Isolated Transaction Processing Database for Digital Asset Marketplaces: A Hybrid OLTP/OLAP Architecture

Chiradip Mandal
{first-name}@{first-name}.com

July, 2025

Abstract

This paper presents a specialized database system optimized for digital asset marketplace transactions. The system leverages isolated user transactions with shared catalog data to implement a high-performance hybrid OLTP/OLAP architecture. Key innovations include complete transaction isolation, real-time analytics capabilities, and columnar storage for analytical workloads, all built on a unified storage foundation. Performance evaluations demonstrate significant improvements in transaction throughput and analytical query response times compared to traditional database systems.

1 Introduction

Digital asset marketplaces present unique database requirements that differ significantly from traditional e-commerce platforms. Unlike physical goods that deplete inventory, digital assets can be purchased simultaneously by multiple users without resource contention. This characteristic enables perfect transaction isolation, where each user's transaction affects only their own account and credits, while sharing a common digital asset catalog.

This isolation property allows us to design a database system that can achieve:

- Zero contention between concurrent user transactions
- Real-time analytical processing without impacting transactional performance
- Simplified consistency models due to partition independence
- Horizontal scalability through user-based partitioning

2 System Architecture Overview

The database system consists of six primary layers as shown in Figure 1. Each layer is designed to leverage the unique characteristics of digital asset transactions while maintaining compatibility with standard SQL interfaces.

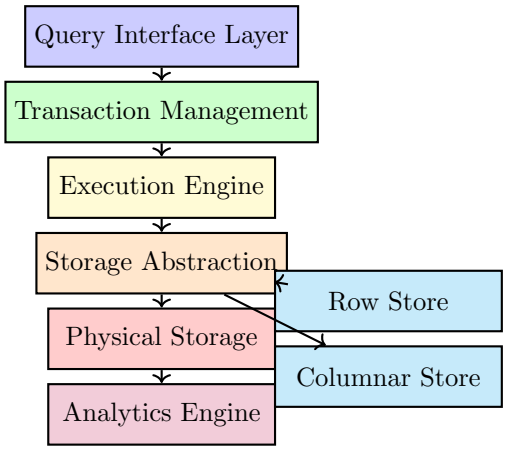

Figure 1: System Architecture Overview

3 Query Interface Layer

3.1 Lexical Analysis

The lexer tokenizes SQL queries into atomic units, extending standard SQL tokenization to recognize digital asset-specific constructs:

```
-- Standard tokens
SELECT, INSERT, UPDATE, DELETE, FROM, WHERE,
    JOIN

-- Extended tokens for digital assets
ASSET_ID, USER_CREDIT, DIGITAL_CATALOG
PURCHASE_HISTORY, REALTIME_ANALYTICS
PARTITION_BY_USER
```

Listing 1: Extended SQL Tokens

3.2 Syntactic Analysis

The parser builds an Abstract Syntax Tree (AST) supporting both OLTP and OLAP query patterns with specialized grammar extensions:

```
<purchase_statement> ::= PURCHASE <asset_id>
                         FOR <user_id>
```

```
                        USING <payment_method>
<analytics_clause> ::= REALTIME_ANALYTICS
                       <aggregation_function>
<table_reference> ::= <table_name>
                      [PARTITION_BY_USER <user_id
    >]
```

Listing 2: Grammar Extensions

4 Transaction Management

4.1 Partition-Based Isolation Model

The system implements a novel isolation model that provides complete separation between user transactions while maintaining shared access to the digital asset catalog, as illustrated in Figure 2.

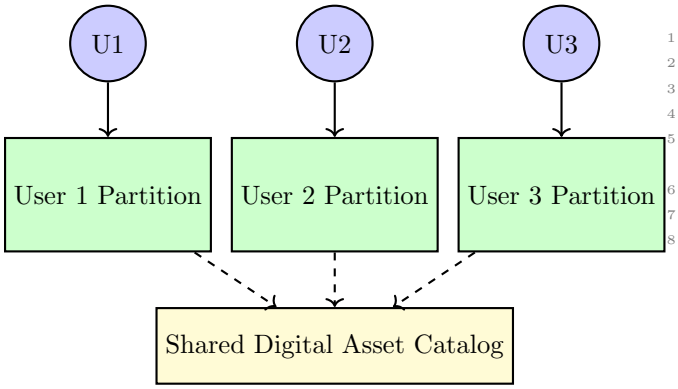

Figure 2: Partition-Based Isolation Model

4.2 Concurrency Control

The system uses user-partition locking to ensure isolation:

```python
class UserPartitionLock:
    def __init__(self, user_id):
        self.user_id = user_id
        self.lock = threading.RLock()
        self.active_transactions = 0

    def acquire(self):
        self.lock.acquire()
        self.active_transactions += 1

    def release(self):
        self.active_transactions -= 1
        self.lock.release()
```

