire system is designed to solve.

The core principle of crash testing is **failure timing precision**. A bug that only appears when a crash occurs during the microsecond window between writing a log record and updating the corresponding database page will never be found by random testing. We need systematic approaches that crash the system at every possible point in every possible operation, then verify that recovery produces the correct result.

### Systematic Crash Point Enumeration

Effective crash testing requires identifying every point in the system where a crash could produce different outcomes, then explicitly testing each one. This goes far beyond simply killing processes at random intervals - we need surgical precision in crash timing.

Operation Type	Critical Crash Points	Recovery Expectation	Detection Method
Single Record Write	Before write, during write, after write but before fsync, after fsync	Pre-fsync writes may be lost, post-fsync writes must survive	Check record presence and integrity after recovery
Transaction Commit	Before commit record write, during commit record write, after commit record but before response	Incomplete commits must be rolled back, complete commits must be preserved	Verify transaction effects visible after recovery
Checkpoint Creation	During active transaction scan, during dirty page enumeration, during checkpoint record write	Partial checkpoints must not be used for recovery	Recovery must start from previous valid checkpoint
Log Rotation	During segment creation, during segment header write, during master record update	Log continuity must be maintained across segment boundaries	Verify LSN sequence integrity across rotation

The `CrashSimulator` component provides deterministic crash injection by maintaining a registry of crash points throughout the codebase. Each critical operation registers itself with the crash simulator, which can then trigger crashes at precisely defined moments.

`CrashSimulator` interface specification:

- `add_crash_point(operation_name)`: Register a potential crash location with unique identifier
- `set_current_operation(operation_name)`: Check if crash should be triggered at this point
- `Configuration`: Specify which operation names should trigger crashes for each test run
- `Deterministic`: Same configuration always produces same crash timing for reproducible tests

## Transaction Atomicity Crash Testing

The fundamental promise of WAL is that transactions are atomic - they either commit completely or have no effect. Crash testing must verify this property under every possible crash timing within transaction processing.

Test Scenario	Transaction Structure	Crash Timing	Expected Recovery Outcome
Single Page Transaction	One page modification with commit	Before commit record fsync	Transaction rolled back, page unchanged
Multi-Page Transaction	Multiple pages modified atomically	After some redo records written, before commit record	All transaction effects rolled back, all pages restored to original state
Overlapping Transactions	Two transactions modifying same page	Crash after T1 commits, before T2 commits	T1 effects visible, T2 effects completely absent
Long-Running Transaction	Many operations over extended time	Crash in middle of operation sequence	Partial transaction effects completely undone

The most revealing test is the **cross-transaction consistency test**. We run two transactions T1 and T2 that modify overlapping sets of pages, with T1 transferring "value" from pages that T2 reads. We crash at various points and verify that recovery never produces a state where T1's modifications are partially visible (which would violate transaction atomicity) or where T2 sees an inconsistent view of T1's changes.

### ARIES Recovery Algorithm Validation

Crash testing must verify that the three-phase ARIES recovery algorithm correctly handles every combination of committed and uncommitted transactions found in crash scenarios.

Recovery Phase	Test Construction	Validation Approach	Critical Properties
Analysis Phase Correctness	Logs with complex transaction histories, missing records, checkpoint records	Compare recovered transaction table and dirty page table against expected values	All committed transactions identified, all active transactions tracked, dirty pages correctly enumerated
Redo Phase Idempotency	Apply same redo log multiple times	Verify identical final state regardless of replay count	LSN-based redo skipping, proper page state progression
Undo Phase Completeness	Active transactions with deep modification chains	Verify all uncommitted changes fully reversed	Undo chain traversal, compensation log record generation

The **recovery correctness oracle** is a critical component for crash testing. Since we're testing that recovery produces correct results, we need an independent way to determine what the correct result should be. We maintain a **golden state** by tracking all committed transactions and their effects in a separate, crash-proof data structure, then compare the post-recovery database state against this golden state.

**⚠ Pitfall: Insufficient Crash Coverage** Many crash tests only crash at obvious points like "after writing log record" or "during fsync." Real bugs often occur at subtle points like "after acquiring lock but before writing record" or "after partial buffer flush but before segment rotation." Use code coverage tools to ensure your crash points exercise every code path that could leave the system in an inconsistent intermediate state.

### Checkpoint Recovery Crash Testing

Checkpoint-based recovery introduces additional complexity because recovery must correctly handle cases where crashes occur during checkpoint creation, leaving the system with partially written checkpoint records or inconsistent master records.

Checkpoint Crash Scenario	System State After Crash	Recovery Behavior	Correctness Criteria
Crash During Checkpoint Record Write	Partial checkpoint record in log	Recovery ignores incomplete checkpoint, uses previous valid checkpoint	Recovery completes successfully, no data loss, checkpoint boundaries respected
Crash During Master Record Update	Master record points to invalid or incomplete checkpoint	Recovery detects master record corruption, falls back to scanning for valid checkpoints	System recovers from most recent complete checkpoint
Crash After Checkpoint But Before Log Truncation	Complete checkpoint exists, old log segments still present	Recovery uses new checkpoint but can still process older log entries	No data loss, recovery efficiency improved but correctness maintained

The **checkpoint atomicity test** is particularly demanding. We create a checkpoint while multiple transactions are active, crash during the checkpoint creation process, then verify that recovery produces identical results whether it uses the incomplete checkpoint (if recovery is smart enough to detect and skip it) or falls back to the previous checkpoint.

### Partial Write and Corruption Testing

Real storage devices can fail in complex ways that produce partial writes, torn pages, and various forms of data corruption. Our crash testing must simulate these hardware-level failures to ensure the WAL system remains robust.

Failure Mode	Simulation Method	Detection Requirement	Recovery Strategy
Torn Page Writes	Write partial log records, corrupt record boundaries	CRC validation detects corruption	Truncate log at first invalid record, recover from that point
Disk Full During Write	Mock filesystem returns ENOSPC at specific points	Write operations fail gracefully	Reserve emergency space, complete critical operations (commit/abort) before failing
Silent Data Corruption	Flip bits in written data after successful write	CRC validation on read detects corruption	Identify corrupted records, determine safe truncation point
Filesystem Metadata Corruption	Simulate directory corruption, inode damage	Detect filesystem inconsistencies	Graceful failure, clear error reporting

## Milestone Checkpoints

Milestone checkpoints serve as integration points where we verify that the implementation correctly demonstrates the concepts being learned. Think of them as practical exams - unlike unit tests that check individual functions, milestone checkpoints verify that the student understands the broader architectural patterns and can implement them correctly in the context of a working system.

Each checkpoint builds incrementally on previous milestones while introducing new complexity. This progressive approach helps learners debug issues systematically rather than being overwhelmed by trying to diagnose problems in a fully integrated system where a bug in any component can manifest as mysterious failures elsewhere.

### Milestone 1: Log Record Format Checkpoint

The first milestone establishes the foundation - correct log record design and serialization. The checkpoint verifies that learners understand the fundamental data structures and can implement binary serialization correctly.

Verification Test	Input	Expected Output	Pass Criteria
Record Serialization Roundtrip	Create each LogRecord variant with sample data	Serialize then deserialize produces identical record	All field values preserved exactly, LSNs match, transaction IDs intact
CRC Integrity Checking	Serialize valid record, flip one bit, deserialize	CRC validation fails, returns appropriate error	100% single-bit error detection, specific corruption error type returned
Variable Length Field Handling	Create RedoRecord and UndoRecord with various image sizes	Successful serialization and parsing regardless of image size	Zero-byte images work, maximum-size images work, length prefixes correct
LSN Monotonicity	Create sequence of records with increasing LSNs	LSN extraction and comparison works correctly	LSN ordering maintained, comparison functions work for recovery algorithms

The **integration test** for Milestone 1 involves creating a complete transaction log scenario: begin transaction, multiple redo records for different pages, commit record. Serialize the entire sequence to bytes, then deserialize and verify that the transaction structure is perfectly reconstructed.

**Milestone 1 Integration Test Scenario:**

1. Transaction T1 modifies pages P1, P2, P3 with specific before/after images
2. Serialize complete transaction log (4 records: 3 redo + 1 commit)
3. Deserialize from bytes and verify transaction completeness
4. Inject corruption in different records and verify detection
5. Test LSN ordering and transaction ID consistency across records

## Milestone 2: Log Writer Checkpoint

The second milestone introduces persistence and durability concepts. The checkpoint verifies that learners understand append-only semantics, fsync timing, and crash consistency.

Verification Test	Operation Sequence	Crash/Restart Simulation	Recovery Validation
Durability Guarantee	Write record, force_sync(), simulate crash	Kill process, restart, scan log	Force-synced record present and intact
Buffer Management	Write many small records rapidly	Normal shutdown, restart, scan log	Records batched efficiently, no corruption, proper ordering
Segment Rotation	Fill segment to size limit, write additional records	Continue writing across rotation boundary	LSN continuity maintained, all records readable across segment files
Concurrent Write Safety	Multiple threads writing simultaneously	Various crash timings during concurrent writes	No record corruption, atomicity of individual record writes

The critical **durability test** involves writing a sequence of records with strategic fsync points, then simulating crashes at various times. Only records that were explicitly force-synced should survive the crash - any record written but not synced may or may not be present, but if present must be valid.

**Milestone 2 Durability Test Sequence:**

1. Write records R1, R2, R3 (no fsync)
2. Call force\_sync() - all three records now durable
3. Write records R4, R5 (no fsync)
4. Simulate crash
5. Restart and scan log: R1, R2, R3 must be present; R4, R5 may be absent
6. If R4 is present, it must be valid; if R5 is present, R4 must also be present (write ordering)

### Milestone 3: Crash Recovery Checkpoint

The third milestone implements the core ARIES recovery algorithm. This is the most complex checkpoint because it requires understanding the interaction between all three recovery phases and their correctness properties.

Recovery Scenario	Pre-Crash Transaction State	Post-Recovery Expected State	Algorithm Verification Points
Simple Committed Transaction	T1: 3 page modifications, commit record written	T1 effects visible on all pages	Analysis identifies T1 as committed, redo applies all T1 records, undo skips T1
Active Transaction Rollback	T2: 5 page modifications, no commit record	T2 effects completely absent	Analysis identifies T2 as active, redo may apply T2 records, undo completely reverses T2
Mixed Transaction Recovery	T1 committed, T2 active, T3 aborted	T1 visible, T2 and T3 absent	Correct transaction state classification, proper redo/undo application
Cross-Transaction Page Dependencies	T1 and T2 both modify page P1 in overlapping timeframes	Final page state reflects committed transactions only	LSN ordering respected, redo idempotency maintained

The **comprehensive recovery test** creates a complex scenario with multiple overlapping transactions, crashes at a strategic point, then verifies that ARIES recovery produces the correct final state. This test validates the entire recovery algorithm stack.

**Milestone 3 Comprehensive Recovery Test:**

1. Start transactions T1, T2, T3
2. T1 modifies pages A, B; T2 modifies pages B, C; T3 modifies pages A, C
3. T1 commits (commit record written and synced)
4. T3 aborts (abort record written and synced)
5. T2 continues with additional modifications to page D
6. Crash occurs (T2 is active, no commit record)
7. Recovery runs: Analysis, Redo, Undo phases
8. Verify final state: T1 effects visible, T2 and T3 effects completely absent
9. Check page LSNs reflect only committed transactions
10. Verify transaction table and dirty page table correctly reconstructed

#### Milestone 4: Checkpointing Checkpoint

The final milestone adds checkpointing to bound recovery time. The checkpoint verifies that learners understand fuzzy checkpoints and can implement log truncation safely.

Checkpoint Test	Checkpoint Timing	Recovery Performance	Safety Verification
Fuzzy Checkpoint Creation	During active transaction processing	Recovery starts from checkpoint, not beginning of log	Active transactions at checkpoint time correctly captured
Log Truncation Safety	After checkpoint completion and new activity	Recovery still works correctly with truncated logs	No required log records truncated, recovery boundaries respected
Master Record Atomicity	Checkpoint creation under system load	Master record always points to valid checkpoint	No partial master record updates, atomic master record replacement
Recovery Performance Improvement	Before/after checkpoint creation	Measurable reduction in recovery time	Recovery time bounded by checkpoint interval, not total log size

The **end-to-end checkpoint test** runs the complete system through multiple checkpoint cycles while processing transactions, crashes at various points, and verifies both correctness and performance characteristics.

**Milestone 4 End-to-End Test:**

1. Process transactions for extended period, creating substantial log
2. Create first checkpoint C1
3. Continue processing, create checkpoint C2
4. Truncate logs based on C1 safety
5. Crash system at random point after C2
6. Measure recovery time - should be bounded by activity since C2, not total history
7. Verify correctness - all committed transactions since C2 recovered correctly
8. Confirm log truncation didn't affect recovery correctness

## Property-Based Testing

Property-based testing represents the cutting edge of WAL validation because it can discover subtle bugs that would never occur to human testers. Think of it as having a malicious adversary who understands your system deeply and crafts the most devious possible inputs to break your assumptions. Instead of testing specific scenarios, property-based testing generates thousands of randomized scenarios and verifies that certain invariant properties hold regardless of the specific input sequence.

The power of property-based testing in WAL systems comes from its ability to explore the vast space of possible transaction interleavings, crash timings, and data patterns that could expose race conditions, edge cases in recovery logic, or subtle violations of ACID properties. A human tester might create a scenario with two transactions modifying three pages - a property-based test might generate scenarios with hundreds of transactions, complex dependency chains, and bizarre edge cases that stress-test every assumption in the implementation.

## WAL Correctness Properties

The foundation of property-based testing is defining **invariant properties** that must hold regardless of the specific sequence of operations or crash timings. These properties encode the fundamental correctness requirements of the WAL system.

Property Category	Invariant Statement	Verification Method	Violation Symptoms
Durability	All committed transactions survive crashes	Generate random transaction workload, crash at random points, verify committed transactions present after recovery	Committed transaction effects missing after recovery
Atomicity	Transactions are never partially visible	Generate transactions with multiple operations, crash during execution, verify all-or-nothing visibility	Some but not all operations of a transaction visible after crash
Consistency	Recovery always produces valid database state	Generate concurrent transactions, crash randomly, verify database constraints satisfied after recovery	Constraint violations, impossible state combinations after recovery
Isolation	Concurrent transactions don't interfere	Generate overlapping transactions with conflicting operations, verify serializable outcomes	Non-serializable anomalies, lost updates, dirty reads after recovery

The **durability property** is the most fundamental: `forall workload, forall crash_point: committed(transaction) -> survives_recovery(transaction)`. Property-based testing generates thousands of random workloads with different transaction structures, crashes at random points during execution, and verifies that every transaction that successfully committed is still visible after recovery.

## Randomized Workload Generation

Effective property-based testing requires sophisticated workload generators that can create realistic but stress-inducing scenarios. The generator must produce transaction patterns that are likely to expose bugs while still being representative of real-world usage.

