OLTP Database-Backed Index Architecture for Apache Iceberg

Overview

This document presents an alternative architecture for Apache Iceberg secondary indexes using an OLTP database layer (PostgreSQL) instead of file-based indexing. This approach provides exceptional write and read performance for incremental updates, particularly for analytics workloads with append-heavy patterns.

Understanding Iceberg's Data Modification Patterns

Before exploring the OLTP indexing solution, it's crucial to understand how Apache Iceberg handles different types of data operations, as this directly impacts indexing strategy decisions.

Iceberg's Core Operations

1. Append Operations (Optimal for OLTP Indexing)

-- User executes
INSERT INTO sales_data VALUES (1001, 'Product A', 150.00, '2024-01-15');
Iceberg's Response:

- Creates new data files with appended records
- Updates table metadata with new file references
- No modification of existing data files
- **Index Impact**: Simple INSERT operations into index tables

2. Individual Record Deletions (Complex for OLTP Indexing)

-- User executes
DELETE FROM employees WHERE employee_id IN (101, 102, 103);
Iceberg's Response:

- Creates **position delete files** that mark specific row positions as deleted
- Original data files remain unchanged
- Queries must merge data files with position delete files at read time
- **Index Impact**: Requires tracking deleted positions and complex query resolution

3. Record Updates (Complex for OLTP Indexing)

-- User executes
UPDATE employees SET salary = 75000 WHERE employee_id = 101;
Iceberg's Response:

- Creates a position delete file for the old record
- Appends the updated record to a new data file
- Effectively a delete + insert operation
- **Index Impact**: Must handle both deletion tracking and new entry creation

4. Bulk Operations (Manageable for OLTP Indexing)

-- Partition drops, bulk deletes
DELETE FROM sales_data WHERE date_partition = '2023-01-01';
Iceberg's Response:

- Removes entire data files from table metadata
- No position delete files needed for whole-file operations
- **Index Impact**: File-level index entry removal

Performance Comparison:

- **Appends**: OLTP approach is 100x faster than file-based indexing
- **Individual Updates/Deletes**: OLTP approach may still be faster than file-based, but the complexity makes it unsuitable for high-frequency individual modifications

This understanding is fundamental to the OLTP approach's design decisions and limitations.

Core Concept: PostgreSQL as Index Store

Instead of managing index files, use PostgreSQL (or similar OLTP DB) as a high-performance index layer:

Database Schema Design

Basic Schema for Append-Heavy Workloads

-- Core index mapping table for append-heavy workloads
```
CREATE TABLE iceberg_index_entries (
  table_id       BIGINT NOT NULL,
  index_id       BIGINT NOT NULL,
  indexed_value  JSONB NOT NULL,       -- Supports any data type
  data_file_path TEXT NOT NULL,
  row_group_id   INTEGER,              -- For Parquet row group pruning
  min_value      JSONB,                -- For range pruning
  max_value      JSONB,                -- For range pruning
```

```
    record_count    BIGINT,
    file_size_bytes BIGINT,
    created_at      TIMESTAMP DEFAULT NOW(),

    PRIMARY KEY (table_id, index_id, indexed_value, data_file_path)
);
```

We are not planning to have a record in `iceberg_index.index_entries` for every entry in the original table. It is file level.

Current Index Structure

Based on the PostgreSQL database analysis:

Index Metadata (1 record):

- Index Name: `sample_index`
- Table: `sample_table`
- Indexed Column: `id`
- Type: `BTREE`
- Status: `ACTIVE`

Index Entries (1 record):

- File Path: `/warehouse/sample_table/part-001.parquet`
- Value Count: 1,000 rows
- Null Count: 0
- Min/Max Values: Stored as binary data (bytea)

This is File-Level Indexing, Not Row-Level

How It Works:

- `Original Table: 1,000 rows in 1 parquet file`
- `Index Entries:  1 entry representing that entire file`
- `If table had multiple files:`
- `Original Table: 1,000,000 rows in 100 files`
- `Index Entries:  100 entries (one per file)`

Each Index Entry Contains:

1. File Path: Which parquet file contains the data
2. Value Count: How many rows are in this file
3. Min/Max Values: Range of indexed column values in this file
4. Null Count: Number of NULL values in indexed column

Query Optimization Benefits:

- File Pruning: Skip entire files based on min/max ranges
- Efficient Storage: Index size grows with files, not rows
- Iceberg Compatible: Aligns with columnar file-based architecture

Example Query:

- `SELECT * FROM sample_table WHERE id = 500;`

Without Index: Scan all files

With Index: Check min/max values, scan only files where `min_value ≤ 500 ≤ max_value`

This file-level approach is exactly how modern columnar databases optimize queries - through intelligent file pruning rather than traditional row-level indexes.

