Best practices for ACID tables

ACID (Atomicity, Consistency, Isolation, Durability) tables in Hive enable transaction support and ensure data consistency. Below are best practices for creating, managing, and querying ACID tables.

When to use and when to Avoid the Acid tables

Create ACID Tables : You need transactional support, frequent updates/deletes, data consistency, and support for complex operations like **MERGE, UPDATE,** and **DELETE**.
Avoid ACID Tables : Your data is append-only, performance is the priority, and you don't require frequent updates or transactional guarantees.

1. Accessing ACID Tables from Hive Queries

ACID Tables in Hive: To enable ACID transactions in Hive, you need to set the appropriate configurations and use the **INSERT, UPDATE, DELETE,** or **MERGE** operations on the table.

Creating ACID Tables in Hive

First, you need to create an ACID table in Hive. For that, ensure the table is created with the **transactional=true** property.

Accessing ACID Tables in Hive

1. **Inserting Data (ACID operations):** ACID tables support **INSERT INTO, INSERT OVERWRITE**, and can even handle transactional inserts.

2. **Update Data:** For updating data, ACID support is required. Use **UPDATE** or **MERGE.**

3. **Merge (Upsert):** A common operation for ACID tables is **MERGE**, which allows you to update existing records or insert new ones depending on a condition.

4. **Delete Data:** You can delete data in an ACID table.

Best Practices for Hive queries

1. Use partitioning and bucketing (depending on data size) to enhance query performance.
2. Execute regular major/minor compactions to handle delta files and optimize storage.
3. Secure tables with tools like Apache Ranger for fine-grained access control.

2. Accessing ACID Tables from Spark (Using Cloudera)

In Cloudera, Spark can access Hive ACID tables by using the **HiveContext** or the **spark.sql or** the **HWC** interface. To enable Spark to work with ACID tables, ensure that your Hive is correctly configured with ACID support, and Spark is able to access the Hive metastore.

1. Setting Up Spark to Access Hive ACID Tables

First, ensure the following configurations are set in Spark to work with Hive ACID tables:

1. **Enable Hive Support in Spark Session:**
   When creating the Spark session, enable Hive support:

   Eg: spark = SparkSession.builder \ .appName("ACID Tables in Spark") \ **.enableHiveSupport()** \
   .config("spark.sql.hive.hiveserver2.url",

"thrift://<hive_server_host>:<port>") \
.config("spark.sql.catalogImplementation", "hive") \ .getOrCreate()

2. Accessing ACID Tables from Spark SQL

After setting up Spark with Hive support, you can perform operations on ACID tables through Spark SQL.

1. **Reading Data from an ACID Table:**
   Spark can read from ACID tables just like regular Hive tables by using **spark.sql (SELECT * from <table_name>)** .
2. **Inserting Data into ACID Tables:**
   We can insert data into ACID tables using **INSERT INTO.**
3. **Updating Data in ACID Tables:**
   Spark supports **MERGE** operations for ACID tables, enabling it to perform upserts.
4. **Delete Data from ACID Tables:**
   Delete operations are also supported.

3. Accessing ACID Tables from HWC

1. Hive Warehouse Connector (HWC): Provides transactional consistency and is optimized for handling large-scale operations on ACID tables.
   Eg: from com.hortonworks.hwc import HiveWarehouseSession hive = HiveWarehouseSession.session(spark).build() df = hive.table("default.acid_table") df.show()

Best Practices for spark queries

1. Use HWC for scenarios requiring strict transactional consistency.
2. Avoid directly writing to ACID tables without compaction management to prevent fragmentation.
3. Optimize Spark jobs by setting appropriate session configurations for table access.

3. Creating and Updating Hive Acid Tables

Use Managed Tables for ACID: Hive ACID is supported on managed tables by default in Hive 3. External tables are not supported for ACID operations.
Transactional Table Design:

- Define primary keys when possible to ensure efficient updates and deletes.
- Choose appropriate file formats, such as ORC, for better performance and compression.

Avoid Legacy Ingest Patterns: Move away from legacy patterns like direct HDFS writes to ingestion mechanisms compatible with Hive ACID tables to maintain consistency and data integrity.
Use Dynamic Partitioning: Dynamic partition inserts are efficient but require enabling configurations such as **hive.exec.dynamic.partition.mode=nonstrict.** Avoid excessive partitioning, as it may impact performance.
Compaction Settings:

- Enable and schedule major/minor compactions to manage delta files effectively.
- Use **hive.compactor.initiator.on** and **hive.compactor.worker.threads** to configure compaction settings.

Schema Evolution: Use **ALTER TABLE** to add or drop columns, ensuring compatibility with existing data.

4. Querying ACID Tables

Query Optimizations:

- Use **ANALYZE TABLE** to gather statistics for query optimization.
- Enable the cost-based optimizer (CBO) for complex query plans.

Avoid Small File Issues: Use compaction and avoid direct file writes to ACID tables to prevent performance degradation due to small files.

Materialized Views and Caching: Leverage materialized views and query caching for repetitive reads.

5. Using Hive ACID Tables with Spark

- **Hive Warehouse Connector (HWC):** Use HWC to access Hive ACID tables from Spark. This ensures transactional guarantees while allowing Spark's distributed processing.
- **Avoid Direct Writes**: Avoid writing directly to Hive ACID tables via Spark unless using HWC, as this might break transactional integrity.
- **Efficient Reads**: Optimize Spark read queries by using partition pruning and explicitly selecting columns.

1. Modes of Accessing Hive ACID Tables with Spark

1.1 Hive Warehouse Connector (HWC)

- Use HWC to seamlessly read/write to Hive ACID tables while maintaining transactional integrity.

