ADR-004-002-async-batching-strategy

ADR-004-002: Async Batching Strategy for Trajectory Logging

Status: Accepted **Date**: 2025-11-16 **Deciders**: Research Team **Related**: SPEC-PPP-004 (Trajectory Logging & MCP Integration)

Context

Trajectory logging must record every user-agent turn during spec-kit execution. Requirements: - **Latency**: <1ms overhead per turn (agent execution is ~10-30 seconds, logging should be <0.01% overhead) - **Throughput**: Support 20-50 turns per agent execution - **Non-blocking**: Must not block tokio async runtime - **Data integrity**: No data loss on crashes

Four approaches were evaluated: 1. **Synchronous single write**: rusqlite, blocks on each turn 2. **Synchronous batched write**: rusqlite with transactions, blocks briefly 3. **Asynchronous single write**: tokio-rusqlite, non-blocking but overhead 4. **Asynchronous batched write**: tokio-rusqlite with buffering

---

Decision

We will use **Asynchronous batched write** (Option 4) with the following parameters: - **Buffer size**: 5-10 turns - **Flush interval**: 500ms - **Library**: tokio-rusqlite (23,000 downloads/month)

Rationale

Performance Comparison

Strategy	Latency/Turn	Throughput	Blocks Runtime	Implementatio
Sync Single	1ms	1,000/sec	❌ Yes	Simple
Sync Batched	0.067ms	15,000/sec	❌ Yes	Medium
Async Single	0.5ms	2,000/sec	✅ No	Medium

Async Batched	**0.1ms**	**10,000/sec**	✅ **No**	Complex

Source: SPEC-PPP-004 findings.md, benchmark estimates from "15k inserts/s with Rust and SQLite"

Winner: Async batched achieves **0.1ms/turn** (10x better than sync single) without blocking runtime.

Async Batched Implementation

```
use tokio::sync::mpsc;
use tokio_rusqlite::Connection;
use std::time::Duration;

pub struct TrajectoryLogger {
    tx: mpsc::Sender<LogEntry>,
}

impl TrajectoryLogger {
    pub fn new(db_path: String) -> Self {
        let (tx, rx) = mpsc::channel(100);  // Bounded channel for
backpressure

        tokio::spawn(async move {
            writer_task(db_path, rx).await
        });

        Self { tx }
    }

    pub async fn log_turn(&self, turn: Turn) -> Result<()> {
        self.tx.send(LogEntry::Turn(turn)).await?;
        Ok(())
    }
}

async fn writer_task(
    db_path: String,
    mut rx: mpsc::Receiver<LogEntry>,
) -> Result<()> {
    let conn = Connection::open(&db_path).await?;
    let mut buffer = Vec::new();

    loop {
        tokio::select! {
            // Receive entries from channel
            Some(entry) = rx.recv() => {
                buffer.push(entry);

                // Flush if buffer full
                if buffer.len() >= 10 {
                    flush_buffer(&conn, &buffer).await?;
                    buffer.clear();
                }
            }

            // Flush on timeout (every 500ms)
```

```
                        _ = tokio::time::sleep(Duration::from_millis(500)) => {
                            if !buffer.is_empty() {
                                flush_buffer(&conn, &buffer).await?;
                                buffer.clear();
                            }
                        }
                    }
                }
            }

            async fn flush_buffer(conn: &Connection, buffer: &[LogEntry]) ->
Result<()> {
                conn.call(move |conn| {
                    let tx = conn.transaction()?;
                    for entry in buffer {
                        // Insert each entry
                        tx.execute("INSERT INTO trajectory_turns (...) VALUES
(...)", ...)?;
                    }
                    tx.commit()
                }).await
            }
```

Key Features: 1. **Bounded channel** (100 capacity) for backpressure - prevents memory overflow 2. **Dual flush triggers**: Buffer full (10 entries) OR timeout (500ms) 3. **Single writer task**: No lock contention (SQLite single-writer limitation) 4. **Transaction per batch**: Atomic writes, no partial data