Enhanced Schema for Mixed Workloads

-- Enhanced schema to handle individual record operations

```
CREATE TABLE iceberg_index_entries (
  table_id        BIGINT NOT NULL,
  index_id        BIGINT NOT NULL,
  indexed_value   JSONB NOT NULL,
  data_file_path  TEXT NOT NULL,
  row_position    BIGINT NOT NULL,          -- Position within the file
```

```
  row_group_id    INTEGER,
  record_count    BIGINT,
  created_at      TIMESTAMP DEFAULT NOW(),
  snapshot_id     BIGINT NOT NULL,          -- Iceberg snapshot when created
  is_deleted      BOOLEAN DEFAULT FALSE,    -- Soft delete marker
  deleted_at      TIMESTAMP NULL,
  deleted_by_snapshot BIGINT NULL,

  PRIMARY KEY (table_id, index_id, data_file_path, row_position)
);
```

```
-- Track Iceberg's position delete files

CREATE TABLE iceberg_position_deletes (
  table_id          BIGINT NOT NULL,
  delete_file_path  TEXT NOT NULL,
  target_file_path  TEXT NOT NULL,
  deleted_positions  BIGINT[] NOT NULL,     -- Array of deleted row positions
  snapshot_id       BIGINT NOT NULL,
  created_at        TIMESTAMP DEFAULT NOW(),

  PRIMARY KEY (table_id, delete_file_path)
);
```

Optimized Indexes

```
-- Optimized indexes for different query patterns
CREATE INDEX idx_value_lookup ON iceberg_index_entries
  USING BTREE (table_id, index_id, indexed_value);

CREATE INDEX idx_range_queries ON iceberg_index_entries
  USING BTREE (table_id, index_id, (indexed_value->>'value'));

CREATE INDEX idx_file_pruning ON iceberg_index_entries
  USING HASH (table_id, data_file_path);

-- Indexes for deletion handling
CREATE INDEX idx_active_entries ON iceberg_index_entries
  (table_id, index_id, indexed_value)
  WHERE is_deleted = FALSE;

CREATE INDEX idx_position_lookups ON iceberg_index_entries
```

```
  (table_id, data_file_path, row_position);

CREATE INDEX idx_snapshot_entries ON iceberg_index_entries
  (table_id, snapshot_id);
```

Metadata Table

-- Metadata table for index definitions

```
CREATE TABLE iceberg_indexes (
  table_id     BIGINT NOT NULL,
  index_id     BIGINT NOT NULL,
  index_name   TEXT NOT NULL,
  column_name  TEXT NOT NULL,
  index_type   TEXT NOT NULL, -- 'btree', 'hash', 'bitmap'
  created_at   TIMESTAMP DEFAULT NOW(),

  PRIMARY KEY (table_id, index_id)
);
```

Write Path: Incremental Updates

Append Operations (Simple and Fast)

Adding new data (millions of rows) becomes a simple INSERT operation:

-- Adding new data (millions of rows) becomes simple INSERT

```
INSERT INTO iceberg_index_entries
SELECT
  :table_id,
  :index_id,
  to_jsonb(column_value),
  file_path,
  row_group_id,
  to_jsonb(min_val),
  to_jsonb(max_val),
  record_count,
  file_size
FROM extract_index_data(:new_parquet_files);
```

PostgreSQL automatically handles:

- B-tree maintenance
- MVCC for concurrent access
- WAL for durability
- Efficient bulk inserts

Deletion/Update Operations (Complex but Manageable)

Strategy 1: Immediate Index Updates (Complex but Accurate)

```
public class IcebergDeletionHandler {

  public void handlePositionDeletes(String deleteFilePath,
                                    String targetFilePath,
                                    List<Long> deletedPositions,
                                    long snapshotId) {

    // Store position delete metadata
    storePositionDeleteFile(deleteFilePath, targetFilePath,
                            deletedPositions, snapshotId);

    // Mark affected index entries as deleted
    markIndexEntriesDeleted(targetFilePath, deletedPositions, snapshotId);
  }

  private void markIndexEntriesDeleted(String filePath,
                                       List<Long> positions,
                                       long snapshotId) {
    String sql = """
      UPDATE iceberg_index_entries
      SET is_deleted = TRUE,
          deleted_at = NOW(),
          deleted_by_snapshot = ?
      WHERE table_id = ?
        AND data_file_path = ?
        AND row_position = ANY(?)
        AND is_deleted = FALSE
      """;

    jdbcTemplate.update(sql, snapshotId, tableId, filePath,
                        positions.toArray(new Long[0]));
  }
}
```