Generator Component	Purpose	Configuration Parameters	Bias Toward Bug Discovery
Transaction Structure Generator	Create transactions with realistic operation patterns	Max operations per transaction, page access patterns, data size distributions	Bias toward transactions that modify overlapping pages, create undo chains
Concurrency Pattern Generator	Control timing and overlap of concurrent transactions	Max concurrent transactions, arrival rate distributions, duration variations	Bias toward high contention scenarios, race condition triggers
Data Pattern Generator	Generate realistic before/after images for operations	Data size ranges, content patterns, compression characteristics	Include edge cases: empty data, maximum sizes, highly compressible vs random data
Crash Timing Generator	Determine when to simulate crashes during workload execution	Crash probability distributions, operation-based vs time-based triggers	Higher crash probability during critical operations like commits and checkpoints

The **transaction interdependency generator** is particularly important for discovering subtle bugs. It creates transactions that have complex relationships - for example, T1 writes a value that T2 reads and transforms, creating a dependency chain that recovery must preserve. If recovery applies these transactions in the wrong order or only partially applies some of them, the interdependency violations become apparent.

**Advanced Transaction Pattern Examples:**

- Transfer Pattern: T1 decreases page A, increases page B by same amount (atomicity test)
- Chain Pattern: T1 writes X, T2 reads X and writes Y, T3 reads Y (dependency test)
- Conflict Pattern: T1 and T2 both modify same page with different values (isolation test)
- Long-Running Pattern: Transaction with many operations over extended time (undo chain test)

## Crash Timing Randomization

The timing of crashes relative to WAL operations is crucial for discovering bugs. Property-based testing must systematically explore crash timings that are most likely to expose consistency violations.

Crash Timing Strategy	Target Scenarios	Bug Discovery Focus	Implementation Approach
Operation-Boundary Crashes	Crash between logical operations	Partial operation visibility, intermediate state exposure	Hook into operation start/end points, inject crashes with configured probability
Critical-Section Crashes	Crash during mutex-protected regions	Lock state corruption, resource leaks	Identify critical sections, bias crash probability higher within them
I/O Operation Crashes	Crash during file writes, fsyncs	Partial write detection, durability guarantee violations	Hook into filesystem operations, simulate various partial write scenarios
Recovery Phase Crashes	Crash during recovery itself	Recovery restart handling, nested failure scenarios	Inject crashes during analysis, redo, and undo phases with different probabilities

The **recovery phase crash testing** is particularly devious because it tests the system's ability to handle crashes during crash recovery itself. This can expose bugs in recovery restart logic, incomplete cleanup of recovery state, or assumptions about recovery running to completion.

## Property Verification Oracles

Property-based testing requires **oracles** - independent mechanisms for determining whether the system behavior is correct. Unlike unit tests where we specify exact expected outputs, property testing requires more abstract correctness criteria that can evaluate any possible system state.

Oracle Type	Verification Approach	Independence Guarantee	Correctness Criteria
Golden State Oracle	Maintain separate, simple implementation of committed state	Completely separate codebase, different algorithms	Post-recovery state matches independently tracked committed transactions
Invariant Checking Oracle	Verify database-level consistency constraints	Built-in constraint validation, independent of WAL implementation	All declared constraints satisfied after any recovery
Serializability Oracle	Check that concurrent execution is equivalent to some serial execution	Graph-based dependency analysis, independent of transaction execution order	No serialization anomalies detectable in final state
Idempotency Oracle	Verify that recovery can be run multiple times safely	Apply recovery repeatedly to same log, verify identical outcomes	Multiple recovery passes produce identical final state

The **golden state oracle** is the most reliable but also most expensive approach. It maintains a completely separate, simplified version of the database that tracks only committed transactions using straightforward, obviously correct algorithms. After any crash and recovery scenario, the WAL system's state is compared against the golden state to verify correctness.

**⚠ Pitfall: Weak Property Specifications** Many developers write property tests that are too weak to catch bugs - for example, testing that "recovery completes without errors" instead of testing that "recovery produces the correct final state." Strong properties require substantial oracle infrastructure to verify, but weak properties provide false confidence. Invest in building robust oracles that can definitively determine correctness.

### Shrinking and Minimal Bug Examples

When property-based testing discovers a bug, the initial failure case is often a complex scenario with hundreds of operations that makes debugging difficult. **Shrinking** automatically reduces the failing test case to a minimal example that still demonstrates the bug.

Shrinking Dimension	Reduction Strategy	Preservation Requirement	Debugging Benefit
Transaction Count	Remove transactions that don't contribute to the bug	Bug still reproduces with fewer transactions	Easier to trace through fewer transaction interactions
Operations Per Transaction	Remove operations that don't affect the bug manifestation	Core bug-triggering operation sequence preserved	Simpler transaction structure to analyze
Data Size	Reduce before/after image sizes	Bug reproduction independent of specific data content	Faster test execution, easier log inspection
Crash Timing	Find earliest crash point that still triggers bug	Bug manifestation timing preserved	Precise identification of problematic code paths

Effective shrinking for WAL systems requires understanding the **dependency structure** of the failing scenario. For example, if the bug involves incorrect undo chain traversal, shrinking must preserve the transaction structure that creates the problematic undo chain, but can eliminate unrelated transactions and operations.

The **minimal failing example** produced by shrinking becomes a regression test that runs quickly and makes the bug's root cause obvious. This bridges the gap between property-based discovery and targeted debugging - property testing finds the bug, shrinking isolates it, and traditional debugging techniques can then fix it efficiently.

## Implementation Guidance

The testing strategy implementation requires sophisticated infrastructure that goes beyond simple unit test frameworks. We need specialized tools for crash simulation, property-based test generation, and correctness verification that are tailored to the unique challenges of WAL systems.

### A. Technology Recommendations Table:

Component	Simple Option	Advanced Option
Unit Testing Framework	<code>cargo test</code> with built-in assert macros	<code>proptest</code> for property-based testing with custom generators
Crash Simulation	Manual process killing with <code>SIGKILL</code>	Custom crash injection framework with precise timing control
Mock Filesystem	In-memory HashMap simulating file operations	<code>tempfile</code> with configurable failure injection and partial write simulation
Property Test Generation	Simple randomized inputs with <code>rand crate</code>	<code>proptest</code> with sophisticated shrinking and custom strategies
Correctness Oracles	Manual state comparison in test assertions	Separate golden-state implementation with automatic comparison
Performance Measurement	Basic <code>std::time::Instant</code> timing	<code>criterion</code> benchmarking framework with statistical analysis

### B. Recommended File/Module Structure:

```
wal-implementation/
src/
  lib.rs           ← Public API and core types
  log_record.rs   ← LogRecord enum and serialization
  log_writer.rs   ← LogWriter component
  recovery_manager.rs   ← RecoveryManager component
  checkpoint_manager.rs   ← CheckpointManager component
  error.rs         ← WalError enum and utilities
tests/
  unit/
    test_log_record.rs   ← Component-level unit tests
    test_log_writer.rs   ← Data structure and serialization tests
    test_recovery_manager.rs   ← Log writer durability tests
    test_checkpoint_manager.rs   ← Recovery algorithm tests
    test_crash_recovery.rs   ← Checkpoint creation tests
  integration/
    test_crash_recovery.rs   ← Cross-component integration tests
    test_milestone_checkpoints.rs   ← End-to-end crash scenarios
    test_milestone_checkpoints.rs   ← Milestone validation tests
  property/
    test_recovery_properties.rs   ← Property-based tests
    generators.rs   ← Correctness property verification
    oracles.rs     ← Custom workload generators
    oracles.rs     ← Correctness checking oracles
testing_infrastructure/
  crash_simulator.rs   ← Reusable testing utilities
  mock_database.rs   ← Crash injection framework
  golden_state.rs   ← Controllable database mock
  workload_generator.rs   ← Independent correctness oracle
  workload_generator.rs   ← Transaction pattern generator
```

## C. Infrastructure Starter Code:

```
// testing_infrastructure/crash_simulator.rs

use std::sync::{Arc, Mutex};

use std::collections::HashSet;

/// Deterministic crash injection for testing WAL crash consistency

pub struct CrashSimulator {

    crash_points: Arc<Mutex<HashSet<String>>>,
    current_operation: Arc<Mutex<String>>,
}

impl CrashSimulator {

    pub fn new() -> Self {
        CrashSimulator {
            crash_points: Arc::new(Mutex::new(HashSet::new())),
            current_operation: Arc::new(Mutex::new(String::new())),
        }
    }

    /// Configure this simulator to crash when the specified operation is reached

    pub fn add_crash_point(&self, operation: &str) {
        self.crash_points.lock().unwrap().insert(operation.to_string());
    }

    /// Check if we should crash at this operation point

    /// Call this at strategic points in WAL operations

    pub fn set_current_operation(&self, operation: &str) {
        *self.current_operation.lock().unwrap() = operation.to_string();
    }
}
```

```
    if self.crash_points.lock().unwrap().contains(operation) {

        // Simulate crash by exiting immediately

        std::process::exit(1);

    }

}

/// Clear all crash points (useful for test cleanup)

pub fn reset(&self) {

    self.crash_points.lock().unwrap().clear();

    *self.current_operation.lock().unwrap() = String::new();

}

}

// testing_infrastructure/mock_database.rs

use std::collections::HashMap;

use std::sync::{Arc, RwLock};

use crate::{PageId, DatabasePage, LSN};

/// Mock database that simulates page-based storage with LSN tracking

/// Provides controllable behavior for testing WAL operations

pub struct MockDatabase {

    pages: Arc<RwLock<HashMap<PageId, DatabasePage>>,

    page_size: usize,

}

impl MockDatabase {

    pub fn new(page_size: usize) -> Self {

        MockDatabase {
```

```
    pages: Arc::new(RwLock::new(HashMap::new())),
    page_size,
}

}

/// Write data to a page at specified offset with LSN tracking
/// This simulates how real databases update pages and track modification LSNs
pub fn write_data(&self, page_id: PageId, offset: usize, data: &[u8], lsn: LSN) {

    let mut pages = self.pages.write().unwrap();

    let page = pages.entry(page_id).or_insert_with(|| DatabasePage {
        page_id,
        data: vec![0; self.page_size],
        lsn: 0,
    });

    // Verify write bounds
    assert!(offset + data.len() <= self.page_size, "Write exceeds page boundary");

    // Apply the write and update page LSN
    page.data[offset..offset + data.len()].copy_from_slice(data);
    page.lsn = lsn;
}

/// Retrieve page data and current LSN
pub fn get_page(&self, page_id: PageId) -> Option<DatabasePage> {
    self.pages.read().unwrap().get(&page_id).cloned()
}
```

```
/// Update entire page (used for recovery testing)

pub fn update_page(&self, page: DatabasePage) {
    self.pages.write().unwrap().insert(page.page_id, page);
}

/// Get all page IDs (useful for checkpoint testing)

pub fn all_page_ids(&self) -> Vec<PageId> {
    self.pages.read().unwrap().keys().copied().collect()
}

/// Clear all pages (test cleanup)

pub fn clear(&self) {
    self.pages.write().unwrap().clear();
}

}

// testing_infrastructure/golden_state.rs

use std::collections::{HashMap, BTreeMap};

use crate::{TransactionId, PageId, LSN};

/// Independent implementation of committed transaction state

/// Used as correctness oracle for property-based testing

pub struct GoldenState {

    committed_transactions: BTreeMap<LSN, TransactionId>,
    page_state: HashMap<PageId, Vec<u8>>,
    transaction_effects: HashMap<TransactionId, Vec<PageOperation>>,
}
```

```
#[derive(Debug, Clone)]  
  
pub struct PageOperation {  
  
    pub page_id: PageId,  
  
    pub offset: usize,  
  
    pub data: Vec<u8>,  
  
}  
  
impl GoldenState {  
  
    pub fn new() -> Self {  
  
        GoldenState {  
  
            committed_transactions: BTreeMap::new(),  
  
            page_state: HashMap::new(),  
  
            transaction_effects: HashMap::new(),  
  
        }  
    }  
  
    /// Record a transaction operation (before commit)  
  
    pub fn record_operation(&mut self, txn_id: TransactionId, page_id: PageId,  
  
                           offset: usize, data: Vec<u8>) {  
  
        let effects = self.transaction_effects.entry(txn_id).or_insert_with(Vec::new);  
  
        effects.push(PageOperation { page_id, offset, data });  
    }  
  
    /// Commit a transaction and apply its effects to the golden state  
  
    pub fn commit_transaction(&mut self, txn_id: TransactionId, commit_lsn: LSN) {  
  
        self.committed_transactions.insert(commit_lsn, txn_id);  
    }  
}
```

```

    if let Some(operations) = self.transaction_effects.remove(&txn_id) {
        for op in operations {
            let page = self.page_state.entry(op.page_id).or_insert_with(|| vec![0; 4096]);
            page[op.offset..op.offset + op.data.len()].copy_from_slice(&op.data);
        }
    }
}

/// Abort a transaction (remove its effects without applying)
pub fn abort_transaction(&mut self, txn_id: TransactionId) {
    self.transaction_effects.remove(&txn_id);
}

/// Compare current state against recovered database state
pub fn verify_matches(&self, recovered_db: &MockDatabase) -> Result<(), String> {
    for (page_id, expected_data) in &self.page_state {
        match recovered_db.get_page(*page_id) {
            Some(page) => {
                if &page.data != expected_data {
                    return Err(format!("Page {} data mismatch", page_id));
                }
            }
            None => return Err(format!("Page {} missing after recovery", page_id)),
        }
    }
    Ok(())
}

```

```
    }
```

```
}
```

#### D. Core Logic Skeleton Code:

```
// tests/integration/test_crash_recovery.rs

use crate::testing_infrastructure::{CrashSimulator, MockDatabase, GoldenState};

use crate::{LogWriter, RecoveryManager, TransactionCoordinator};

/// Comprehensive crash recovery test that validates ARIES algorithm correctness

/// This is the key integration test for Milestone 3

#[test]

fn test_comprehensive_crash_recovery() {

    // TODO 1: Set up test environment with MockDatabase, LogWriter, RecoveryManager

    // TODO 2: Create multiple overlapping transactions with different commit states

    // TODO 3: Configure CrashSimulator to crash at strategic point during execution

    // TODO 4: Execute transaction workload until crash occurs

    // TODO 5: Restart system and run recovery

    // TODO 6: Verify recovered state matches GoldenState oracle

    // TODO 7: Confirm all ARIES phases (Analysis, Redo, Undo) executed correctly

    // Hint: Use GoldenState to track what the correct final state should be

    // Hint: Crash timing should be after some commits but during active transaction

}

/// Property-based test that verifies durability property holds for all workloads

/// This validates the fundamental WAL guarantee across randomized scenarios

#[cfg(feature = "proptest")]

#[proptest]

fn test_durability_property(workload: Vec<TransactionPattern>) {

    // TODO 1: Set up fresh WAL system for each test case

    // TODO 2: Execute the generated transaction workload
```

```
// TODO 3: At random point, simulate crash and restart

// TODO 4: Run recovery and verify all committed transactions survived

// TODO 5: Use GoldenState oracle to verify correctness

// TODO 6: Ensure no uncommitted transactions are visible

// Hint: TransactionPattern should be generated by custom proptest strategy

// Hint: Crash point should also be randomized within workload execution

}

// tests/unit/test_log_writer.rs

/// Test that force_sync() guarantees durability across crashes

/// This is critical for Milestone 2 correctness

#[test]

fn test_force_sync_durability() {

    // TODO 1: Create LogWriter with temporary directory

    // TODO 2: Write several log records without calling force_sync()

    // TODO 3: Call force_sync() on specific records

    // TODO 4: Simulate crash by dropping LogWriter and reopening files

    // TODO 5: Verify only force-synced records are readable after crash

    // TODO 6: Test that partial writes are handled correctly

    // Hint: Use std::fs::OpenOptions with create/append flags

    // Hint: Simulate crash by calling std::mem::drop() then reopening files

}

/// Test that segment rotation maintains LSN continuity

#[test]
```

```

fn test_segment_rotation_continuity() {

    // TODO 1: Configure LogWriter with small segment size limit

    // TODO 2: Write records until segment rotation occurs

    // TODO 3: Continue writing across segment boundary

    // TODO 4: Verify LSN sequence is maintained across files

    // TODO 5: Test reading records across segment boundaries

    // TODO 6: Verify segment metadata is correctly maintained

    // Hint: Check that LSN gaps don't occur at segment boundaries

    // Hint: Verify both segments can be read sequentially

}