---

Buffer Size Selection

Buffer Size	Latency	Flush Frequency	Data Loss Risk
1 (no buffer)	0.5ms	Every turn	Minimal
5	0.2ms	Every 5 turns	Low
10	**0.1ms**	**Every 10 turns**	**Low**
50	0.02ms	Every 50 turns	Medium
100	0.01ms	Every 100 turns	High

Decision: Buffer size = **10**

Rationale: - Typical agent execution: 20-50 turns → 2-5 flushes per run - Data loss window: Max 10 turns (acceptable, <1 second of work) - Performance: 10x better than no buffering (0.1ms vs 1ms) - Memory: ~5KB per buffer (10 turns × 500 bytes/turn)

Flush Interval Selection

Interval	Worst-Case Delay	Data Loss Window	Performance
100ms	100ms	100ms work	High overhead
250ms	250ms	250ms work	Medium overhead
500ms	**500ms**	**<1sec work**	**Optimal**
1000ms	1s	1s work	Low overhead

5000ms	5s	5s work	Too high

Decision: Flush interval = **500ms**

Rationale: - Agent turns are ~5-10 seconds apart (user typing time) - 500ms guarantees data persisted before next turn 99% of the time - Data loss window: <1 second of work (acceptable for logging) - Performance: Reduces write frequency by 5-10x vs 100ms

---

Why Not Synchronous?

Problem: Synchronous writes block tokio runtime

```
// BAD: Blocks entire async runtime
pub fn log_turn_sync(conn: &Connection, turn: Turn) -> Result<()> {
    conn.execute(
        "INSERT INTO trajectory_turns (...) VALUES (...)",
        params![...],
    )?;  // ← Blocks ALL async tasks for ~1ms
    Ok(())
}
```

Impact: - Blocks consensus synthesis (~1s per agent) - Blocks UI updates (TUI frozen during logging) - Cascading delays across all async tasks

Evidence: Rust async best practice - never block in async fn

Source: tokio documentation - "Blocking operations in async code"

Why Not Async Single Write?

Comparison: Async batched vs async single

Metric	Async Single	Async Batched
Latency	0.5ms	**0.1ms (5x better)**
Lock contention	High (every turn)	**Low (every 10 turns)**
Transaction overhead	High	**Low (amortized)**
Complexity	Medium	Medium

Decision: 5x performance improvement justifies slightly more complex code.

Consequences

Positive

1. ✅ **10x performance**: 0.1ms/turn vs 1ms/turn (sync single)
2. ✅ **Non-blocking**: Tokio runtime stays responsive
3. ✅ **High throughput**: 10,000 turns/sec capacity
4. ✅ **Atomic batches**: Transaction guarantees no partial data
5. ✅ **Backpressure**: Bounded channel prevents memory overflow

Negative

1. ⚠ **Data loss window**: Up to 10 turns (500ms) if crash occurs
   - Mitigation: Acceptable for logging use case (not critical transactions)
   - Alternative: Set buffer size=1 for critical data (trade performance)
2. ⚠ **Complexity**: More complex than synchronous writes
   - Mitigation: Well-tested pattern (used in production systems)
   - Code: ~50 lines vs ~10 lines (synchronous)
3. ⚠ **Debugging**: Async stack traces harder to debug
   - Mitigation: Comprehensive logging in writer_task
   - Error handling: Explicit Result propagation

Neutral

1. 📊 **Memory overhead**: ~5KB per buffer (10 turns)
   - Negligible: <0.01% of typical memory usage

Trade-offs Considered

Flush on Agent Completion

Alternative: Don't use timeout, flush only when agent completes

```
// No timeout, manual flush
impl TrajectoryLogger {
    pub async fn flush(&self) -> Result<()> {
        // Signal writer to flush immediately
    }
}

// Usage
logger.log_turn(turn1).await?;
logger.log_turn(turn2).await?;
// ... agent completes
logger.flush().await?;  // ← Manual flush
```

Rejected because: - Requires caller to remember to flush (error-prone) - If agent crashes, data is lost entirely (no timeout fallback) - More complex API (2 methods instead of 1)

Accepted approach: Automatic timeout ensures data persists even if flush() not called.