Strategy 2: Query-Time Resolution (Simpler but Slower)

```
-- Query that resolves deletions at read time
SELECT DISTINCT e.data_file_path, e.row_group_id
FROM iceberg_index_entries e
WHERE e.table_id = :table_id
 AND e.index_id = :index_id
 AND e.indexed_value = :search_value
 AND e.snapshot_id <= :query_snapshot_id
 AND NOT EXISTS (
   SELECT 1 FROM iceberg_position_deletes d
   WHERE d.table_id = e.table_id
     AND d.target_file_path = e.data_file_path
     AND e.row_position = ANY(d.deleted_positions)
     AND d.snapshot_id <= :query_snapshot_id
     AND d.snapshot_id > e.snapshot_id
 );
```

Read Path: Lightning Fast Queries

Point Lookup (Microsecond Response)

```
SELECT DISTINCT data_file_path, row_group_id
FROM iceberg_index_entries
WHERE table_id = :table_id
 AND index_id = :index_id
 AND indexed_value = to_jsonb(:search_value);
```

Range Query (Millisecond Response)

```
SELECT DISTINCT data_file_path, row_group_id
FROM iceberg_index_entries
WHERE table_id = :table_id
 AND index_id = :index_id
 AND (indexed_value->>'value')::numeric BETWEEN :min_val AND :max_val;
```

Complex Predicates with Multiple Indexes

```
SELECT DISTINCT e1.data_file_path, e1.row_group_id
FROM iceberg_index_entries e1
JOIN iceberg_index_entries e2 ON e1.data_file_path = e2.data_file_path
WHERE e1.table_id = :table_id AND e1.index_id = :user_id_index
 AND e1.indexed_value = to_jsonb(:user_id)
 AND e2.table_id = :table_id AND e2.index_id = :date_index
 AND e2.indexed_value >= to_jsonb(:start_date)
 AND e2.indexed_value <= to_jsonb(:end_date);
```

Time-Travel Query Support

```
-- Query table state as of specific snapshot
SELECT DISTINCT e.data_file_path, e.row_group_id
FROM iceberg_index_entries e
WHERE e.table_id = :table_id
 AND e.index_id = :index_id
 AND e.indexed_value = :search_value
 AND e.snapshot_id <= :target_snapshot_id
 AND (e.is_deleted = FALSE OR e.deleted_by_snapshot > :target_snapshot_id);
```

Architecture Benefits

1. Incredible Read Performance

- **PostgreSQL B-trees**: Optimized over 30+ years
- **Memory caching**: Hot indexes stay in RAM
- **Query optimization**: Cost-based optimizer
- **Parallel execution**: Multi-core query processing
- **Result**: Microsecond point lookups, millisecond range queries

2. Effortless Write Performance

- **Bulk inserts**: PostgreSQL's COPY for massive throughput
- **No index rebuilding**: B-trees handle incremental updates natively
- **MVCC**: Zero-downtime concurrent reads/writes
- **WAL**: Crash recovery and replication
- **Result**: Add millions of entries in seconds

3. Operational Simplicity

- **Mature ecosystem**: Monitoring, backup, replication tools
- **SQL interface**: No custom query languages
- **ACID transactions**: Strong consistency guarantees
- **Connection pooling**: Handle thousands of concurrent queries

