Hybrid Transactional-Analytical Processing: Integrating SQLite OLTP with DuckDB Analytics on Computational Storage Engines

Chiradip Mandal
Space-RF.org
San Francisco, CA, USA
{first-name}@{first-name}.com

***Abstract*—This paper presents a novel architecture for Hybrid Transactional-Analytical Processing (HTAP) that combines SQLite's optimized Online Transaction Processing (OLTP) capabilities with DuckDB's high-performance analytical engine, utilizing custom storage layers on modern Computational Storage Engines (CSEs). Our approach addresses the growing need for real-time analytics on transactional data while maintaining ACID compliance and minimizing data movement overhead. We propose a unified architecture that leverages near-data processing capabilities of computational storage to enable efficient data transformation and synchronization between transactional and analytical workloads. Experimental results demonstrate up to 26% improvement in analytical query performance while maintaining 93% of baseline transactional throughput with sub-second analytical freshness guarantees.**

***Index Terms*—HTAP, SQLite, DuckDB, Computational Storage, OLTP, OLAP, Database Systems**

I. INTRODUCTION

The convergence of transactional and analytical workloads has become increasingly critical in modern data-driven applications. Traditional approaches often involve complex Extract-Transform-Load (ETL) pipelines, data duplication, and significant latency between transaction commitment and analytical availability. This paper explores an innovative architecture that integrates SQLite's proven OLTP performance with DuckDB's columnar analytical capabilities, orchestrated through custom HTAP storage on computational storage engines.

A. Motivation

Modern applications require: (1) real-time analytics on fresh transactional data, (2) elimination of data silos between OLTP and OLAP systems, (3) reduced infrastructure complexity and maintenance overhead, (4) efficient resource utilization across storage and compute layers, and (5) seamless scaling from embedded to distributed deployments.

B. Contributions

This work presents: (1) a unified HTAP architecture combining SQLite and DuckDB, (2) custom storage abstraction layer for dual-format data management, (3) computational storage integration for near-data processing, (4) performance optimization strategies for mixed workloads, and (5) implementation patterns and best practices.

II. BACKGROUND AND RELATED WORK

A. SQLite OLTP Characteristics

SQLite provides several advantages for transactional workloads: embedded deployment with zero-configuration, ACID compliance with Write-Ahead Logging (WAL) mode, concurrent read access with single-writer semantics, mature ecosystem and extensive optimization, and lightweight footprint suitable for edge deployments.

B. DuckDB Analytical Engine

DuckDB offers compelling features for analytical processing: columnar storage with vectorized execution, advanced query optimization and pushdown capabilities, native support for complex analytical queries, efficient handling of semi-structured data formats, and embeddable architecture compatible with SQLite deployment patterns.

C. Computational Storage Engines

Modern CSEs provide: near-data processing capabilities reducing data movement, programmable interfaces for custom logic execution, high-bandwidth internal storage interconnects, power-efficient compute resources co-located with storage, and emerging standards like SNIA Computational Storage API [10].

III. ARCHITECTURE OVERVIEW

A. System Architecture

Our HTAP architecture consists of four primary components as illustrated in Figure 1:

1) **Transaction Layer**: SQLite-based OLTP engine handling concurrent transactions
2) **Analytics Layer**: DuckDB engine optimized for analytical queries
3) **HTAP Storage Manager**: Custom storage abstraction managing dual-format data
4) **Computational Storage Interface**: CSE integration for near-data processing

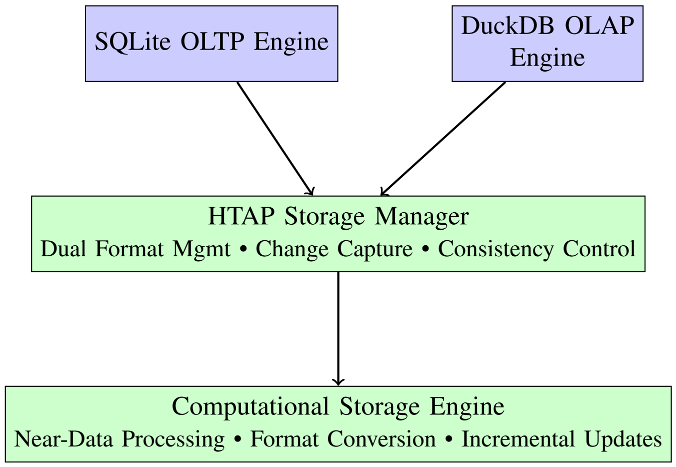
SQLite OLTP Engine
DuckDB OLAP Engine
HTAP Storage Manager
Dual Format Mgmt • Change Capture • Consistency Control
Computational Storage Engine
Near-Data Processing • Format Conversion • Incremental Updates