```

## E. Language-Specific Hints:

- **File I/O:** Use `std::fs::OpenOptions` with `.create(true).append(true)` for append-only log files
- **Durability:** Call `file.sync_all()` to implement fsync semantics in Rust
- **Serialization:** Use `bincode` crate for efficient binary serialization of log records
- **Error Handling:** Define custom `WalError` enum with `#[derive(Debug, thiserror::Error)]`
- **Concurrency:** Use `Arc<Mutex<T>>` for shared mutable state, `Arc<RwLock<T>>` for reader-writer scenarios
- **Testing:** Use `tempfile::TempDir` for isolated test directories that auto-cleanup
- **Property Testing:** Add `proptest = "1.0"` to Cargo.toml for advanced property-based testing
- **Benchmarking:** Use `criterion = "0.4"` for performance measurement of critical paths

## F. Milestone Checkpoint Commands:

BASH

```
# Milestone 1: Log Record Format

cargo test test_log_record --features=milestone1

# Expected: All serialization roundtrip tests pass, CRC validation working

# Milestone 2: Log Writer

cargo test test_log_writer --features=milestone2

# Expected: Durability tests pass, segment rotation working, buffer management correct

# Milestone 3: Crash Recovery

cargo test test_crash_recovery --features=milestone3

# Expected: ARIES recovery phases working, transaction atomicity preserved

# Milestone 4: Checkpointing

cargo test test_checkpoint --features=milestone4

# Expected: Fuzzy checkpoints created, recovery time bounded, log truncation safe

# Property-based testing (all milestones)

cargo test --features=proptest

# Expected: Thousands of randomized scenarios pass, no property violations found

# Performance benchmarks

cargo bench

# Expected: Recovery time scales with recent activity, not total log size
```

## G. Debugging Tips:

Symptom	Likely Cause	How to Diagnose	Fix
Recovery hangs indefinitely	Infinite loop in undo chain traversal	Add logging to recovery phases, check for circular undo references	Validate undo chain construction, add cycle detection
Committed transaction missing after recovery	Force sync not called before reporting commit	Check fsync timing in transaction commit path	Ensure commit record is force-synced before responding to client
Partial transaction visible after crash	Redo applied but undo skipped incorrectly	Examine transaction table construction during analysis phase	Fix transaction state classification logic
Checkpoint creation blocks transactions	Checkpoint not properly fuzzy	Add logging to checkpoint state transitions	Implement non-blocking checkpoint data collection
Property test failures hard to debug	Generated test cases too complex	Implement custom shrinking strategies	Add shrinking that preserves bug-triggering structure
CRC validation fails on valid records	Endianness or padding issues in serialization	Hexdump serialized records, check byte alignment	Use explicit endianness, pack structs properly

## Debugging Guide

**Milestone(s):** All milestones - debugging strategies span Milestone 1 (Log Record Format) corruption detection and serialization issues, Milestone 2 (Log Writer) partial write handling and fsync timing, Milestone 3 (Crash Recovery) ARIES recovery failures and consistency violations, and Milestone 4 (Checkpointing) checkpoint state machine errors and truncation safety

Building a Write-Ahead Logging system is like constructing a sophisticated flight recorder that must work flawlessly even when everything else is failing. Just as aviation investigators need specialized tools and techniques to extract meaning from damaged black box recordings, WAL implementers need systematic approaches to diagnose and fix the subtle bugs that can compromise crash safety and data integrity. The debugging challenge is amplified because WAL bugs often manifest only during crash scenarios or under specific timing conditions that are difficult to reproduce in normal testing.

The debugging process for WAL systems requires understanding both the theoretical guarantees (what should happen according to ARIES principles) and the practical implementation details (what actually happens when filesystem calls interact with kernel buffers, disk controllers, and hardware failures). Unlike application-level bugs that might cause incorrect results or performance issues, WAL bugs can silently corrupt data or violate

durability guarantees, making them particularly insidious and requiring proactive detection strategies rather than reactive debugging after problems are discovered.

## Common Symptoms and Causes

WAL implementations exhibit characteristic failure patterns that experienced developers learn to recognize. These symptoms often appear during stress testing, crash simulation, or production incidents where the WAL's crash recovery mechanisms are actually exercised. Understanding the relationship between observable symptoms and underlying causes enables faster diagnosis and prevents the trial-and-error debugging that can waste days of development time.

The most challenging aspect of WAL debugging is that many issues only manifest during recovery after a simulated or real crash. This means that bugs can remain dormant in the codebase for extended periods until specific crash timing or data patterns trigger them. The following symptom-to-cause mapping represents the most common failure modes encountered during WAL development and the systematic approaches to identify their root causes.

## Recovery Failures and Data Consistency Issues

Recovery failures represent the most critical category of WAL bugs because they directly violate the system's core durability guarantees. These issues typically manifest when the recovery process encounters unexpected log states or when the three-phase ARIES algorithm fails to reconstruct consistent database state.

Symptom	Likely Root Cause	Diagnostic Approach	Primary Fix Strategy
Recovery hangs during analysis pass	Infinite loop in log scanning due to corrupted LSN chain	Examine log file with hex editor around last processed LSN; check for LSN gaps or backwards jumps	Add LSN monotonicity validation in log reader with circuit breaker
Database state inconsistent after recovery	Redo operations applied out of LSN order or missing undo operations	Compare recovered pages against golden state oracle; trace LSN ordering in transaction table	Implement strict LSN ordering enforcement in redo pass
Recovery reports "missing checkpoint" but checkpoint exists	Master record corruption or checkpoint LSN mismatch	Validate master record CRC; compare checkpoint_Lsn against actual log contents	Implement master record validation with fallback to log scan
Transactions committed but missing after recovery	Commit records written but not fsynced before crash	Trace fsync timing relative to commit acknowledgments; check for premature commit responses	Ensure strict write-ahead logging: fsync before commit acknowledgment
Active transactions incorrectly marked as committed	Transaction table corruption during analysis pass	Dump transaction table state after analysis; verify against actual commit records in log	Fix transaction state tracking logic in analysis pass
Recovery fails with "invalid record type"	Log record corruption or serialization bugs	Parse log records manually; check for proper enum variant serialization	Add comprehensive record validation with type safety checks
Redo operations fail with "page not found"	Page allocation tracking inconsistency or missing page creation records	Audit page lifecycle in log; verify page creation precedes all modifications	Implement page existence validation before applying redo operations
Undo operations generate incorrect CLRs	Before-image corruption or undo chain traversal bugs	Trace undo chain manually; verify before-image integrity against known good state	Fix undo chain navigation and CLR generation logic

**⚠ Pitfall: Ignoring Recovery Testing** Many developers focus on normal operation testing but skip comprehensive crash recovery testing. Recovery bugs can lurk undetected until production crashes occur. The fix is implementing systematic crash injection testing at every point in the write transaction flow, not just at convenient boundaries.

## Log Writer and Durability Violations

Log writer issues often manifest as durability violations where committed transactions are lost after crashes, or as performance problems where excessive fsync calls create bottlenecks. These bugs frequently involve the subtle interaction between application-level buffering, filesystem caches, and actual disk persistence.

Symptom	Likely Root Cause	Diagnostic Approach	Primary Fix Strategy
Committed transactions lost after crash	fsync not called before commit acknowledgment	Add logging around fsync calls; measure time between log write and fsync	Implement strict force-write semantics with fsync before commit response
Log corruption after partial segment writes	Concurrent writers without proper serialization	Check for race conditions in log append operations; verify buffer pool thread safety	Implement atomic log append with proper locking or lock-free queues
Performance degradation with high throughput	Excessive fsync calls blocking subsequent writes	Profile fsync frequency and timing; measure write amplification	Implement group commit optimization batching multiple transactions per fsync
Log segments not rotating despite size limits	Segment size calculation bugs or rotation trigger failures	Monitor actual file sizes versus reported current_size values	Fix segment size tracking and rotation trigger logic
Buffer pool exhaustion under load	Memory leak in buffer management or incorrect buffer reuse	Monitor buffer allocation patterns; check for unreturned buffers	Implement proper buffer lifecycle management with leak detection
Log records appear out of order	Race condition in LSN assignment versus actual write order	Trace LSN assignment timing relative to physical write completion	Ensure LSN assignment happens atomically with log append operation
Disk full errors not handled gracefully	Missing disk space monitoring or reservation mechanisms	Monitor available disk space during high write loads	Implement disk space monitoring with reserved emergency space
Log write hangs indefinitely	Filesystem deadlock or hardware failure	Check system logs for I/O errors; test with different filesystem configurations	Add write timeout detection with fallback error handling

**⚠ Pitfall: Trusting Filesystem Guarantees** Developers often assume that successful file writes are durable, but data may remain in filesystem caches until fsync is called. The fix is implementing explicit durability checkpoints with fsync timing validation, not relying on filesystem default behaviors.

## Serialization and Data Format Issues

Serialization bugs can cause subtle data corruption that only becomes apparent during recovery when log records are parsed and applied. These issues often involve endianness, variable-length field encoding, or CRC checksum calculation errors.

Symptom	Likely Root Cause	Diagnostic Approach	Primary Fix Strategy
CRC validation failures on valid records	Incorrect CRC calculation or field ordering during serialization	Compare calculated CRC against expected value; verify field serialization order	Fix CRC calculation to match exact serialization byte order
Deserialization fails with "unexpected EOF"	Variable-length field encoding bugs or length prefix corruption	Parse record manually with hex editor; verify length prefixes match actual field sizes	Implement robust length validation with bounds checking
LSN values appear to jump backwards	Endianness issues on different architectures or LSN counter overflow	Test on different architectures; check for integer overflow in LSN generation	Use consistent endianness with overflow detection for LSN assignment
Transaction IDs corrupted after serialization	Integer encoding issues or field alignment problems	Compare serialized transaction IDs against original values across multiple architectures	Implement architecture-independent integer serialization
Record type enum deserialization failures	Invalid enum variant values or version compatibility issues	Check enum variant encoding against deserialization expectations	Add enum validation with forward compatibility for unknown variants
Page data corruption in before/after images	Buffer management issues or incorrect data copying	Compare original page data against serialized before-images	Fix data copying logic with proper buffer lifecycle management
Checkpoint records missing critical fields	Incomplete serialization of complex data structures	Verify all checkpoint fields are present after deserialization	Implement comprehensive checkpoint validation with required field checking
Log reader fails on segment boundaries	Record spanning multiple segments or incorrect segment header parsing	Test log reading across segment file boundaries	Fix segment boundary handling with proper record reconstruction

**⚠ Pitfall: Platform-Specific Serialization** Writing serialization code that works on the development platform but fails on different architectures or operating systems. The fix is using architecture-independent encoding formats (like little-endian with explicit byte ordering) and testing on multiple platforms early in development.

## Checkpoint and Log Truncation Problems

Checkpointing bugs can cause recovery performance issues, disk space exhaustion, or data loss when log truncation removes records that are still needed for recovery. These problems often involve complex interactions between the checkpoint state machine, active transactions, and log truncation timing.

Symptom	Likely Root Cause	Diagnostic Approach	Primary Fix Strategy
Recovery scans entire log despite recent checkpoint	Checkpoint LSN not properly recorded in master record	Verify master record contents against actual checkpoint records in log	Fix master record updates to correctly reference checkpoint LSN
Log disk space grows without bound	Log truncation not occurring or truncating insufficient data	Monitor log truncation frequency and amount of data removed per truncation	Implement more aggressive truncation with proper safety margin calculation
Data loss after checkpoint and log truncation	Truncation removed records still needed by active transactions	Audit truncation safety calculation against actual transaction requirements	Fix safe truncation LSN calculation to consider all active transaction requirements
Checkpoint creation blocks normal transactions	Synchronous checkpoint implementation instead of fuzzy checkpoint	Measure transaction latency during checkpoint operations	Implement true fuzzy checkpointing with concurrent transaction support
Master record corruption after checkpoint	Atomic update failure or incorrect checksum calculation	Validate master record integrity before and after checkpoint operations	Implement atomic master record updates with proper validation
Checkpoint state machine stuck in transition	Concurrent checkpoint requests or state transition bugs	Monitor checkpoint state machine transitions and timing	Fix state machine logic with proper concurrency control
Recovery fails with "checkpoint not found"	Master record points to truncated checkpoint or corruption	Trace master record LSN against available log segments	Implement checkpoint LSN validation before truncation
Dirty page table inconsistent after checkpoint	Race condition between page modifications and checkpoint scanning	Compare checkpoint dirty page table against actual page modification times	Fix checkpoint scanning to properly handle concurrent page modifications

**⚠ Pitfall: Unsafe Log Truncation** Truncating log segments before ensuring all referenced data is no longer needed for recovery. The fix is implementing conservative truncation policies that maintain safety margins and validate truncation safety before removing any log data.

## Diagnostic Techniques

Effective WAL debugging requires specialized tools and techniques that can extract meaningful information from binary log files, trace complex state transitions during recovery, and validate the intricate invariants that ensure crash safety. These diagnostic approaches go beyond traditional application debugging to address the unique challenges of persistent data structures, crash timing dependencies, and distributed state consistency.

The key insight for WAL diagnostics is that the log file itself is the primary source of truth for understanding system behavior. Unlike traditional applications where runtime state can be inspected directly, WAL systems must be diagnosed primarily through their persistent artifacts: log files, checkpoint records, and master records. This requires building diagnostic tools that can parse, validate, and interpret these binary formats to reconstruct the sequence of events that led to problematic behavior.

### Log File Inspection and Analysis

The most fundamental diagnostic technique for WAL systems is comprehensive log file analysis that can parse, validate, and interpret the binary log records to understand exactly what operations occurred and in what order. This capability is essential because WAL bugs often involve subtle ordering issues or data corruption that can only be detected through careful examination of the actual log contents.