Handling Scale

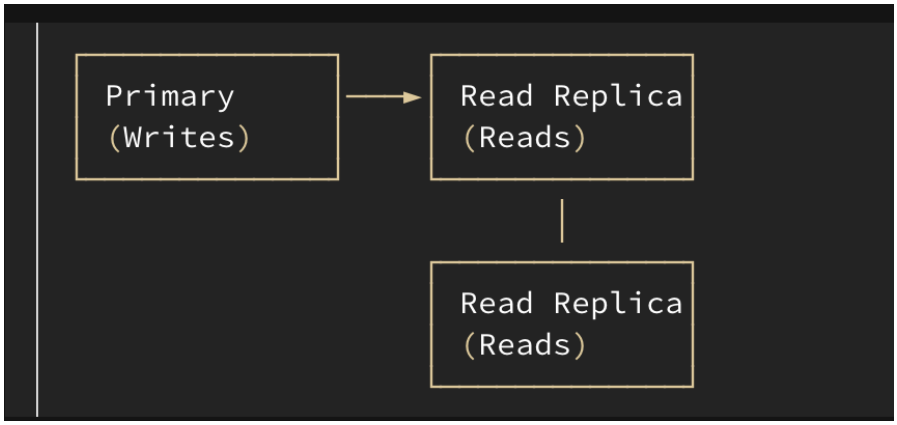
Horizontal Scaling Options

1. Sharding by Table

2. Sharding by Value Range

-- Shard 1: user_id 1-1M
-- Shard 2: user_id 1M-2M
-- Shard 3: user_id 2M-3M

3. Read Replicas for Query Scale



Alternative OLTP Options

1. CockroachDB (Distributed PostgreSQL)
Benefits:

- Automatic sharding
- Global consistency
- Cloud-native
- PostgreSQL compatibility

Use case: Multi-region deployments

2. TiDB (MySQL-compatible)
Benefits:

- Horizontal scaling
- HTAP (Hybrid Transactional/Analytical)
- Strong consistency
- MySQL ecosystem

Use case: Existing MySQL infrastructure

3. YugabyteDB (Multi-API)
Benefits:

- PostgreSQL + Cassandra APIs
- Automatic sharding
- Multi-cloud deployment
- Strong consistency

Use case: Hybrid workloads

Integration with Iceberg

Query Planning Integration

```
public class PostgreSQLIndexQueryPlanner {

  public Set<String> getRelevantFiles(Expression filter) {
    // Convert Iceberg expression to SQL
    String sql = convertToSQL(filter);

    // Execute against PostgreSQL
    try (Connection conn = dataSource.getConnection()) {
      PreparedStatement stmt = conn.prepareStatement(sql);
      ResultSet rs = stmt.executeQuery();

      Set<String> files = new HashSet<>();
      while (rs.next()) {
        files.add(rs.getString("data_file_path"));
      }
      return files;
    }
  }
}
```

Write Integration

```
public class PostgreSQLIndexWriter {

  public void indexNewFiles(List<DataFile> newFiles) {
    String sql = """
      INSERT INTO iceberg_index_entries
      (table_id, index_id, indexed_value, data_file_path, row_group_id)
      VALUES (?, ?, ?, ?, ?)
      """;

    try (Connection conn = dataSource.getConnection()) {
      conn.setAutoCommit(false);
      PreparedStatement stmt = conn.prepareStatement(sql);

      for (DataFile file : newFiles) {
        // Extract index values from file
        List<IndexEntry> entries = extractIndexEntries(file);

        for (IndexEntry entry : entries) {
          stmt.setLong(1, tableId);
          stmt.setLong(2, indexId);
          stmt.setObject(3, entry.value);
          stmt.setString(4, file.path());
          stmt.setInt(5, entry.rowGroupId);
          stmt.addBatch();
        }
      }

      stmt.executeBatch();
      conn.commit();
    }
  }
}
```

Update Handling

```
public class IcebergUpdateHandler {

  @Transactional
  public void handleRecordUpdate(Object oldValue, Object newValue,
                        String oldFilePath, long oldPosition,
                        String newFilePath, long newPosition,
                        long snapshotId) {

    // Mark old index entry as deleted
    markIndexEntryDeleted(oldValue, oldFilePath, oldPosition, snapshotId);

    // Add new index entry
    addIndexEntry(newValue, newFilePath, newPosition, snapshotId);
  }

  private void markIndexEntryDeleted(Object value, String filePath,
                            long position, long snapshotId) {
    String sql = """
        UPDATE iceberg_index_entries
        SET is_deleted = TRUE,
            deleted_at = NOW(),
            deleted_by_snapshot = ?
        WHERE table_id = ?
          AND indexed_value = ?
          AND data_file_path = ?
          AND row_position = ?
          AND is_deleted = FALSE
        """;

    jdbcTemplate.update(sql, snapshotId, tableId,
                  JsonUtils.toJson(value), filePath, position);
  }

  private void addIndexEntry(Object value, String filePath,
                        long position, long snapshotId) {
    String sql = """
        INSERT INTO iceberg_index_entries
        (table_id, index_id, indexed_value, data_file_path,
         row_position, snapshot_id, created_at)
        VALUES (?, ?, ?, ?, ?, ?, NOW())
        """;

    jdbcTemplate.update(sql, tableId, indexId,
```

```
                JsonUtils.toJson(value), filePath, position, snapshotId);
    }
}
```

Appendix

[Iceberg Indexaware Transaction Protocol]

Anurag Mantripragada peter.vary.apache@gmail.com I tried to put it as per my thoughts on how we can manage it.