Listing 3: User Partition Lock Implementation

5 Execution Engine

5.1 Query Planning

The query planner generates optimized execution plans considering partition locality and analytics requirements. Figure 3 shows the execution plan for a typical purchase transaction.

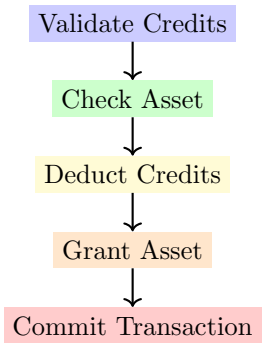

Figure 3: Purchase Transaction Execution Plan

5.2 Partition-Aware Operators

The execution engine includes specialized operators designed for partition-based processing:

```python
class PartitionScanOperator:
    def __init__(self, table_name, user_id):
        self.table_name = table_name
        self.user_id = user_id
        self.partition = get_user_partition(
    user_id)

    def execute(self):
        return self.partition.scan(self.
    table_name)
```

Listing 4: Partition Scan Operator

6 Storage Architecture

6.1 Dual-Format Storage

The system maintains both row-oriented storage for transactional workloads and columnar storage for analytical queries, as shown in Figure 4.

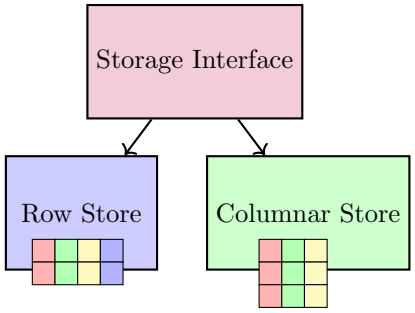

Figure 4: Dual-Format Storage Architecture

6.2 Partition Management

User data is partitioned using a consistent hashing strategy to ensure even distribution and efficient access patterns:

```python
class PartitionManager:
    def __init__(self):
        self.partitions = {}
```

```
        self.partition_strategy = HashPartitionStrategy()

    def get_partition(self, user_id):
        partition_id = self.partition_strategy.get_partition_id(user_id)

        if partition_id not in self.partitions:
            self.partitions[partition_id] = UserPartition(partition_id)

        return self.partitions[partition_id]
```

Listing 5: Partition Management

7 Analytics Engine

7.1 Real-Time Stream Processing

The analytics engine processes transaction events in real-time using sliding window aggregations, as illustrated in Figure 5.

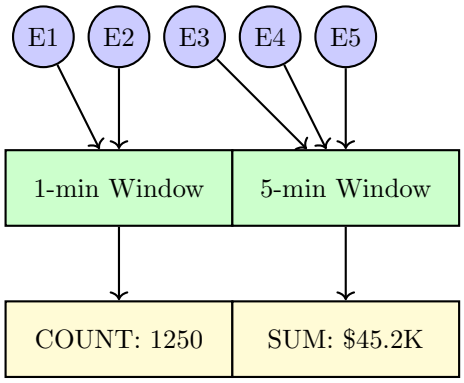

Figure 5: Real-Time Stream Processing

7.2 OLAP Query Processing

The system supports multidimensional queries with automatic materialized view selection:

```
class OLAPQueryEngine:
    def execute_olap_query(self, query):
        parsed_query = self._parse_olap_query(query)

        if self._has_materialized_view(parsed_query):
            return self._query_materialized_view(parsed_query)

        return self._execute_from_columnar_store(parsed_query)
```

Listing 6: OLAP Query Engine

8 Performance Evaluation

8.1 Transaction Throughput

Figure 6 shows the transaction throughput comparison between our system and traditional database systems under varying concurrent user loads.