Building effective log inspection tools requires understanding both the binary serialization format and the higher-level semantic constraints that valid log sequences must satisfy. These tools should validate not just individual record integrity but also cross-record invariants like LSN monotonicity, transaction lifecycle consistency, and referential integrity between related records.

Diagnostic Tool	Purpose	Key Capabilities	Usage Pattern
<b>Log File Parser</b>	Parse binary log files into human-readable format	CRC validation, record type identification, LSN sequence validation, transaction grouping	<code>wal-debug parse logfile.wal   grep "txn_id=1234"</code>
<b>LSN Sequence Validator</b>	Verify LSN monotonicity and detect gaps or duplicates	Gap detection, backwards jump identification, sequence integrity checking	<code>wal-debug validate-lsn logfile.wal</code> reports sequence violations
<b>Transaction Lifecycle Tracer</b>	Track complete transaction lifecycles from begin to commit/abort	Transaction state transitions, orphaned transactions, commit/abort matching	<code>wal-debug trace-txn 1234 logfile.wal</code> shows complete transaction history
<b>Checkpoint Consistency Checker</b>	Validate checkpoint records against actual log contents	Active transaction verification, dirty page validation, LSN consistency checking	<code>wal-debug check-checkpoint checkpoint-lsn.wal</code>
<b>Before/After Image Validator</b>	Verify data integrity in redo and undo records	Data corruption detection, image consistency validation, page-level integrity	<code>wal-debug validate-images logfile.wal</code> reports corrupted data
<b>Recovery Simulation Tool</b>	Simulate recovery process without modifying actual database	Dry-run recovery, state reconstruction, consistency validation	<code>wal-debug simulate-recovery --from-checkpoint checkpoint.wal</code>
<b>Log Statistics Analyzer</b>	Generate statistics about log contents and patterns	Record type distribution, transaction size analysis, throughput patterns	<code>wal-debug stats logfile.wal</code> generates comprehensive report
<b>Corruption Pattern Detector</b>	Identify systematic corruption patterns	Bit flip detection, sector boundary issues, systematic corruption patterns	<code>wal-debug detect-corruption logfile.wal</code>

The log file parser serves as the foundation for all other diagnostic tools by converting binary log records into structured data that can be analyzed programmatically. This tool must handle all record types defined in the system ( `RedoRecord` , `UndoRecord` , `CommitRecord` , `AbortRecord` , `CheckpointRecord` ) and validate their integrity using CRC checksums and structural constraints.

**Critical Insight:** Build diagnostic tools incrementally alongside the WAL implementation. Don't wait until bugs occur to create debugging capabilities. The tools themselves often reveal implementation bugs during development.

## State Reconstruction and Validation

State reconstruction techniques allow developers to understand exactly what the system state should be at any point during recovery and compare that expected state against actual outcomes. This capability is crucial for diagnosing subtle consistency bugs where the recovery process produces plausible but incorrect results.

The key technique is maintaining a **golden state oracle** that independently tracks what the database state should be based on committed transactions, then comparing this oracle against the actual recovered database state to detect discrepancies. This approach can catch bugs that traditional testing might miss because the recovered state appears reasonable but doesn't match the expected outcome.

Reconstruction Technique	Purpose	Implementation Approach	Validation Method
Golden State Oracle	Independent tracking of expected database state	Separate implementation that processes same log records	Compare oracle state against recovered database
Transaction Table Reconstruction	Rebuild transaction state from log analysis	Scan log records and maintain transaction lifecycle state	Validate against checkpoint transaction table
Dirty Page Table Validation	Verify dirty page tracking accuracy	Track page modifications independently	Compare against checkpoint dirty page table
LSN Watermark Tracking	Monitor LSN progress through recovery phases	Record LSN checkpoints at each recovery phase	Detect recovery phase regression or stalling
Page-Level Consistency Checking	Verify individual page consistency after recovery	Apply all redo operations to clean pages independently	Compare against actual recovered pages
Transaction Dependency Analysis	Identify transaction ordering constraints	Build dependency graph from read/write conflicts	Validate recovery respects dependency ordering
Undo Chain Validation	Verify backward transaction traversal	Reconstruct undo chains from log records	Validate against actual undo operation sequence
Checkpoint State Verification	Validate checkpoint accurately captures system state	Independently calculate active transactions and dirty pages	Compare against checkpoint record contents

The golden state oracle technique deserves special attention because it provides an independent verification mechanism for recovery correctness. The oracle processes the same log records as the main recovery system but uses a completely separate implementation, making it unlikely that both systems will have the same bugs.

```
// Example golden state oracle structure (in Implementation Guidance)

struct GoldenStateOracle {

    pages: HashMap<PageId, Vec<u8>>,

    committed_transactions: HashSet<TransactionId>,

    active_transactions: HashMap<TransactionId, Vec<Operation>>,

}
```

RUST

**Design Principle:** Always implement validation alongside core functionality. The validation logic often catches bugs that unit tests miss because it checks invariants that are easy to overlook in normal testing scenarios.

## Performance and Resource Monitoring

WAL systems can exhibit performance pathologies that don't cause correctness issues but make the system unusable in practice. These issues often involve excessive fsync calls, memory leaks in buffer management, or disk space exhaustion from insufficient log truncation. Systematic monitoring can detect these issues before they become critical and guide optimization efforts.

The challenge with WAL performance debugging is that many issues only appear under high load or specific timing conditions. Building comprehensive monitoring into the system from the beginning enables detection of performance regressions and provides the data needed to optimize critical paths.

Monitoring Category	Key Metrics	Collection Method	Analysis Technique
Write Path Latency	fsync timing, buffer allocation time, serialization overhead	High-resolution timestamps around critical operations	Percentile analysis, outlier detection
Recovery Performance	Analysis phase duration, redo operation throughput, undo chain traversal time	Phase timing with operation counting	Bottleneck identification, scalability analysis
Memory Usage	Buffer pool utilization, transaction table size, dirty page table growth	Periodic sampling of data structure sizes	Memory leak detection, growth pattern analysis
Disk I/O Patterns	Fsync frequency, write amplification, seek patterns	Filesystem-level I/O tracing	Sequential vs random access analysis
Log File Management	Segment rotation frequency, truncation effectiveness, disk space utilization	File system monitoring with size tracking	Growth rate analysis, space reclamation efficiency
Concurrency Metrics	Lock contention, thread pool utilization, queue depths	Lock timing and thread activity monitoring	Contention hotspot identification
Error Rates	CRC validation failures, partial write detection, recovery restart frequency	Error counting with categorization	Error pattern analysis, failure correlation
Checkpoint Efficiency	Checkpoint duration, transaction blocking time, log truncation amount	Checkpoint lifecycle timing	Checkpoint overhead analysis

Effective monitoring requires carefully placed instrumentation that captures timing and resource usage without significantly impacting the performance being measured. The key is identifying the critical paths and bottlenecks where monitoring overhead is justified by the diagnostic value provided.

**⚠ Pitfall: Performance Monitoring Overhead** Adding too much monitoring can significantly impact WAL performance, especially for high-frequency operations like log record serialization. The fix is using sampling-based monitoring for hot paths and detailed monitoring only for debugging builds or when specific issues are being investigated.

## Logging and Observability

Strategic logging and observability are essential for understanding WAL system behavior both during normal operation and when debugging complex issues. Unlike application-level logging that primarily serves user-facing diagnostics, WAL logging must capture the low-level details needed to understand crash recovery scenarios, timing-sensitive operations, and subtle consistency invariants.

The fundamental challenge is balancing comprehensive observability against performance impact. WAL systems are inherently performance-sensitive because they're in the critical path of every write transaction, so logging strategy must carefully prioritize which information is essential and which can be captured selectively or in debugging builds only.

Think of WAL logging like the instrumentation in a race car - it needs to capture critical performance and safety metrics without adding weight or complexity that affects the primary mission. The logging system itself must be crash-safe and must not introduce additional failure modes that could compromise the very durability guarantees the WAL is designed to provide.

## **Strategic Logging Points**

Identifying the right logging points requires understanding both the normal operation flow and the failure scenarios that are most likely to require debugging. The goal is creating a trace of system behavior that enables reconstruction of the sequence of events leading to any problematic state, while minimizing the performance overhead of logging itself.

The most valuable logging points are those that capture state transitions, invariant validations, and external interactions (like filesystem operations) where the WAL system interfaces with unreliable external components. These logs serve as breadcrumbs that can be followed backwards from a failure point to understand the root cause.

Logging Category	When to Log	Essential Information	Log Level	Performance Impact
Transaction Lifecycle	Transaction begin, commit preparation, commit completion, abort	Transaction ID, LSN assignments, duration, participant count	INFO	Low - infrequent
Log Record Operations	Record creation, serialization, write completion, fsync completion	Record type, LSN, transaction ID, serialization size, timing	DEBUG	Medium - frequent
Recovery Phase Transitions	Analysis start/end, redo start/end, undo start/end	Phase duration, records processed, state table sizes	INFO	Low - recovery only
Checkpoint Lifecycle	Checkpoint initiation, dirty page scanning, checkpoint completion	Checkpoint LSN, active transaction count, dirty page count, duration	INFO	Low - periodic
Error Detection	CRC failures, partial writes, corruption detection	Error type, LSN context, recovery actions, data patterns	WARN	Low - error cases only
Filesystem Interactions	File open/close, fsync calls, disk space checks	File paths, operation duration, error codes, available space	DEBUG	Medium - frequent
Buffer Management	Buffer allocation, pool exhaustion, buffer reuse	Buffer sizes, pool utilization, allocation failures	TRACE	High - very frequent
Invariant Violations	LSN monotonicity, transaction state consistency, page consistency	Violated constraint, expected vs actual values, context information	ERROR	Low - should be rare

The transaction lifecycle logging deserves special attention because it provides the end-to-end view needed to understand how individual transactions flow through the WAL system. This logging should capture not just the major milestones but also the timing relationships that are crucial for understanding performance issues and race conditions.

**Logging Principle:** Log state transitions and external interactions comprehensively, but make data processing and internal computations observable only at debug levels. The goal is understanding what the system did and when, not every detail of how it did it.

## Crash-Safe Logging Infrastructure

A critical but often overlooked requirement is that the WAL system's own logging infrastructure must be crash-safe and must not interfere with the durability guarantees provided by the WAL itself. This means the logging system cannot use the same WAL instance it's instrumenting (which would create circular dependencies) and must handle crashes gracefully without blocking or corrupting the main WAL operations.

The solution is implementing a separate, lightweight logging infrastructure that uses direct file I/O with immediate flushing for critical log messages, while buffering less critical messages to avoid performance impact. This logging system should fail gracefully - if logging fails, the WAL operations should continue successfully rather than being blocked by logging infrastructure problems.

Infrastructure Component	Crash Safety Requirement	Implementation Strategy	Failure Handling
<b>Log Message Buffering</b>	No message loss on crash	Separate log file with immediate flush for ERROR/WARN levels	Buffer loss acceptable for DEBUG/TRACE
<b>File Handle Management</b>	No resource leaks affecting main WAL	Direct file I/O bypassing standard library buffering	Automatic handle recovery on write failure
<b>Message Serialization</b>	No blocking or corruption	Simple text format with minimal parsing requirements	Skip corrupted entries, continue processing
<b>Timestamp Generation</b>	Monotonic timing for event ordering	High-resolution monotonic clock source	Fallback to sequence numbers if clock fails
<b>Thread Safety</b>	No interference with WAL worker threads	Lock-free message queues or thread-local buffers	Per-thread logging to avoid contention
<b>Disk Space Management</b>	No impact on WAL disk space	Separate disk partition or aggressive log rotation	Disable non-critical logging if space critical
<b>Error Reporting</b>	No recursive logging failures	Simple error codes without complex formatting	Silent failure for logging system errors
<b>Configuration Changes</b>	Runtime log level adjustment	Lock-free configuration updates	Conservative defaults if configuration corrupt

The most important aspect is ensuring that logging failures never cause WAL operations to fail or block. The logging system should be designed with the principle that WAL operations are more important than capturing their logs, so any conflict should be resolved in favor of WAL functionality.

**⚠ Pitfall: Logging System Dependency** Creating dependencies between the WAL system and complex logging frameworks can introduce additional failure modes and performance bottlenecks. The fix is implementing a minimal, dedicated logging system specifically designed for the reliability and performance requirements of WAL operations.

## Observability Integration and Monitoring

Beyond basic logging, comprehensive observability requires integration with monitoring systems that can aggregate metrics, detect anomalies, and alert operators to potential issues before they cause system failures. This integration should provide both real-time operational visibility and historical analysis capabilities for performance optimization and capacity planning.

The key insight is that WAL systems have predictable operational patterns during normal operation, so deviations from these patterns can serve as early warning indicators of developing problems. Effective monitoring looks for these pattern deviations rather than just threshold violations on individual metrics.

Observability Integration	Metrics Collected	Alerting Conditions	Analysis Capabilities
Operational Metrics Export	Transaction throughput, commit latency, recovery time	Throughput drops, latency spikes, recovery failures	Performance trending, capacity planning
Health Check Endpoints	System status, recent error counts, resource utilization	Health check failures, error rate increases	Service discovery integration
Distributed Tracing	Transaction flow across components, timing correlation	Trace completion failures, abnormal timing patterns	End-to-end latency analysis
Custom Dashboards	WAL-specific visualizations, recovery state, checkpoint status	Visual anomaly detection, operational state changes	Real-time operational awareness
Log Aggregation	Centralized log collection, structured log parsing	Log volume anomalies, error pattern detection	Historical incident analysis
Alerting Integration	Critical error notifications, performance degradation alerts	System unavailability, data consistency issues	Incident response automation
Capacity Monitoring	Disk space trends, memory usage patterns, I/O utilization	Resource exhaustion prediction	Growth planning and optimization
Performance Profiling	CPU usage breakdowns, I/O wait patterns, lock contention	Performance regression detection	Optimization opportunity identification

The integration should be designed to minimize overhead during normal operation while providing comprehensive visibility when issues occur. This often means using sampling for high-frequency metrics and full capture only for error conditions or when debugging specific issues.

**Observability Principle:** Design observability to answer the questions you'll have during an incident. What was the system doing when the problem occurred? How did it get into that state? What changed recently? These questions drive the choice of what to monitor and log.

## Implementation Guidance

This section provides concrete tools and techniques for implementing robust debugging capabilities in your WAL system. The focus is on creating debugging infrastructure that grows with your implementation and provides systematic approaches to diagnosing the complex failure modes inherent in crash recovery systems.

### Technology Recommendations

Building effective debugging tools requires choosing technologies that balance ease of development against the specialized requirements of binary log file analysis and crash simulation. The recommendations prioritize tools that can handle binary data effectively and provide the precision needed for timing-sensitive WAL operations.