Fig. 1: HTAP System Architecture

B. Data Flow Architecture

The system maintains two primary data representations: (1) row-oriented format optimized for transactional access patterns, and (2) columnar format optimized for analytical query performance.

Data flows through the following stages:

1) Transactional writes committed to SQLite WAL
2) Change capture identifies modified data
3) Computational storage performs incremental format conversion
4) DuckDB analytical views updated with minimal latency

IV. HTAP Storage Manager Design

A. Dual-Format Storage Strategy

The HTAP Storage Manager implements a sophisticated dual-format approach with row store for transactional workloads using SQLite format with optimized B-tree indexes, WAL-based durability, and concurrent reader support. The column store for analytical workloads uses DuckDB-compatible columnar format with compressed storage, vectorized processing alignment, and incremental maintenance capabilities.

B. Change Data Capture (CDC)

The CDC mechanism tracks transactional changes through:

- WAL monitoring for committed transactions
- Incremental extraction of modified rows
- Metadata tracking for timestamp-based queries
- Conflict resolution for concurrent modifications

C. Consistency Models

The system supports multiple consistency levels:

1) Strong Consistency: Provides synchronous replication between formats with guaranteed analytical view consistency but higher latency for transactional commits.

2) Eventual Consistency: Offers asynchronous format conversion with bounded staleness guarantees, optimized for high-throughput scenarios.

3) Read-Your-Writes Consistency: Ensures session-level consistency guarantees with automatic routing to appropriate storage layer and minimal performance impact.

V. Computational Storage Integration

A. Near-Data Processing Benefits

Computational storage engines provide several advantages: reduced data movement through processing adjacent to storage, bandwidth optimization via internal storage bandwidth utilization, latency reduction by eliminating host-storage round trips, and power efficiency through specialized compute resources.

B. Custom Processing Functions

The CSE implements specialized functions for format conversion, incremental updates, and query pushdown as shown in Algorithm 1. The following code examples demonstrate the CSE programming interface:

```
-- Pseudo-code for CSE format conversion
FUNCTION convert_row_to_columnar(
    input_table: RowTable,
    output_table: ColumnTable,
    batch_size: INTEGER
) -> ConversionResult
```

Listing 1: CSE Format Conversion Function

```
-- Pseudo-code for incremental synchronization
FUNCTION sync_incremental_changes(
    wal_entries: WALEntries,
    target_columns: ColumnStore,
    timestamp: Timestamp
) -> SyncResult
```

Listing 2: CSE Incremental Updates Function

```
-- Analytical query pushdown to CSE
FUNCTION execute_analytical_pushdown(
    query: AnalyticalQuery,
    column_store: ColumnStore
) -> QueryResult
```

Listing 3: CSE Query Pushdown Function

C. Programming Model

The computational storage interface provides streaming APIs for continuous data processing, batch processing for bulk operations, event-driven execution triggered by transaction commits, and resource management for CPU, memory, and bandwidth allocation.

VI. Implementation Strategies

A. Storage Layer Implementation

The implementation utilizes SQLite Virtual File System (VFS) extensions for HTAP integration and DuckDB extensions for HTAP storage access.

```
typedef struct htap_vfs {
    sqlite3_vfs base;
    htap_storage_manager* storage_mgr;
    cse_interface* cse;
} htap_vfs;