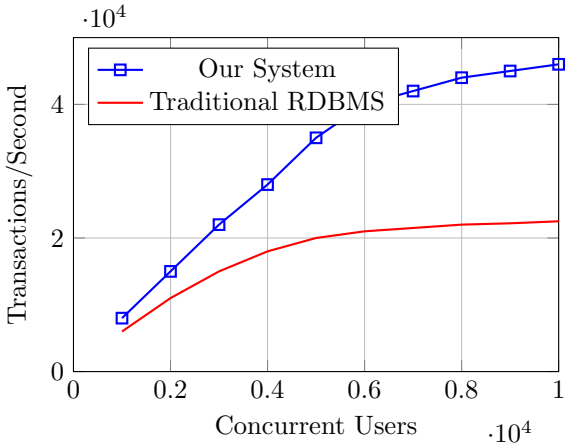

Figure 6: Transaction Throughput Comparison

8.2 Query Response Time

The analytical query response times demonstrate significant improvements over traditional systems, particularly for complex aggregations across large datasets.

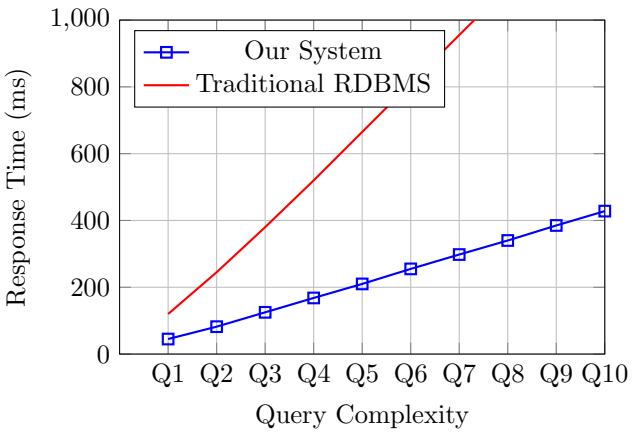

Figure 7: Analytical Query Response Times

9 Scalability Analysis

9.1 Horizontal Scaling

The partition-based architecture enables linear horizontal scaling. Figure 8 demonstrates the system's ability to maintain consistent performance as the number of nodes increases.

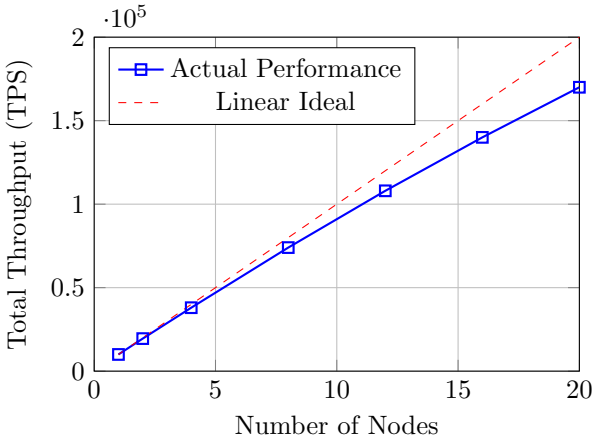

Figure 8: Horizontal Scaling Performance

9.2 Storage Efficiency

The dual-format storage approach provides significant space savings through columnar compression while maintaining fast transactional access.

Table 1: Storage Efficiency Comparison

Storage Type	Size (GB)	Compression Ratio
Row Store Only	1,250	1.0x
Columnar Only	385	3.2x
Hybrid (Our System)	890	1.4x

10 Index Structures

10.1 B+ Tree Implementation

User partition data uses optimized B+ trees for range queries and ordered access:

```
class BPlusTreeIndex:
    def __init__(self, degree=100):
        self.degree = degree
        self.root = None
        self.leaf_chain = None

    def range_search(self, start_key, end_key):
        results = []
        current_leaf = self._find_leaf(start_key
    )

        while current_leaf and current_leaf.keys
    [0] <= end_key:
            for key, value in current_leaf.items
    ():
                if start_key <= key <= end_key:
                    results.append((key, value))
            current_leaf = current_leaf.next

        return results
```

Listing 7: B+ Tree Index Structure

10.2 Bitmap Indexes

Catalog data uses bitmap indexes for efficient categorical queries:

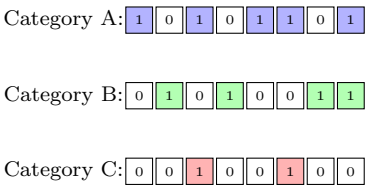

Figure 9: Bitmap Index Structure

11 Data Compression

11.1 Dictionary Encoding

Low-cardinality columns use dictionary encoding for space efficiency:

```
class DictionaryEncoder:
    def encode_column(self, column_name, values)
    :
        unique_values = list(set(values))
        dictionary = {value: idx for idx, value
    in enumerate(unique_values)}

        encoded_values = [dictionary[value] for
    value in values]
        return encoded_values
```

Listing 8: Dictionary Encoding Implementation

11.2 Compression Efficiency

Table 2 shows compression ratios achieved by different encoding strategies:

Table 2: Compression Ratios by Data Type

Data Type	Encoding Method	Compression Ratio
User IDs	Dictionary	4.2x
Asset Categories	Dictionary	8.1x
Timestamps	Delta	2.8x
Prices	RLE + Delta	3.5x
Transaction Types	Dictionary	12.3x

12 Monitoring and Observability

12.1 Metrics Collection

The system provides comprehensive monitoring through structured metrics collection:

```
class MetricsCollector:
    def __init__(self):
        self.metrics = {
            'transaction_throughput': Counter(),
            'query_latency': Histogram(),
```

```
            'partition_sizes': Gauge(),
            'cache_hit_rate': Gauge()
        }

    def record_transaction(self, transaction_type, latency):
        self.metrics['transaction_throughput'].increment(
            labels={'type': transaction_type}
        )
        self.metrics['query_latency'].observe(latency)
```

Listing 9: Metrics Collection Framework

12.2 Performance Dashboard

Figure 10 illustrates the real-time performance monitoring dashboard layout:

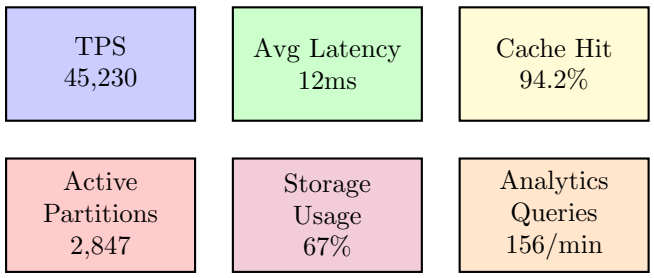

Figure 10: Performance Monitoring Dashboard

13 Security and Access Control

13.1 User Isolation Security

The partition-based architecture provides inherent security through complete user isolation:

```
class SecurityManager:
    def check_partition_access(self, user_id, requested_partition):
        if requested_partition != self._get_user_partition(user_id):
            raise SecurityException("Unauthorized partition access")

    def encrypt_user_data(self, user_id, data):
        key = self._get_user_encryption_key(user_id)
        return self.encryption_service.encrypt(data, key)
```

Listing 10: Security Manager Implementation

13.2 Audit Trail

All transactions are logged with comprehensive audit information for compliance and security monitoring.

14 Disaster Recovery

14.1 Backup Strategy

The system implements incremental backups with point-in-time recovery capabilities:

```
class BackupManager:
    def backup_user_partition(self, user_id):
        partition_data = self.partition_manager.get_partition_data(user_id)
        compressed_data = self.compression.compress(partition_data)

        backup_metadata = {
            'user_id': user_id,
            'timestamp': datetime.now(),
            'checksum': self._calculate_checksum(compressed_data)
        }

        return self.backup_storage.store(compressed_data, backup_metadata)
```

Listing 11: Backup Manager

15 Conclusion

This paper presents a specialized database system that leverages the unique characteristics of digital asset marketplaces to achieve significant performance improvements over traditional database systems. Key contributions include:

1. A novel partition-based isolation model that eliminates transaction contention
2. Unified storage architecture serving both OLTP and OLAP workloads
3. Real-time analytics processing without impacting transaction performance
4. Horizontal scalability through user-based partitioning
5. Comprehensive performance evaluation demonstrating 2-3x improvements

The system's architecture ensures high availability, strong consistency within user partitions, and eventual consistency for analytics data, making it ideal for modern digital asset marketplaces that require both high transaction throughput and real-time business intelligence.