Component	Simple Option	Advanced Option
Log File Parser	Text output with <code>printf</code> debugging	Binary analysis with <code>hex</code> crate and structured output
Crash Simulation	Manual process termination with <code>kill -9</code>	Systematic fault injection with <code>CrashSimulator</code> component
Performance Monitoring	<code>std::time::Instant</code> with manual timing	Integration with <code>metrics</code> crate and Prometheus
Binary Analysis	Manual hex dump inspection	Custom binary parser with validation
Log Visualization	Text-based output with grep/awk	Web-based timeline visualization
State Validation	Manual comparison of expected vs actual	Property-based testing with <code>proptest</code> crate
Memory Debugging	Basic leak detection with allocation tracking	Advanced profiling with <code>valgrind</code> or <code>heaptrack</code>
Concurrency Testing	Manual threading scenarios	Systematic testing with <code>loom</code> crate

### Recommended File Structure

Organizing debugging tools and infrastructure in a clear directory structure helps maintain the code as the project grows and makes it easier for team members to understand and extend the debugging capabilities.

```

wal-project/
├── src/
│   ├── lib.rs           ← Core WAL library
│   ├── log_writer.rs    ← LogWriter component
│   ├── recovery_manager.rs ← RecoveryManager component
│   ├── checkpoint_manager.rs ← CheckpointManager component
│   └── diagnostics/
│       ├── mod.rs        ← Diagnostics module exports
│       ├── log_parser.rs  ← Binary log file parsing
│       ├── crash_simulator.rs ← Crash injection for testing
│       ├── golden_oracle.rs ← Independent state validation
│       ├── corruption_detector.rs ← Systematic corruption detection
│       └── monitoring.rs   ← Performance metrics collection
├── tools/
│   ├── wal-debug/
│   │   ├── main.rs        ← Main debugging CLI tool
│   │   ├── commands/
│   │   │   ├── parse.rs    ← CLI argument parsing and dispatch
│   │   │   ├── validate.rs  ← Individual debug commands
│   │   │   ├── simulate.rs  ← Log file parsing command
│   │   │   └── stats.rs     ← Log validation command
│   │   └── crash-test/
│   │       └── main.rs     ← Recovery simulation command
│   └── tests/
│       ├── integration/
│       │   ├── crash_recovery.rs  ← Log statistics analysis
│       │   └── corruption_handling.rs ← Crash testing harness
│       └── property_tests/
│           └── recovery_properties.rs ← Automated crash scenario testing
└── tests/
    ├── integration/
    │   ├── crash_recovery.rs  ← Integration tests with crash scenarios
    │   └── corruption_handling.rs ← Comprehensive crash testing
    └── property_tests/
        └── recovery_properties.rs ← Corruption detection testing
└── examples/
    ├── debug_session.rs      ← Property-based testing
    └── monitoring_setup.rs   ← WAL invariant validation

```

## Debugging Infrastructure Components

The debugging infrastructure should be implemented as reusable components that can be integrated into both the main WAL library and standalone debugging tools. This approach ensures consistency and allows debugging capabilities to be used both during development and in production troubleshooting.

### Log File Parser Implementation:

```
// Complete binary log parser for diagnostic analysis

use std::fs::File;

use std::io::{BufReader, Read, Seek, SeekFrom};

use crc32fast::Hasher;

pub struct LogFileParser {

    reader: BufReader<File>,

    current_offset: u64,

    file_size: u64,

    validation_mode: ValidationMode,

}

impl LogFileParser {

    pub fn new(file_path: &str, validation_mode: ValidationMode) -> WalResult<Self> {

        let file = File::open(file_path).map_err(WalError::Io)?;

        let metadata = file.metadata().map_err(WalError::Io)?;

        let reader = BufReader::new(file);

        Ok(LogFileParser {

            reader,

            current_offset: 0,

            file_size: metadata.len(),

            validation_mode,

        })
    }

    // Parse next log record from file with comprehensive validation

    pub fn parse_next_record(&mut self) -> WalResult<Option<ParsedRecord>> {


```

```
// TODO 1: Check if we've reached end of file

// TODO 2: Read record header (LSN, type, length fields)

// TODO 3: Validate header fields for consistency

// TODO 4: Read record body based on length from header

// TODO 5: Calculate and verify CRC32 checksum

// TODO 6: Deserialize record body based on type field

// TODO 7: Validate record-specific invariants (LSN ordering, etc.)

// TODO 8: Return ParsedRecord with metadata for analysis

unimplemented!()

}
```

```
// Scan entire log file and build comprehensive analysis

pub fn analyze_log_file(&mut self) -> WalResult<LogAnalysis> {

    // TODO 1: Initialize analysis state (transaction table, statistics)

    // TODO 2: Parse all records in sequential order

    // TODO 3: Track transaction lifecycles and detect orphaned transactions

    // TODO 4: Validate LSN monotonicity and detect gaps

    // TODO 5: Build dirty page table from redo records

    // TODO 6: Verify checkpoint consistency against actual log contents

    // TODO 7: Detect corruption patterns and systematic issues

    // TODO 8: Generate comprehensive analysis report

    unimplemented!()

}
```

```
pub struct ParsedRecord {

    pub offset: u64,
```

```
pub record: LogRecord,  
  
pub raw_data: Vec<u8>,  
  
pub crc_valid: bool,  
  
pub validation_errors: Vec<String>,  
  
}  
  
pub struct LogAnalysis {  
  
pub total_records: usize,  
  
pub transaction_summary: HashMap<TransactionId, TransactionSummary>,  
  
pub lsn_gaps: Vec<(LSN, LSN)>,  
  
pub corruption_count: usize,  
  
pub checkpoint_consistency: CheckpointAnalysis,  
  
pub performance_metrics: LogPerformanceMetrics,  
  
}  
  
#[derive(Debug)]  
  
pub enum ValidationMode {  
  
    Strict,      // Fail on any validation error  
  
    Permissive, // Report errors but continue parsing  
  
    Recovery,   // Skip corrupted records and continue  
  
}
```

### Crash Simulation Component:

```
// Systematic crash injection for testing recovery scenarios

use std::sync::{Arc, Mutex};

use std::collections::HashSet;

use rand::Rng;

pub struct CrashSimulator {

    crash_points: Arc<Mutex<HashSet<String>>>,
    crash_probability: f64,
    current_operation: Arc<Mutex<String>>,
    crash_count: Arc<Mutex<u32>>,
}

impl CrashSimulator {

    pub fn new() -> Self {
        CrashSimulator {
            crash_points: Arc::new(Mutex::new(HashSet::new())),
            crash_probability: 0.0,
            current_operation: Arc::new(Mutex::new(String::new())),
            crash_count: Arc::new(Mutex::new(0)),
        }
    }
}

// Add specific crash injection point for systematic testing

pub fn add_crash_point(&self, operation: &str) {
    // TODO 1: Add operation name to crash points set
    // TODO 2: Log crash point registration for test reproducibility
    unimplemented!()
}
```

```
// Check if crash should be triggered at current operation

pub fn check_crash_trigger(&self, operation: &str) -> bool {

    // TODO 1: Update current operation for tracking

    // TODO 2: Check if operation is in crash points set

    // TODO 3: Apply crash probability if probabilistic mode enabled

    // TODO 4: Log crash trigger decision for debugging

    // TODO 5: Increment crash count if crash triggered

    // TODO 6: Return true if crash should occur

    unimplemented!()

}

// Configure probabilistic crash injection for stress testing

pub fn set_crash_probability(&mut self, probability: f64) {

    // TODO 1: Validate probability is between 0.0 and 1.0

    // TODO 2: Set crash probability for random crash injection

    // TODO 3: Log probability change for test reproducibility

    unimplemented!()

}

// Integration with main WAL components for crash testing

macro_rules! crash_point {
    ($simulator:expr, $operation:expr) => {

        if $simulator.check_crash_trigger($operation) {

            std::process::exit(9); // Simulate hard crash

        }
    }
}
```

```
};
```

```
}
```

### **Golden State Oracle Implementation:**

```
// Independent implementation for validating recovery correctness

use std::collections::{HashMap, HashSet};

pub struct GoldenStateOracle {

    pages: HashMap<PageId, Vec<u8>>,
    committed_transactions: HashSet<TransactionId>,
    active_transactions: HashMap<TransactionId, Vec<PendingOperation>>,
    transaction_commit_order: Vec<(TransactionId, LSN)>,
}

impl GoldenStateOracle {

    pub fn new() -> Self {
        GoldenStateOracle {
            pages: HashMap::new(),
            committed_transactions: HashSet::new(),
            active_transactions: HashMap::new(),
            transaction_commit_order: Vec::new(),
        }
    }

    // Process log record and update oracle state independently

    pub fn process_log_record(&mut self, record: &LogRecord) -> WalResult<()> {
        // TODO 1: Match on record type to determine processing
        // TODO 2: For redo records, track operation in active transaction
        // TODO 3: For commit records, apply all transaction operations atomically
        // TODO 4: For abort records, discard all transaction operations
        // TODO 5: Update pages only for committed transactions
        // TODO 6: Maintain transaction commit ordering for validation
    }
}
```

```
unimplemented!()

}

// Compare oracle state against recovered database for validation

pub fn verify_recovery_correctness(&self, recovered_db: &MockDatabase) -> WalResult<()> {
    unimplemented!()

    // TODO 1: Compare page contents between oracle and recovered database

    // TODO 2: Verify all committed transactions are reflected in recovered state

    // TODO 3: Ensure no active (uncommitted) transaction changes are visible

    // TODO 4: Validate page-level consistency and integrity

    // TODO 5: Check transaction ordering is preserved in final state

    // TODO 6: Report detailed discrepancies if validation fails

    unimplemented!()

}

// Generate comprehensive validation report

pub fn generate_validation_report(&self, recovered_db: &MockDatabase) -> ValidationReport {
    unimplemented!()

    // TODO 1: Compare all pages and identify discrepancies

    // TODO 2: Analyze transaction commit patterns

    // TODO 3: Check for missing or extra data in recovered state

    // TODO 4: Generate detailed report with specific failures

    unimplemented!()

}

pub struct ValidationReport {
    pub pages_correct: usize,
```

```
pub pages_incorrect: usize,  
  
pub page_discrepancies: Vec<PageDiscrepancy>,  
  
pub transaction_issues: Vec<TransactionValidationError>,  
  
pub overall_success: bool,  
  
}
```

## Milestone Debugging Checkpoints

Each milestone should include specific debugging validation points that verify not just functional correctness but also the robustness of error handling and edge case management.

### Milestone 1 (Log Record Format) Debugging Checkpoint:

After implementing log record serialization and deserialization:

```
# Validate serialization round-trip correctness  
  
cargo test record_serialization_roundtrip  
  
# Test with intentionally corrupted data  
  
../tools/wal-debug validate-corruption test-logs/corrupted.wal  
  
# Expected behavior: Clean detection of corruption with specific error messages  
  
# Signs of problems: Panics on invalid data, silent corruption acceptance
```

BASH

### Milestone 2 (Log Writer) Debugging Checkpoint:

After implementing atomic log append and durability guarantees:

```
# Test crash safety with systematic crash injection  
  
cargo test crash_recovery_basic --features crash_testing  
  
# Validate fsync timing and durability  
  
../tools/crash-test --scenario partial_write --iterations 100  
  
# Expected behavior: All committed transactions preserved after crash  
  
# Signs of problems: Lost transactions, corrupted log files, partial writes
```

BASH

### Milestone 3 (Crash Recovery) Debugging Checkpoint:

After implementing ARIES recovery phases:

```
# Comprehensive recovery validation with golden oracle
cargo test recovery_correctness_validation

# Test with complex crash scenarios
./tools/crash-test --scenario complex_workload --crash_points all

# Expected behavior: Perfect state reconstruction matching oracle
# Signs of problems: Missing transactions, incorrect page contents, recovery failures
```

BASH

### Milestone 4 (Checkpointing) Debugging Checkpoint:

After implementing fuzzy checkpointing and log truncation:

```
# Validate checkpoint consistency and truncation safety
cargo test checkpoint_truncation_safety

# Long-running stability test with periodic checkpointing
./tools/wal-debug stability-test --duration 1h --checkpoint_interval 30s

# Expected behavior: Stable operation with bounded disk usage
# Signs of problems: Unbounded log growth, checkpoint inconsistency, truncation errors
```

BASH

## Future Extensions

**Milestone(s):** All milestones - future extensions build upon the foundation established in Milestone 1 (Log Record Format), Milestone 2 (Log Writer), Milestone 3 (Crash Recovery), and Milestone 4 (Checkpointing) to support enhanced performance, advanced features, and scalability improvements

The Write-Ahead Logging implementation provides a solid foundation for durability and crash recovery, but real-world database systems require additional capabilities to handle production workloads effectively. Think of the current WAL implementation as a reliable foundation that can be extended with performance optimizations,

advanced features, and scalability improvements - like building a house where you start with a strong foundation and then add rooms, better insulation, and modern amenities as needs evolve.

This section explores three categories of extensions that transform the basic WAL into a production-ready system: performance optimizations that reduce overhead and improve throughput, advanced features that enable new use cases and operational capabilities, and scalability improvements that handle larger workloads and distributed environments. Each extension builds upon the existing architecture while maintaining the fundamental durability guarantees established in the core implementation.

## Performance Optimizations

Performance optimization in WAL systems centers around reducing I/O overhead, improving concurrency, and accelerating recovery times. The current implementation provides correctness but leaves significant performance improvements on the table. Think of performance optimization as tuning a race car - the basic engine works, but careful tuning of fuel injection, aerodynamics, and weight distribution can dramatically improve lap times without compromising reliability.

### Group Commit Optimization

**Group commit** represents one of the most impactful optimizations for write-heavy workloads by amortizing fsync costs across multiple transactions. The current implementation calls `force_sync()` for each transaction commit, resulting in one fsync per transaction. Group commit batches multiple pending commits into a single fsync operation, dramatically reducing the I/O overhead per transaction.

#### Decision: Implement Group Commit Batching

- **Context:** Individual fsync calls for each transaction create significant I/O bottlenecks, limiting throughput to the inverse of fsync latency
- **Options Considered:** Per-transaction fsync (current), fixed-size batching, timeout-based batching, adaptive batching
- **Decision:** Implement timeout-based batching with configurable batch size limits
- **Rationale:** Timeout-based batching provides predictable commit latency while maximizing throughput during high-load periods
- **Consequences:** Improves throughput by 10-100x under write-heavy workloads at the cost of slightly higher commit latency

The group commit mechanism extends the `LogWriter` component with a **commit coordinator** that collects pending commit requests and flushes them in batches. When a transaction requests commit, instead of immediately calling fsync, the coordinator adds the transaction to a pending batch and either waits for more transactions to arrive or for a timeout to expire before flushing the entire batch.