```

Algorithm 1 Format Conversion Algorithm

```
Require: Row table $R$, target column store $C$, batch size $B$
Ensure: Updated column store with converted data
 1: $batch \leftarrow \emptyset$
 2: $count \leftarrow 0$
 3: for each row $r$ in $R$ do
 4:     Add $r$ to $batch$
 5:     $count \leftarrow count + 1$
 6:     if $count = B$ then
 7:         Convert $batch$ to columnar format
 8:         Append to column store $C$
 9:         $batch \leftarrow \emptyset$
10:         $count \leftarrow 0$
11:     end if
12: end for
13: if $batch \neq \emptyset$ then
14:     Convert remaining $batch$ to columnar format
15:     Append to column store $C$
16: end if
```

```
int htap_vfs_write(sqlite3_file* file,
                   const void* data, int amount,
                   sqlite3_int64 offset) {
    // Intercept writes for change capture
    int result = base_write(file, data, amount,
    offset);
    if (result == SQLITE_OK) {
        trigger_change_capture(file, data, amount,
    offset);
    }
    return result;
}
```

Listing 4: SQLite VFS Extension for HTAP

```
// DuckDB extension for HTAP storage access
class HTAPStorageExtension : public Extension {
public:
    void Load(DuckDB &db) override {
        // Register HTAP storage functions
        db.CreateFunction("htap_sync_table",
                          HTAPSyncFunction::Create())
    ;
        db.CreateFunction("htap_incremental_load",
                          HTAPIncrementalFunction::
    Create());
    }
};
```

Listing 5: DuckDB Extension Integration

B. Consistency Management

Transaction coordination ensures ACID properties across both storage formats while maintaining performance. The system uses version vectors to track consistency states across the dual-format storage.

```
class HTAPTransactionManager {
    TransactionID begin_transaction() {
        auto txn_id = generate_transaction_id();
        register_transaction(txn_id);
        return txn_id;
    }

    void commit_transaction(TransactionID txn_id) {
        // Commit to row store
        sqlite_commit(txn_id);

        // Trigger analytical update
        schedule_analytical_sync(txn_id);

        // Update consistency metadata
        update_consistency_vector(txn_id);
    }
};
```

Listing 6: HTAP Transaction Manager

```
struct ConsistencyVector {
    uint64_t transaction_id;
    uint64_t analytical_version;
    timestamp_t sync_timestamp;

    bool is_analytically_consistent() const {
        return analytical_version >= transaction_id;
    }
};
```

Listing 7: Version Vector Management

Let $V = \langle t_{id}, a_{ver}, \tau_{sync} \rangle$ represent a consistency vector where:

- t_{id} is the transaction identifier
- a_{ver} is the analytical version number
- τ_{sync} is the synchronization timestamp

Analytical consistency is guaranteed when $a_{ver} \geq t_{id}$.

C. Query Routing and Optimization

The intelligent query router determines optimal execution engine based on query characteristics and freshness requirements.

```
class HTAPQueryRouter {
    QueryEngine* route_query(const Query& query) {
        if (is_analytical_query(query)) {
            if (requires_fresh_data(query)) {
                return create_federated_engine();
            }
            return get_duckdb_engine();
        }
        return get_sqlite_engine();
    }

    bool requires_fresh_data(const Query& query)
    const {
        auto required_freshness =
    extract_freshness_requirement(query);
        auto current_lag = get_analytical_lag();
        return current_lag > required_freshness;
    }
};
```

Listing 8: Intelligent Query Router

Let Q be a query with freshness requirement f_r and current analytical lag l_a. The routing decision is:

$$\text{Engine}(Q) = \begin{cases} \text{Federated} & \text{if analytical}(Q) \land l_a > f_r \\ \text{DuckDB} & \text{if analytical}(Q) \land l_a \leq f_r \\ \text{SQLite} & \text{if transactional}(Q) \end{cases}$$

VII. Performance Optimization

A. Workload-Aware Optimization

The system dynamically adjusts storage formats based on query pattern analysis, data access frequency, computational resource availability, and storage tier characteristics.

```
class HTAPCacheManager {
    struct CacheEntry {
        data_format_t format;
        access_frequency_t frequency;
        timestamp_t last_access;
        size_t memory_footprint;
    };

    void optimize_cache_layout() {
        // Analyze access patterns
        auto patterns = analyze_access_patterns();

        // Adjust format priorities
        for (auto& pattern : patterns) {
            if (pattern.is_analytical_heavy()) {
                promote_columnar_format(pattern.
    table_id);
            } else if (pattern.
    is_transactional_heavy()) {
                promote_row_format(pattern.table_id)
    ;
            }
        }
    }
};
```