WAL Mode

Alternative: Enable Write-Ahead Logging (WAL) for even better performance

```
conn.execute("PRAGMA journal_mode=WAL", [])?;
conn.execute("PRAGMA synchronous=NORMAL", [])?;
```

Performance: - Latency: 0.02ms/turn (5x better than batched) - Throughput: 50,000/sec

Deferred to Phase 3 because: - Current performance (0.1ms) already meets requirements (<1ms target) - WAL adds complexity (separate WAL files to manage) - Marginal benefit (0.1ms → 0.02ms not critical)

Decision: Implement WAL in Phase 3 if benchmarks show bottleneck.

Configuration

Add to `config.toml`:

```
[ppp.trajectory]
buffer_size = 10              # Turns to buffer before flush
flush_interval_ms = 500       # Flush interval in milliseconds
channel_capacity = 100        # Backpressure limit
```

Tuning guidance: - **Low latency priority**: Set `buffer_size=1, flush_interval_ms=100` - **High throughput priority**: Set `buffer_size=50, flush_interval_ms=1000` - **Balanced (default)**: Use documented values

---

Testing Strategy

Unit Tests

```
#[tokio::test]
async fn test_async_batching() {
    let logger = TrajectoryLogger::new(":memory:".to_string());

    // Log 5 turns (below buffer size)
    for i in 0..5 {
        logger.log_turn(Turn { turn_number: i, ...
}).await.unwrap();
    }

    // Wait for timeout flush (500ms)
    tokio::time::sleep(Duration::from_millis(600)).await;

    // Verify all 5 turns persisted
    let count = query_turn_count();
    assert_eq!(count, 5);
}

#[tokio::test]
async fn test_buffer_full_flush() {
    let logger = TrajectoryLogger::new(":memory:".to_string());

    // Log 10 turns (exactly buffer size)
    for i in 0..10 {
        logger.log_turn(Turn { turn_number: i, ...
}).await.unwrap();
    }

    // Should flush immediately (no timeout wait needed)
    tokio::time::sleep(Duration::from_millis(10)).await;
```

```
        // Verify all 10 turns persisted
        let count = query_turn_count();
        assert_eq!(count, 10);
    }
```

Benchmark Tests

```
    #[tokio::test]
    async fn bench_async_batched() {
        let logger = TrajectoryLogger::new("bench.db".to_string());

        let start = Instant::now();
        for i in 0..1000 {
            logger.log_turn(Turn { turn_number: i, ...
}).await.unwrap();
        }
        let elapsed = start.elapsed();

        // Target: <0.1ms per turn
        assert!(elapsed < Duration::from_millis(100));
    }
```

References

1. Kerkour, S. (2024). "15k inserts/s with Rust and SQLite" - https://kerkour.com/high-performance-rust-with-sqlite
2. tokio-rusqlite documentation - https://lib.rs/crates/tokio-rusqlite
3. tokio async best practices - https://tokio.rs/tokio/tutorial/spawning
4. SPEC-PPP-004 findings.md - Performance benchmarks

Decision Drivers

Driver	Weight	Async Batched	Sync Single	Async Single
Performance	40%	✅ 0.1ms	❌ 1ms	⚠ 0.5ms
Non-blocking	30%	✅ Yes	❌ No	✅ Yes
Data Integrity	20%	⚠ 500ms window	✅ Immediate	⚠ Immediate
Complexity	10%	⚠ Medium	✅ Simple	⚠ Medium

Total Score: Async Batched **85%**, Async Single **70%**, Sync Single **40%**

Winner: Async Batched by significant margin.