Component	Purpose	Key Responsibilities
CommitCoordinator	Batch management	Collect pending commits, manage timeouts, coordinate group fsync
CommitBatch	Batch state	Track pending transactions, manage completion notifications
CommitWaiter	Synchronization	Block transaction threads until their batch completes
BatchMetrics	Observability	Track batch sizes, wait times, throughput improvements

The group commit algorithm follows these steps:

1. Transaction calls `commit_transaction()` which creates a `CommitWaiter` and adds it to the current batch
2. If the batch is not full and timeout hasn't expired, the transaction thread waits on the `CommitWaiter`
3. When the batch fills up or timeout expires, the coordinator writes all pending commit records
4. The coordinator calls `force_sync()` once for the entire batch
5. All `CommitWaiter` instances in the batch are notified, unblocking their transaction threads
6. A new empty batch is created for subsequent commits

This optimization requires careful handling of commit ordering and failure scenarios. If the group fsync fails, all transactions in the batch must receive the same error. If the system crashes during group commit, recovery must handle the case where some transactions in a batch may have been durably written while others were not.

## Log Compression

**Log compression** reduces storage overhead and improves I/O bandwidth by compressing log records before writing them to disk. WAL files often contain repetitive data patterns - similar page IDs, repeated field names in structured records, and redundant metadata - making them excellent candidates for compression.

### Decision: Implement Per-Segment Compression with LZ4

- **Context:** Log files consume significant disk space, and I/O bandwidth limits write throughput
- **Options Considered:** No compression, per-record compression, per-segment compression, streaming compression
- **Decision:** Implement per-segment compression using LZ4 algorithm
- **Rationale:** LZ4 provides good compression ratios with minimal CPU overhead; segment-level compression maximizes compression efficiency while maintaining segment-based recovery
- **Consequences:** Reduces storage costs by 40-60% and improves effective I/O bandwidth at the cost of CPU overhead and implementation complexity

The compression system integrates with the existing `LogSegment` structure by adding a compression layer between the in-memory record buffer and the disk file. Compression occurs when segments are sealed (transitioned from active to read-only), ensuring that the current active segment remains uncompressed for efficient appends.

Component	Purpose	Key Responsibilities
<code>CompressionEngine</code>	Algorithm abstraction	Provide pluggable compression algorithms (LZ4, Zstd, Snappy)
<code>CompressedSegment</code>	Compressed storage	Manage compressed segment files with metadata headers
<code>CompressionMetadata</code>	Segment information	Track original size, compression ratio, block boundaries
<code>CompressionBuffer</code>	Memory management	Manage compression scratch buffers and temporary storage

The compression workflow operates as follows:

1. When a `LogSegment` reaches the size threshold, it's marked for compression
2. The compression engine reads the entire segment into memory and applies the chosen algorithm
3. A compressed segment file is created with a metadata header containing original size and block boundaries
4. The original uncompressed segment is replaced with the compressed version
5. During recovery, compressed segments are decompressed on-demand as log records are read

Decompression during recovery must handle partial segments and corruption scenarios. The compression metadata includes checksums and block boundaries that allow recovery to skip corrupted compressed blocks and continue processing valid data.

## Parallel Recovery

**Parallel recovery** accelerates the crash recovery process by processing multiple log records concurrently during the redo phase. The current ARIES implementation processes log records sequentially, but many redo operations can execute in parallel if they target different pages or don't conflict with each other.

## Decision: Implement Page-Level Parallel Redo

- **Context:** Sequential redo processing creates recovery bottlenecks for large logs, extending database downtime
- **Options Considered:** Sequential redo (current), transaction-level parallelism, page-level parallelism, operation-level parallelism
- **Decision:** Implement page-level parallel redo with dependency tracking
- **Rationale:** Page-level parallelism provides significant speedup while maintaining simple conflict detection (operations on different pages are independent)
- **Consequences:** Reduces recovery time by 3-10x depending on workload characteristics, but increases implementation complexity and memory usage

The parallel recovery system extends the `RecoveryManager` with a **parallel redo coordinator** that identifies independent operations and executes them across multiple worker threads. The coordinator maintains dependency graphs to ensure that operations affecting the same page execute in LSN order while allowing operations on different pages to proceed concurrently.

Component	Purpose	Key Responsibilities
<code>RedoCoordinator</code>	Parallel orchestration	Schedule redo operations across worker threads with dependency management
<code>RedoWorker</code>	Operation execution	Execute redo operations on assigned pages in LSN order
<code>DependencyTracker</code>	Conflict detection	Track page-level dependencies and enforce ordering constraints
<code>RedoScheduler</code>	Load balancing	Distribute redo operations evenly across available worker threads

The parallel redo algorithm works as follows:

1. The analysis phase builds a dependency graph showing which operations affect each page
2. Operations are grouped by target page and sorted by LSN within each group
3. The scheduler assigns page groups to worker threads, ensuring no two workers handle the same page
4. Each worker processes its assigned pages sequentially (maintaining LSN order within each page)
5. Workers coordinate through barriers to ensure later LSN operations don't execute before earlier ones complete
6. The undo phase remains sequential since it must process transactions in reverse chronological order

This approach requires careful handling of cross-page dependencies and ensures that page-level locks prevent race conditions between parallel redo workers.

## Advanced Features

Advanced features extend the WAL system beyond basic durability to support sophisticated operational requirements. These features enable capabilities like backup and restore, data replication, and compliance with regulatory requirements for data retention and auditability.

### Point-in-Time Recovery

**Point-in-time recovery (PITR)** enables restoring the database to any specific timestamp by replaying log records up to that point in time. This capability is essential for recovering from logical errors (like accidental data deletion) where the database is consistent but contains incorrect data.

#### Decision: Implement LSN-Based Point-in-Time Recovery

- **Context:** Users need to recover from logical errors by restoring to a specific point in time before the error occurred
- **Options Considered:** No PITR support, timestamp-based PITR, LSN-based PITR, hybrid approach
- **Decision:** Implement LSN-based PITR with timestamp mapping
- **Rationale:** LSN-based recovery provides precise control and leverages existing recovery infrastructure; timestamp mapping provides user-friendly interface
- **Consequences:** Enables precise recovery from logical errors but requires additional metadata tracking and log retention policies

The PITR system extends the recovery infrastructure with **recovery targets** that specify the desired endpoint for recovery operations. Instead of recovering to the end of the log, the system stops at a specified LSN or timestamp and leaves the database in a consistent state at that point.

Component	Purpose	Key Responsibilities
RecoveryTarget	Target specification	Define recovery endpoint by LSN, timestamp, or transaction ID
TimestampMapper	Time mapping	Maintain mapping between timestamps and LSNs for time-based recovery
TargetValidator	Consistency checking	Ensure recovery target results in consistent database state
PITRManager	Recovery orchestration	Coordinate point-in-time recovery with standard ARIES recovery

The PITR algorithm modifies the standard ARIES recovery process:

1. The analysis phase proceeds normally but stops when reaching the target LSN
2. The transaction table is examined to identify transactions that were active at the target point
3. The redo phase replays all committed changes up to the target LSN

4. The undo phase rolls back any transactions that were active at the target point
5. The system creates a new checkpoint at the target point to establish the new recovery baseline

PITR requires careful handling of transaction boundaries to ensure the recovered state is consistent. The system must ensure that partially committed transactions are either fully committed (if their commit record appears before the target) or fully rolled back (if their commit record appears after the target).

## Logical Replication

**Logical replication** captures database changes at a logical level (insert/update/delete operations) rather than physical level (page modifications) to enable features like cross-database replication, schema evolution, and selective data synchronization.

### Decision: Implement Change Data Capture with Logical Log Records

- **Context:** Applications need to replicate data changes to other systems, data warehouses, and analytics platforms
- **Options Considered:** Physical replication (page-level), logical replication (row-level), trigger-based capture, application-level capture
- **Decision:** Implement logical replication using enhanced log records with row-level change information
- **Rationale:** Logical replication provides flexibility for cross-platform replication and selective synchronization while leveraging existing WAL infrastructure
- **Consequences:** Enables sophisticated replication scenarios but increases log overhead and complexity

The logical replication system enhances the existing `LogRecord` types with **logical change records** that capture row-level modifications in addition to page-level changes. These records include enough information to reconstruct the logical operation (table name, primary key, changed columns) without requiring access to the original database pages.

Component	Purpose	Key Responsibilities
<code>LogicalLogRecord</code>	Change capture	Capture table-level changes with row identifiers and column values
<code>ChangeStreamReader</code>	Stream processing	Read logical changes from WAL and emit replication events
<code>ReplicationFilter</code>	Change filtering	Apply table and column filters for selective replication
<code>ChangePublisher</code>	Event publishing	Publish change events to downstream systems via messaging or APIs

The logical replication workflow operates alongside the existing WAL system:

1. Write operations generate both physical `RedoRecord` entries and logical `LogicalLogRecord` entries
2. The logical records include table schema information, primary key values, and before/after row images
3. Replication consumers read the logical change stream and apply changes to target systems
4. Change filtering allows consumers to subscribe to specific tables or column sets
5. Schema evolution is handled by including schema metadata in logical records

This approach requires coordination between the WAL system and the database's schema management to ensure logical records contain sufficient information for replication consumers to interpret changes correctly.

## Distributed WAL

**Distributed WAL** extends the logging system across multiple nodes to support distributed transactions and provide high availability through log replication. This enables building distributed database systems that maintain ACID properties across network boundaries.

### Decision: Implement Multi-Master WAL with Consensus-Based Ordering

- **Context:** Distributed applications require transaction coordination across multiple database nodes while maintaining consistency
- **Options Considered:** Single-master replication, multi-master with conflict resolution, consensus-based distributed WAL, blockchain-based logging
- **Decision:** Implement consensus-based distributed WAL using Raft for log ordering
- **Rationale:** Consensus protocols provide strong consistency guarantees required for distributed transactions while maintaining availability during network partitions
- **Consequences:** Enables true distributed transactions but introduces network latency and complexity in failure scenarios

The distributed WAL system extends the single-node architecture with **distributed consensus** for globally ordering log records across multiple nodes. Each node maintains a local WAL segment, but global transaction ordering is established through a consensus protocol that ensures all nodes agree on the sequence of log records.

Component	Purpose	Key Responsibilities
DistributedLogWriter	Multi-node coordination	Coordinate log writes across multiple nodes with consensus
ConsensusManager	Ordering protocol	Implement Raft consensus for global log record ordering
GlobalLSNManager	LSN coordination	Maintain globally consistent LSN sequence across all nodes
DistributedRecovery	Multi-node recovery	Coordinate recovery across multiple nodes with log reconstruction

The distributed WAL algorithm works as follows:

1. Transactions can span multiple nodes, with each node maintaining local operation records
2. At commit time, all participating nodes propose their local records to the consensus group
3. The consensus protocol establishes a global ordering of all proposed records
4. Once consensus is reached, all nodes append the globally ordered records to their local WALs
5. Commit acknowledgment is sent only after the consensus group confirms durable storage
6. Recovery reconstructs the global state by merging local WAL segments according to the consensus ordering

This approach handles network partitions by maintaining consistency within the majority partition while blocking commits in minority partitions, ensuring that distributed transactions maintain ACID properties even during network failures.

## Scalability Improvements

Scalability improvements address the limitations that emerge when WAL systems handle high-throughput workloads or large-scale deployments. These enhancements focus on reducing contention, distributing load, and parallelizing operations that become bottlenecks in the current single-threaded design.

### Multiple Log Files

**Multiple log files** eliminate the single-writer bottleneck by partitioning log records across multiple independent log streams. This approach increases write throughput by allowing concurrent writes to different log files while maintaining recovery correctness through cross-log coordination.

## Decision: Implement Hash-Based Log Partitioning

- **Context:** Single log file creates write contention bottleneck under high concurrent load
- **Options Considered:** Single log (current), transaction-based partitioning, page-based partitioning, hash-based partitioning
- **Decision:** Implement hash-based partitioning using transaction ID hash for log file selection
- **Rationale:** Hash-based partitioning provides even load distribution while keeping transaction records together for efficient recovery
- **Consequences:** Increases write throughput linearly with partition count but complicates recovery with cross-log coordination requirements

The multiple log files system extends the `LogWriter` architecture with a **log partition manager** that routes log records to different files based on partitioning criteria. Each partition operates as an independent log stream with its own LSN sequence and segment rotation policy.

Component	Purpose	Key Responsibilities
<code>LogPartitionManager</code>	Partition coordination	Route log records to appropriate partitions and coordinate cross-partition operations
<code>PartitionedLogWriter</code>	Multi-stream writing	Manage multiple independent log writers with separate LSN sequences
<code>CrossLogRecovery</code>	Multi-log recovery	Coordinate recovery across multiple log partitions with global ordering
<code>PartitionLoadBalancer</code>	Load distribution	Monitor partition utilization and rebalance hot partitions

The partitioned logging algorithm operates as follows:

1. Incoming log records are routed to partitions based on transaction ID hash
2. Each partition maintains an independent LSN sequence prefixed with partition ID
3. Transactions that span multiple partitions write records to multiple logs
4. Commit records include references to all partitions touched by the transaction
5. Recovery scans all partitions and merges records using timestamp ordering for cross-partition consistency
6. Checkpoints coordinate across all partitions to establish global recovery points

This approach requires careful handling of cross-partition transactions and global ordering during recovery to ensure consistency when transactions modify data that affects multiple partitions.

## Partitioned Recovery

**Partitioned recovery** parallelizes the crash recovery process by dividing the database into independent recovery domains that can be processed concurrently. This approach dramatically reduces recovery time for

large databases by allowing multiple recovery processes to work simultaneously on different data partitions.

### Decision: Implement Table-Based Recovery Partitioning

- **Context:** Recovery time increases linearly with database size, creating unacceptable downtime for large databases
- **Options Considered:** Sequential recovery (current), page-range partitioning, table-based partitioning, transaction-based partitioning
- **Decision:** Implement table-based partitioning with cross-table dependency tracking
- **Rationale:** Table-based partitioning aligns with application data organization and provides natural isolation boundaries for most workloads
- **Consequences:** Reduces recovery time significantly but requires sophisticated dependency tracking for cross-table transactions

The partitioned recovery system extends the `RecoveryManager` with **recovery domains** that operate independently during the redo phase while maintaining global consistency during analysis and undo phases. Each domain handles a subset of tables and their associated indexes.

Component	Purpose	Key Responsibilities
<code>RecoveryDomain</code>	Partition management	Handle recovery for assigned tables with independent redo processing
<code>DependencyGraph</code>	Cross-domain coordination	Track transactions that span multiple recovery domains
<code>GlobalTransactionTable</code>	Cross-partition state	Maintain transaction state across all recovery domains
<code>RecoveryCoordinator</code>	Domain orchestration	Coordinate recovery phases across multiple independent domains

The partitioned recovery algorithm modifies ARIES recovery:

1. The analysis phase scans the entire log but partitions discovered transactions and dirty pages by table
2. Recovery domains are created for each table group, with cross-domain transactions tracked separately
3. The redo phase executes in parallel across domains, with coordination for cross-domain transactions
4. The undo phase processes transactions sequentially but delegates page modifications to appropriate domains
5. Each domain creates independent checkpoints that are aggregated into a global checkpoint

This approach requires careful synchronization to ensure that cross-table foreign key constraints and triggers are handled correctly during parallel recovery operations.