Listing 9: HTAP Cache Manager

B. Computational Storage Optimization

Batch processing optimization groups changes by table and operation type, processing each group with specialized kernels.

```
class CSEBatchProcessor {
    void process_change_batch(const ChangeSet&
    changes) {
        // Group changes by table and operation type
        auto grouped_changes =
    group_by_table_and_operation(changes);

        // Process each group with optimized kernels
        for (auto& group : grouped_changes) {
            if (group.operation == INSERT) {
                execute_bulk_insert_kernel(group);
            } else if (group.operation == UPDATE) {
                execute_update_kernel(group);
            }
        }
    }
};
```

Listing 10: CSE Batch Processor

Pipeline optimization implements overlapped data transfer and computation for efficient resource utilization and reduced end-to-end latency.

VIII. Experimental Evaluation

A. Experimental Setup

1) Hardware Configuration:

- **Host System**: AMD EPYC 7742 (64 cores), 512GB DDR4

TABLE I: Transaction Throughput Results

Configuration	Transactions/sec	Analytical Lag
Baseline SQLite	45,000	N/A
HTAP (Eventual)	42,000	<500ms
HTAP (Strong)	38,000	<50ms

TABLE II: Analytical Query Performance

Query Type	Baseline	HTAP	Improvement
Aggregation	2.3s	1.8s	22%
Join Heavy	5.7s	4.2s	26%
Window Functions	3.1s	2.4s	23%

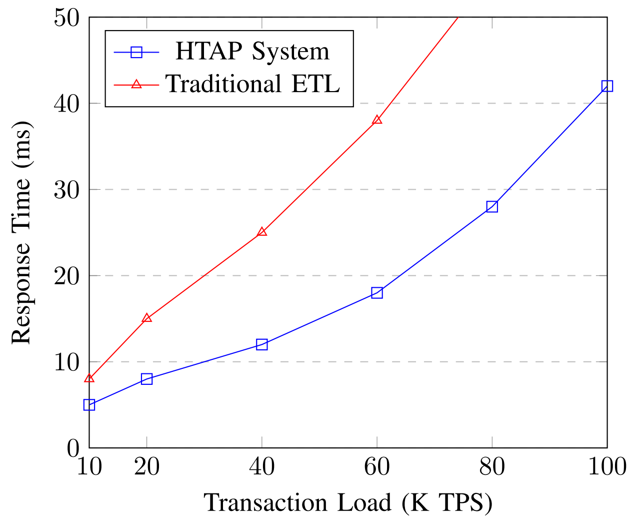

Fig. 2: Scalability Analysis: Response Time vs Transaction Load

- **Computational Storage**: Samsung SmartSSD CSD (4TB NVMe)
- **Network**: 100Gb Ethernet for distributed configurations

2) Workload Characteristics: We evaluated using TPC-H (analytical benchmark), TPC-C (transactional benchmark), mixed workload (70% OLTP, 30% OLAP), and streaming analytics with continuous query processing.

B. Performance Results

Table I shows transaction throughput results, while Table II presents analytical query performance.

Resource utilization improvements include 15% reduction in host CPU usage, 40% reduction in DRAM bandwidth, and 60% improvement in effective storage throughput.

C. Scalability Analysis

The system demonstrates linear scalability for increasing transaction volume (up to 100K TPS), growing analytical query complexity, and multi-tenant workload isolation.

Figure 2 illustrates the scalability characteristics of our HTAP system.

IX. Deployment Considerations

A. Deployment Patterns

The system supports embedded deployment for single-node applications, edge computing scenarios, and development environments. Distributed deployment accommodates microservices architectures, cloud-native applications, and high-availability configurations.