## Asynchronous Checkpointing

**Asynchronous checkpointing** eliminates the performance impact of checkpoint operations by moving checkpoint I/O to background threads and streaming checkpoint data continuously rather than creating periodic snapshots. This approach maintains the recovery benefits of checkpointing while eliminating checkpoint-related latency spikes.

### Decision: Implement Streaming Checkpoint with Background Writers

- **Context:** Current fuzzy checkpointing still causes periodic I/O spikes and transaction latency during checkpoint writes
- **Options Considered:** Synchronous checkpointing (current), asynchronous checkpointing, streaming checkpointing, incremental checkpointing
- **Decision:** Implement streaming checkpoint with background I/O threads
- **Rationale:** Streaming checkpoints eliminate periodic I/O spikes while providing continuous recovery point advancement
- **Consequences:** Provides consistent performance with no checkpoint-related latency spikes but increases system complexity and resource usage

The asynchronous checkpointing system replaces periodic checkpoint operations with **continuous checkpoint streaming** that incrementally writes checkpoint data in the background while transactions continue normally. The system maintains multiple checkpoint generations to handle concurrent checkpoint operations.

Component	Purpose	Key Responsibilities
StreamingCheckpoint	Continuous checkpointing	Stream checkpoint data continuously without blocking transaction processing
CheckpointStreamer	Background I/O	Handle background writes of checkpoint data with flow control
CheckpointRegistry	Generation management	Track multiple concurrent checkpoint generations and coordinate completion
IncrementalTracker	Change tracking	Monitor incremental changes since last checkpoint for efficient streaming

The streaming checkpoint algorithm operates continuously:

1. Background threads continuously scan dirty pages and write checkpoint data
2. Checkpoint data is written incrementally without requiring atomic snapshots
3. Multiple checkpoint generations exist simultaneously during streaming operations
4. Completed checkpoints are atomically promoted to become the new recovery baseline

5. Old checkpoint generations are garbage collected after newer ones are confirmed
6. Recovery uses the most recent complete checkpoint as the starting point

This approach requires sophisticated coordination between checkpoint streams and concurrent transactions to ensure checkpoint consistency while maintaining transaction performance.

**⚠ Pitfall: Checkpoint Consistency Without Snapshots** A common mistake in streaming checkpoint implementations is failing to maintain temporal consistency when checkpoint data is written incrementally over time. Without atomic snapshots, the checkpoint might capture an inconsistent view where some changes appear in the checkpoint while their dependent changes don't, violating referential integrity. The solution is to use logical timestamps (LSNs) to ensure checkpoint consistency by only including changes up to a specific LSN boundary, even when the physical checkpoint write spans multiple time periods.

## Implementation Guidance

The future extensions require careful architectural planning to integrate with the existing WAL implementation without breaking established interfaces or compromising correctness guarantees. Each extension should be implemented as an optional enhancement that can be enabled through configuration flags.

## Technology Recommendations

Extension Category	Simple Option	Advanced Option
Group Commit	Fixed-size batching with timer	Adaptive batching with load prediction
Compression	LZ4 with fixed block size	Multi-algorithm with adaptive selection
Parallel Recovery	Fixed thread pool	Work-stealing with dynamic load balancing
Logical Replication	JSON change events	Avro/Protobuf with schema registry
Distributed WAL	Simple majority voting	Full Raft with dynamic membership
Multiple Log Files	Hash-based static partitioning	Consistent hashing with rebalancing

## Recommended File Structure

```
src/                                RUST

├── wal/
│   ├── core/                         ← existing core WAL implementation
│   |   ├── log_writer.rs
│   |   ├── recovery_manager.rs
│   |   └── checkpoint_manager.rs
│   ├── extensions/                  ← new extension modules
│   |   ├── performance/
│   |   |   ├── group_commit.rs
│   |   |   ├── compression.rs
│   |   |   └── parallel_recovery.rs
│   |   ├── advanced/
│   |   |   ├── pitr.rs
│   |   |   ├── logical_replication.rs
│   |   |   └── distributed_wal.rs
│   |   └── scalability/
│   |       ├── partitioned_writer.rs
│   |       ├── partitioned_recovery.rs
│   |       └── streaming_checkpoint.rs
│   └── config/
│       ├── extension_config.rs   ← configuration for enabling extensions
│       └── performance_config.rs
```

## Infrastructure Starter Code

RUST

```
// Configuration management for WAL extensions

use std::time::Duration;

use serde::{Deserialize, Serialize};

#[derive(Debug, Clone, Serialize, Deserialize)]

pub struct WalExtensionConfig {

    pub group_commit: GroupCommitConfig,

    pub compression: CompressionConfig,

    pub parallel_recovery: ParallelRecoveryConfig,

    pub partitioning: PartitioningConfig,

}

#[derive(Debug, Clone, Serialize, Deserialize)]

pub struct GroupCommitConfig {

    pub enabled: bool,

    pub max_batch_size: usize,

    pub batch_timeout: Duration,

    pub adaptive_batching: bool,

}

#[derive(Debug, Clone, Serialize, Deserialize)]

pub struct CompressionConfig {

    pub enabled: bool,

    pub algorithm: CompressionAlgorithm,

    pub compression_level: u32,

    pub min_segment_size: u64,

}
```

```
#[derive(Debug, Clone, Serialize, Deserialize)]  
  
pub enum CompressionAlgorithm {  
  
    None,  
  
    Lz4,  
  
    Zstd,  
  
    Snappy,  
  
}  
  
impl Default for WalExtensionConfig {  
  
    fn default() -> Self {  
  
        Self {  
  
            group_commit: GroupCommitConfig {  
  
                enabled: false,  
  
                max_batch_size: 100,  
  
                batch_timeout: Duration::from_millis(10),  
  
                adaptive_batching: false,  
  
            },  
  
            compression: CompressionConfig {  
  
                enabled: false,  
  
                algorithm: CompressionAlgorithm::Lz4,  
  
                compression_level: 1,  
  
                min_segment_size: 1024 * 1024, // 1MB  
  
            },  
  
            parallel_recovery: ParallelRecoveryConfig {  
  
                enabled: false,  
  
                worker_threads: num_cpus::get(),  
  
                batch_size: 1000,  
            },  
        }  
    }  
}
```

```
        },

        partitioning: PartitioningConfig {

            enabled: false,

            partition_count: 4,

            partition_strategy: PartitionStrategy::Hash,

        },
    },
}

}

// Extension registry for managing optional components

pub struct ExtensionRegistry {

    group_commit: Option<Box<dyn GroupCommitManager>>,

    compression: Option<Box<dyn CompressionEngine>>,

    logical_replication: Option<Box<dyn ChangeStreamPublisher>>,

}

impl ExtensionRegistry {

    pub fn new(config: &WalExtensionConfig) -> WalResult<Self> {

        let mut registry = Self {

            group_commit: None,

            compression: None,

            logical_replication: None,

        };

        if config.group_commit.enabled {

            registry.group_commit = Some(Box::new(
                DefaultGroupCommitManager::new(&config.group_commit)?
            ));

        }
    }
}
```

```
        ));

    }

    if config.compression.enabled {
        registry.compression = Some(Box::new(
            create_compression_engine(&config.compression)?
        ));
    }

    Ok(registry)
}

pub fn get_group_commit(&self) -> Option<&dyn GroupCommitManager> {
    self.group_commit.as_ref().map(|g| g.as_ref())
}

}
```

## Core Logic Skeletons

```
// Group commit manager skeleton                                         RUST

pub trait GroupCommitManager: Send + Sync {

    /// Submit a transaction for group commit batching

    /// Returns when the transaction has been durably committed

    fn commit_transaction(&self, txn_id: TransactionId, commit_record: CommitRecord) ->
    WalResult<LSN>;

}

pub struct DefaultGroupCommitManager {

    // TODO: Add fields for batch management, timing, coordination
}

impl DefaultGroupCommitManager {

    pub fn new(config: &GroupCommitConfig) -> WalResult<Self> {

        // TODO 1: Initialize batch tracking structures (pending commits, batch counter)

        // TODO 2: Create timing mechanisms for batch timeout handling

        // TODO 3: Set up coordination primitives for thread synchronization

        // TODO 4: Start background batch processing thread

        todo!()
    }

}

impl GroupCommitManager for DefaultGroupCommitManager {

    fn commit_transaction(&self, txn_id: TransactionId, commit_record: CommitRecord) ->
    WalResult<LSN> {

        // TODO 1: Create commit waiter for this transaction

        // TODO 2: Add transaction to current batch (thread-safe)

        // TODO 3: Check if batch is full or should be flushed immediately
    }
}
```

```
// TODO 4: If batch ready, trigger immediate flush; otherwise wait for batch
completion

// TODO 5: Handle timeout scenarios and ensure transaction doesn't wait
indefinitely

// TODO 6: Return LSN once batch has been successfully written and synced

// Hint: Use condition variables or channels for cross-thread coordination

todo!()

}

}

// Compression engine skeleton

pub trait CompressionEngine: Send + Sync {

    /// Compress a log segment when it becomes read-only

    fn compress_segment(&self, segment: &LogSegment) -> WalResult<CompressedSegment>;


    /// Decompress segment data during recovery

    fn decompress_segment(&self, compressed: &CompressedSegment) -> WalResult<Vec<u8>>;


}

pub struct Lz4CompressionEngine {

    // TODO: Add compression parameters, buffer management
}

impl CompressionEngine for Lz4CompressionEngine {

    fn compress_segment(&self, segment: &LogSegment) -> WalResult<CompressedSegment> {

        // TODO 1: Read entire segment into memory buffer

        // TODO 2: Apply LZ4 compression with configured compression level

        // TODO 3: Create compression metadata (original size, block boundaries, checksums)

        // TODO 4: Write compressed data with metadata header to new file
    }
}
```

```

    // TODO 5: Atomically replace original segment with compressed version

    // TODO 6: Return CompressedSegment handle for future decompression

    // Hint: Handle compression failures gracefully - keep original if compression
    doesn't help

    todo!()

}

fn decompress_segment(&self, compressed: &CompressedSegment) -> WalResult<Vec<u8>> {

    // TODO 1: Read compression metadata to determine original size and structure

    // TODO 2: Allocate output buffer with original size

    // TODO 3: Decompress data using LZ4 algorithm

    // TODO 4: Validate decompressed data integrity using stored checksums

    // TODO 5: Handle partial decompression for corrupted segments

    // TODO 6: Return decompressed data ready for log record parsing

    // Hint: Implement streaming decompression for large segments to avoid memory
    pressure

    todo!()

}

}

// Parallel recovery coordinator skeleton

pub struct ParallelRecoveryCoordinator {

    // TODO: Add worker thread pool, dependency tracking, coordination primitives

}

impl ParallelRecoveryCoordinator {

    pub fn new(config: &ParallelRecoveryConfig, storage: Arc<dyn Storage>) ->
    WalResult<Self> {

        // TODO 1: Create worker thread pool based on configuration

```

```

    // TODO 2: Initialize dependency tracking structures for page-level conflicts

    // TODO 3: Set up work queues and coordination channels between workers

    // TODO 4: Create barrier synchronization for cross-worker coordination

    todo!()

}

pub fn parallel_redo(&mut self, redo_records: Vec<RedoRecord>) -> WalResult<()> {

    // TODO 1: Analyze redo records to build page-level dependency graph

    // TODO 2: Group records by target page to enable parallel processing

    // TODO 3: Distribute page groups across worker threads for load balancing

    // TODO 4: Execute redo operations in parallel while maintaining LSN ordering
    // within each page

    // TODO 5: Use barriers to coordinate completion of LSN boundaries across workers

    // TODO 6: Handle errors from individual workers and coordinate error recovery

    // Hint: Sort records by LSN within each page group before distributing to workers

    todo!()

}

}

```

## Milestone Checkpoints

### Group Commit Extension Checkpoint:

- Run performance benchmark: `cargo test --release test_group_commit_performance`
- Expected: 5-10x throughput improvement under concurrent write load
- Verify commit latency remains bounded: maximum batch timeout should never exceed configured value
- Test crash recovery: ensure batched commits are properly recovered or rolled back

### Compression Extension Checkpoint:

- Compress sample log files: `cargo run --bin compress_logs --input logs/ --output compressed/`
- Expected: 40-60% size reduction with LZ4, <10ms compression time per 64MB segment

- Verify decompression during recovery: compressed segments should decompress identically to originals
- Test partial corruption: recovery should handle corrupted compression blocks gracefully

### Parallel Recovery Extension Checkpoint:

- Create large log file with diverse page updates: `cargo run --bin generate_test_log --pages 10000 --records 1000000`
- Run parallel recovery: `cargo test test_parallel_recovery_performance`
- Expected: 3-5x recovery speedup on multi-core systems, identical final database state
- Monitor worker utilization: all workers should be actively processing throughout recovery

### Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
Group commit timeout	Batch never fills, timer thread blocked	Check batch fill rates and timer thread status	Adjust batch size or timeout, verify timer thread isn't deadlocked
Compression corruption	Algorithm mismatch or buffer overflow	Compare checksums and validate compression parameters	Use consistent algorithm versions, add buffer bounds checking
Parallel recovery deadlock	Cross-page dependencies not properly tracked	Analyze dependency graph and worker thread states	Improve dependency analysis, add deadlock detection and recovery
PITR inconsistency	Transaction boundary not properly handled	Check transaction state at recovery target LSN	Ensure target LSN falls at transaction boundary, not mid-transaction
Replication lag	Change stream backpressure or network issues	Monitor change stream buffer sizes and network latency	Add flow control, implement change stream buffering and retry logic

## Glossary

**Milestone(s):** All milestones - this glossary provides essential vocabulary and terminology definitions that support Milestone 1 (Log Record Format), Milestone 2 (Log Writer), Milestone 3 (Crash Recovery), and Milestone 4 (Checkpointing)

## Core WAL Concepts

Think of this glossary as your technical dictionary for Write-Ahead Logging. Each term represents a fundamental building block in the WAL architecture, much like how a construction glossary defines the difference between a foundation, beam, and rafter. Understanding these precise definitions is crucial because WAL systems use highly specific terminology that can be confusing when terms sound similar but have distinct technical meanings.

**Write-Ahead Logging (WAL):** A technique where all database changes are recorded in a sequential log file before being applied to the actual database pages. The "write-ahead" principle ensures that log records describing a change must reach persistent storage before the change itself is written to the database. This ordering guarantee enables crash recovery by providing a complete history of intended changes.

**ARIES:** Algorithm for Recovery and Isolation Exploiting Semantics. This is the industry-standard recovery algorithm that defines the three-phase recovery process: Analysis (scan log to build recovery state), Redo (replay all committed changes), and Undo (rollback incomplete transactions). ARIES is notable for its use of physiological logging and support for fuzzy checkpoints.

**Log Sequence Number (LSN):** A monotonically increasing 64-bit identifier assigned to each log record. LSNs serve multiple purposes: they provide a total ordering of all logged operations, act as timestamps for recovery decisions, and enable efficient log scanning. Each page in the database also stores the LSN of the most recent change applied to that page.