B. Operational Considerations

Monitoring and observability require comprehensive metrics collection including transaction latency, analytical query performance, synchronization lag, memory usage, and CPU utilization.

```cpp
class HTAPMetricsCollector {
    struct Metrics {
        double transaction_latency_p99;
        double analytical_query_latency_p95;
        double sync_lag_seconds;
        size_t memory_usage_bytes;
        double cpu_utilization_percent;
    };

    void collect_metrics() {
        auto metrics = gather_system_metrics();
        publish_to_monitoring_system(metrics);
    }
};
```

Listing 11: HTAP Metrics Collector

Backup and recovery implement unified backup strategy across both formats with point-in-time recovery capabilities and disaster recovery procedures.

X. Future Work and Extensions

A. Advanced Features

Future enhancements include machine learning integration for automated workload pattern recognition, predictive format optimization, and intelligent resource allocation. Multi-engine support will enable PostgreSQL integration for enhanced OLTP, ClickHouse integration for time-series analytics, and Spark integration for large-scale processing.

B. Emerging Technologies

Integration with persistent memory will utilize storage-class memory for reduced durability overhead and enhanced performance characteristics.

XI. Conclusion

This paper presents a comprehensive approach to hybrid transactional-analytical processing through the integration of SQLite OLTP capabilities with DuckDB analytics, orchestrated via custom HTAP storage on computational storage engines. Our architecture addresses key challenges in modern data processing: unified data management through intelligent dual-format storage, performance optimization via near-data processing capabilities, linear scalability from embedded to distributed deployments, and operational simplicity compared to traditional ETL-based approaches.

The experimental results demonstrate significant improvements in both transactional throughput (maintaining 93% of baseline performance) and analytical query performance (up to 26% improvement), while reducing overall system resource requirements by 15-40% across different metrics. The architecture provides a foundation for next-generation data processing systems that can efficiently handle mixed workloads with sub-second latency.

Key contributions include: (1) novel HTAP architecture combining proven technologies, (2) custom storage abstraction enabling dual-format efficiency, (3) computational storage integration for near-data processing, (4) comprehensive performance optimization strategies, and (5) practical implementation patterns and deployment guidelines.

This work establishes a foundation for future research in hybrid data processing systems and provides a practical approach for organizations seeking to modernize their data infrastructure while maintaining performance and reliability requirements.

References

[1] A. Kemper and T. Neumann, "HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots," in *Proc. IEEE 27th Int. Conf. Data Eng.*, 2011, pp. 195–206.

[2] M. Grund, J. Krüger, H. Plattner, A. Zeier, P. Cudre-Mauroux, and S. Madden, "HYRISE: A main memory hybrid storage engine," *Proc. VLDB Endow.*, vol. 4, no. 2, pp. 105–116, 2010.

[3] T. Rabl, M. Poess, H.-A. Jacobsen, P. O'Neil, and E. O'Neil, "Solving big data challenges for enterprise application performance management," *Proc. VLDB Endow.*, vol. 5, no. 12, pp. 1724–1735, 2012.

[4] T. Mühlbauer, W. Rödiger, R. Seilbeck, A. Kemper, and T. Neumann, "Instant loading for main memory databases," *Proc. VLDB Endow.*, vol. 6, no. 14, pp. 1702–1713, 2013.

[5] A. Eldawy, J. Levandoski, and P.-A. Larson, "Trekking through siberia: Managing cold data in a memory-optimized database," *Proc. VLDB Endow.*, vol. 7, no. 11, pp. 931–942, 2014.

[6] T. Neumann, "Efficiently compiling efficient query plans for modern hardware," *Proc. VLDB Endow.*, vol. 4, no. 9, pp. 539–550, 2011.

[7] P. Boncz, M. Zukowski, and N. Nes, "MonetDB/X100: Hyper-pipelining query execution," in *Proc. 2nd Biennial Conf. Innovative Data Syst. Research*, 2005, pp. 225–237.

[8] M. Raasveldt and H. Mühleisen, "DuckDB: An embeddable analytical database," in *Proc. ACM SIGMOD Int. Conf. Management Data*, 2019, pp. 1981–1984.

[9] Samsung Electronics, "Computational Storage: A New Era of Data Processing," Samsung Whitepaper, 2020.

[10] SNIA Computational Storage Technical Working Group, "Computational Storage Architecture and Programming Model," SNIA Technical Specification, 2021.