**durability:** One of the ACID properties, guaranteeing that once a transaction commits, its changes will survive system crashes, power failures, and other catastrophic events. In WAL systems, durability is achieved by forcing log records to persistent storage via `fsync` before acknowledging transaction commits.

**fsync:** A system call that forces all buffered data for a file to be written to persistent storage. This is critical for durability guarantees because operating systems typically buffer writes in memory for performance. Without `fsync`, a crash could lose recently written log records even though the application believes they were saved.

## Log Record Types and Structure

**log record:** The fundamental unit of information written to the WAL. Each log record contains a header with metadata (LSN, transaction ID, record type, length) followed by operation-specific data and a CRC checksum for corruption detection. Log records are written sequentially and never modified after creation.

**redo operation:** During crash recovery, the process of replaying a committed transaction's changes to restore them to the database. Redo operations use after-images stored in log records to reconstruct the final state of modified pages. Redo operations must be idempotent since recovery might apply the same record multiple times.

**undo operation:** During crash recovery, the process of removing uncommitted changes from the database. Undo operations use before-images stored in log records to restore pages to their state before the transaction began. Undo operations generate Compensation Log Records (CLRs) to ensure the rollback itself is logged.

**after-image:** The new content of a database page location after a transaction's modification. After-images are stored in redo log records and used during recovery to restore committed changes. The after-image represents exactly what should be written to the specified page offset.

**before-image:** The original content of a database page location before a transaction's modification. Before-images are stored in undo log records and used during recovery to reverse uncommitted changes. Before-images enable transactions to be rolled back to their starting state.

**Compensation Log Record (CLR):** A special log record written during undo operations to describe the reversal of a previous change. CLRs are crucial for ensuring that undo operations themselves are logged, preventing infinite undo chains if recovery is interrupted. CLRs point to the next record that should be undone, effectively shortening the undo chain.

## Recovery and Transaction Management

**transaction table:** A data structure built during the Analysis phase of recovery that tracks the state of all transactions found in the log. For each transaction, the table maintains the transaction status (active, committed, aborted), the LSN of the transaction's most recent log record, and the LSN of the next record to undo if rollback is needed.

**dirty page table:** A data structure built during the Analysis phase that tracks which database pages have been modified but may not have been written to disk. Each entry contains a page ID and the LSN of the oldest log record that modified that page. This table determines which pages need redo processing during recovery.

**idempotency:** The property that applying the same operation multiple times produces the same result as applying it once. This is crucial for redo operations during recovery because a crash during recovery itself could require re-applying the same log records. WAL implementations must ensure that redo operations can be safely repeated.

**undo chain:** A linked sequence of log records for a single transaction, connected by LSN pointers that enable backward traversal. During undo processing, the recovery system follows this chain to find all operations that must be reversed. The chain is traversed from the transaction's final log record back to its beginning.

## Checkpointing and Log Management

**fuzzy checkpoint:** A checkpoint created without stopping concurrent transactions or waiting for all dirty pages to be written to disk. Fuzzy checkpoints record a snapshot of the transaction table and dirty page table at a specific LSN, enabling recovery to start from that point rather than scanning the entire log. The "fuzzy" nature means the checkpoint reflects a range of LSNs rather than a single atomic point.

**checkpoint LSN:** The Log Sequence Number that serves as the temporal boundary for a checkpoint. This LSN identifies the logical point in time when the checkpoint was initiated. Recovery uses the checkpoint LSN to determine which log records must be processed and which can be ignored because their effects are already captured in the checkpoint.

**log truncation:** The process of safely removing old log segments that are no longer needed for recovery. Truncation can only occur after a checkpoint has been completed and all transactions that were active before the checkpoint have either committed or aborted. Proper truncation is essential for preventing unbounded log growth.

**master record:** A small, well-known file that serves as the bootstrap entry point for recovery. The master record contains the file location and LSN of the most recent valid checkpoint. During recovery startup, the system reads the master record to determine where to begin log scanning. Updates to the master record use atomic rename operations for crash safety.

## File and Storage Management

**append-only semantics:** The principle that log records are always written to the end of the log file and never modified in place. This design simplifies concurrency control (only one writer position), enables efficient sequential I/O, and provides natural ordering guarantees. Append-only semantics also simplify crash recovery because partial writes can only occur at the end of the file.

**log rotation:** The process of creating a new log segment file when the current segment reaches a configured size limit. Rotation prevents individual log files from becoming unwieldy and enables parallel processing during recovery. Each segment maintains metadata about its LSN range to support efficient seeking during recovery.

**force-write:** An operation that writes data to a file and immediately calls `fsync` to ensure the data reaches persistent storage before returning. Force-writes are used for critical log records like transaction commits where durability must be guaranteed before acknowledging success to the client.

**buffer management:** The strategy for batching multiple log records in memory before writing them to disk as a group. Effective buffer management reduces I/O overhead by amortizing the cost of system calls and `fsync` operations across multiple records. Buffers must be flushed before they become full to maintain bounded commit latency.

## Error Handling and Recovery

**partial write:** A scenario where only part of a write operation completed before a crash, leaving incomplete data on disk. In log files, partial writes can only occur at the end because of append-only semantics. Recovery systems must detect partial writes and truncate them to restore the log to a consistent state.

**torn page:** A situation where part of a database page write completed but other parts did not, resulting in a page that contains a mixture of old and new data. While primarily a database-level concern, WAL systems help detect torn pages by storing page LSNs that can be compared against log records.

**corruption detection:** Techniques for identifying invalid or damaged log records. WAL implementations use CRC checksums, magic numbers, and structural validation to detect corruption. When corruption is detected, recovery systems must decide whether to stop processing, skip the corrupted record, or attempt repair.

**group commit:** An optimization where multiple transactions share the cost of a single `fsync` operation by batching their commit records together. Group commit significantly improves throughput for workloads with many small transactions while maintaining durability guarantees. The optimization is transparent to applications.

## Testing and Validation

**crash consistency:** The guarantee that system state remains valid and recoverable after unexpected crashes. Crash-consistent systems can always be restored to a coherent state, even if some recent work is lost. WAL systems achieve crash consistency through careful ordering of log writes and `fsync` operations.

**property-based testing:** A testing approach that verifies system invariants using randomly generated inputs rather than fixed test cases. For WAL systems, property-based testing might verify that recovered state always matches a golden oracle implementation regardless of crash timing or workload patterns.

**golden state oracle:** An independent, simplified implementation of system logic used to verify the correctness of the main implementation. The oracle maintains the "correct" system state and is compared against the main system after recovery to detect bugs. Oracles prioritize correctness over performance.

## Performance and Optimization

**log compression:** Techniques for reducing the storage overhead of log records without compromising recovery correctness. Compression can be applied to individual records or entire log segments. Compressed logs must support random access for recovery processing, which may require index structures.

**parallel recovery:** Accelerating crash recovery by processing multiple log records concurrently. Parallel recovery must carefully manage dependencies between operations to ensure the final state matches sequential processing. This typically involves analyzing operation conflicts and creating dependency graphs.

**streaming checkpoint:** A checkpointing approach that continuously writes checkpoint data in the background rather than creating periodic snapshot points. Streaming checkpoints reduce the latency spike associated with traditional checkpoints and provide more frequent recovery waypoints.

## Advanced Features

**point-in-time recovery:** The ability to restore a database to any specific timestamp in the past, not just the most recent consistent state. Point-in-time recovery requires mapping between wall-clock time and LSNs, and careful handling of transaction boundaries to avoid partial transactions.

**logical replication:** Capturing database changes at a logical level (INSERT, UPDATE, DELETE operations) rather than physical page modifications. Logical replication enables cross-database replication, data transformation during replication, and selective replication of table subsets.

**distributed WAL:** Extending Write-Ahead Logging across multiple nodes in a distributed system. Distributed WAL requires consensus protocols to establish global ordering, techniques for handling node failures, and coordination of recovery across multiple machines.

## Data Structure Types

The following table defines the key data structures used throughout the WAL implementation:

Type	Purpose	Key Fields
LogRecord	Base type for all log entries	Discriminated union of record types
RedoRecord	Stores after-image for committed changes	lsn, txn_id, page_id, offset, after_image
UndoRecord	Stores before-image for rollback	lsn, txn_id, page_id, offset, before_image
CommitRecord	Marks transaction completion	lsn, txn_id
AbortRecord	Marks transaction rollback	lsn, txn_id
CheckpointRecord	Recovery waypoint information	lsn, active_transactions, dirty_pages
LSN	Log sequence number type	64-bit monotonic identifier
TransactionId	Transaction identifier type	64-bit unique transaction ID
PageId	Database page identifier	32-bit page number
LogSegment	Individual log file management	file, path, size tracking, LSN range
DatabasePage	Page data with LSN tracking	page_id, data, lsn
TransactionState	Transaction metadata during recovery	status, last_lsn, undo_next_lsn

## Error Types and Handling

The following table defines error types and their handling strategies:

Error Type	Cause	Detection Method	Recovery Action
WalError::Io	File system failures	System call return codes	Retry with backoff or fail
WalError::Corruption	Invalid checksums or magic numbers	CRC validation, structure checks	Skip record or abort recovery
WalError::InvalidRecord	Malformed log record structure	Schema validation	Log error and continue
WalError::CheckpointInProgress	Concurrent checkpoint operations	State machine check	Wait or retry operation
WalError::TruncationFailed	Unable to remove old log segments	File deletion failure	Continue with warning
WalError::MasterRecordCorrupted	Bootstrap file damage	Checksum validation	Use backup or rebuild

## State Management Enums

The following table defines enumeration types used for state tracking:

Enum Type	Values	Purpose
TransactionStatus	Active, Committed, Aborted	Track transaction state during recovery
CheckpointState	Idle, Collecting, Writing, Complete	Manage fuzzy checkpoint state machine
DiskSpaceStatus	Normal, Warning, Critical, Emergency	Monitor available disk space
ValidationMode	Strict, Permissive, Recovery	Control log validation strictness
CompressionAlgorithm	None, Lz4, Zstd, Snappy	Select compression method
PartitionStrategy	Hash, Range, RoundRobin	Choose log partitioning approach

## Constants and Configuration

The following table defines important constants and configuration values:

Constant	Value	Purpose
DEFAULT_SEGMENT_SIZE	64MB	Maximum size for log segments before rotation
PAGE_SIZE	4KB	Standard database page size
RECORD_HEADER_SIZE	21 bytes	Fixed size of log record headers
CRC_SIZE	4 bytes	Size of CRC32 checksum field
MASTER_RECORD_MAGIC	0xCHEC_KPNT	Magic number for master record validation
MASTER_RECORD_VERSION	1	Current master record format version
SMALL_RECORD_SIZE	256 bytes	Buffer size for small log records
MEDIUM_RECORD_SIZE	1KB	Buffer size for typical log records
LARGE_RECORD_SIZE	4KB	Buffer size for large log records

## Implementation Guidance

This glossary serves as your reference while implementing the WAL system. When you encounter unfamiliar terminology in the codebase or documentation, refer back to these definitions to maintain conceptual clarity.

### A. Technology Recommendations Table:

Component	Simple Option	Advanced Option
Serialization	JSON with serde	Custom binary format with bit packing
Checksums	CRC32 with crc32fast crate	BLAKE3 for cryptographic integrity
File I/O	std::fs with manual fsync	tokio async I/O with custom buffer management
Concurrency	std::sync Mutex/RwLock	crossbeam lock-free data structures
Testing	Standard unit tests	proptest property-based testing

### B. Recommended File Structure:

```
wal-implementation/
src/
  lib.rs           ← Public API and re-exports
  types/
    mod.rs         ← Type definitions and constants
    records.rs     ← LogRecord and variant definitions
    errors.rs      ← WalError and result types
    identifiers.rs ← LSN, TransactionId, PageId types
  storage/
    mod.rs         ← Storage trait and implementations
    memory.rs      ← MemoryStorage for testing
    disk.rs        ← Disk-based storage
  log_writer/
    mod.rs         ← LogWriter implementation
    segment.rs     ← LogSegment management
    buffer.rs      ← BufferPool implementation
  recovery/
    mod.rs         ← RecoveryManager implementation
    transaction_table.rs ← TransactionTable implementation
    dirty_page_table.rs ← DirtyPageTable implementation
  checkpoint/
    mod.rs         ← CheckpointManager implementation
    state_machine.rs ← CheckpointStateMachine
    master_record.rs ← MasterRecord handling
  testing/
    mod.rs         ← Test utilities and fixtures
    crash_simulator.rs ← CrashSimulator for testing
    golden_oracle.rs ← GoldenStateOracle implementation
tests/
  integration/    ← End-to-end integration tests
  property/       ← Property-based test suites
  crash_safety/   ← Crash injection test scenarios
```

## C. Core Type Definitions:

```
// Essential type aliases that appear throughout the codebase

pub type LSN = u64;

pub type TransactionId = u64;

pub type PageId = u32;

pub type WalResult<T> = Result<T, WalError>;


// Constants referenced in the glossary

pub const DEFAULT_SEGMENT_SIZE: u64 = 64 * 1024 * 1024; // 64MB

pub const PAGE_SIZE: usize = 4096; // 4KB

pub const RECORD_HEADER_SIZE: usize = 21; // bytes

pub const CRC_SIZE: usize = 4; // bytes

pub const MASTER_RECORD_MAGIC: u32 = 0xCHEC_5350; // "CHEC" + "SP" (space)

pub const MASTER_RECORD_VERSION: u32 = 1;
```

## D. Glossary Usage Patterns:

When implementing components, use this glossary to:

1. **Verify terminology consistency:** Ensure your variable names and comments use the preferred terms defined here
2. **Understand type relationships:** Reference the data structure tables when designing component interfaces
3. **Handle errors appropriately:** Use the error type definitions to implement proper error handling
4. **Maintain conceptual clarity:** Return to concept definitions when implementation details become confusing

## E. Milestone Checkpoints:

After implementing each milestone, verify your understanding by:

1. **Milestone 1:** Ensure all log record types are implemented with correct field names and types from the glossary
2. **Milestone 2:** Verify that fsync, append-only semantics, and buffer management work as defined
3. **Milestone 3:** Test that transaction table, dirty page table, and undo chains function according to their definitions
4. **Milestone 4:** Confirm that fuzzy checkpoints and log truncation behavior match the glossary descriptions

## F. Common Terminology Mistakes:

### ⚠ Pitfall: Confusing LSN with transaction ID

- **Problem:** Using transaction IDs where LSNs are needed for temporal ordering
- **Fix:** Remember that LSNs provide global ordering across all transactions, while transaction IDs only identify specific transactions

### ⚠ Pitfall: Misunderstanding "fuzzy" in fuzzy checkpoints

- **Problem:** Thinking fuzzy means imprecise or unreliable
- **Fix:** Fuzzy means the checkpoint captures state across a range of LSNs rather than a single atomic point

### ⚠ Pitfall: Using "log" generically

- **Problem:** Referring to application logs, error logs, and transaction logs interchangeably
- **Fix:** Always specify "Write-Ahead Log" or "WAL" when referring to the transaction log specifically