  Configuration:

  1. Add the HWC assembly JAR to your Spark application:
     a. **Eg:**--jars /path/to/hive-warehouse-connector-assembly-<version>.jar
  2. Enable HWC using Spark session properties :
     a. **Eg:** spark = SparkSession.builder .appName("HiveWarehouseConnectorExample") .config("spark.sql.hive.hiveserver2.jdbc.url", "jdbc:hive2://<HS2-URL>") .config("spark.datasource.hive.warehouse.read.mode", "DIRECT_READER") .enableHiveSupport() .getOrCreate()

3. **Direct Reader Mode (DIRECT_READER)** optimizes read queries by bypassing HiveServer2 and reading directly from the underlying storage, ensuring better performance for large datasets.

1.2 Using Direct Reader Mode

- Direct Reader Mode optimizes read-heavy workloads by bypassing HiveServer2 (HS2) and accessing table data directly from the Hive warehouse. This is a preferred approach when working with large volumes of data that are read frequently but not updated.

Configuration Best Practices for Direct Reader Mode

1. **Enable Direct Reader Mode**
   Set **spark.datasource.hive.warehouse.read.mode** to **DIRECT_READER** in the Spark session configuration:
   Eg: <property>
   <name>spark.datasource.hive.warehouse.read.mode</name>
   <value>DIRECT_READER</value> </property>

2. **Use Partition Pruning**
   Enable partition pruning to reduce the amount of data read by Spark:
   Eg:<property> <name>spark.sql.hive.metastorePartitionPruning</name>
   <value>true</value> </property>

3. **Manage Compactions**
   Regularly compact Hive ACID tables to reduce the number of delta files. This can be done via **ALTER TABLE <table_name> COMPACT 'MAJOR';** commands.

4. **Schema Handling**
   Direct Reader Mode does not support schema evolution, so ensure the schema is consistent across systems.

5. **Avoid Unsupported Operations**
   Do not use write operations (e.g., updates or deletes) in Direct Reader

mode. This mode is optimized for read-only operations only.

6. **Monitor and Tune Query Performance**
   Use query optimizations like **spark.sql.shuffle.partitions** to manage the number of partitions used in a job:
   Eg:<property> <name>spark.sql.shuffle.partitions</name> <value>200</value> </property>

Limitations of Direct Reader Mode

1. Direct Reader is suitable for reads but cannot handle transactional write operations.
2. Schema changes, especially those involving adding or dropping columns, require a different approach.

1.3 Using JDBC Mode

- JDBC Mode connects Spark to Hive via HiveServer2 (HS2), providing transactional consistency and supporting both read and write operations. This is ideal when you need to handle transactions (updates, deletes) or need features like schema evolution.

Configuration Best Practices for JDBC Mode

1. **Configure JDBC Connection**
   In Spark, set up the JDBC URL and credentials in your Spark session:
   Eg: spark = SparkSession.builder \ .appName("JDBCExample") \
   .config("spark.sql.hive.hiveserver2.jdbc.url",
   "jdbc:hive2://<hostname>:<port>") \
   .config("spark.sql.hive.hiveserver2.jdbc.user", "<username>") \
   .config("spark.sql.hive.hiveserver2.jdbc.password", "<password>") \
   .enableHiveSupport()

2. **Enable Transactional Support**
   JDBC mode allows you to perform both read and write operations with transactional guarantees. Ensure ACID properties are respected when performing DML operations:
   Eg: INSERT INTO TABLE <table_name> VALUES (...)

3. **Optimize Query Execution**
   Use **spark.sql.shuffle.partitions** to optimize the number of partitions used by Spark when performing large table scans:
   Eg: <property> <name>spark.sql.shuffle.partitions</name> <value>200</value> </property>

4. **Manage Schema Evolution**
   JDBC Mode supports schema evolution, so you can manage schema changes through the normal **ALTER TABLE** commands without worrying about data consistency issues.

5. **Use HiveServer2 for Transactional Operations**
   For updates or deletes on Hive ACID tables, always leverage HiveServer2 in JDBC mode to ensure consistency during these operations.

6. **Use JDBC Connection Pools**
   For high concurrency workloads, configure connection pooling to avoid the overhead of constantly opening and closing JDBC connections.

Limitations of JDBC Mode:

1. It can introduce additional latency due to network round trips between Spark and HiveServer2.
2. It may require more resources, especially for large-scale data writes.

6. General Recommendations

1. **Monitor Transactions and Locks**: Use **SHOW TRANSACTIONS** and **SHOW LOCKS** to monitor active transactions and diagnose potential issues.

2. **Governance and Security**:
   - Implement row-level security and column masking using Apache Ranger.
   - Avoid direct file access to managed tables to ensure ACID guarantees and compliance with governance policies.

3. **Data Retention and Archival**:
    - Implement compaction strategies for historical data.
    - Retain old partitions based on business requirements while using compacted delta files for efficient storage

7. DDL Management Best Practices

1. **Version Control**: Maintain versions of your DDL scripts in a repository for traceability.

2. **Validation and Automation:**
    - Automate DDL testing using CI/CD pipelines.
    - Validate schemas against a predefined standard to ensure compatibility.

3. **Comments and Metadata:** Add detailed comments and leverage table properties for documentation.

Implementing these best practices will help maintain the efficiency, reliability, and scalability of Hive ACID table operations in your data ecosystem.

Direct Reader Mode

In Cloudera, when using **Direct Reader Mode** for Spark to connect to **Hive non-ACID tables** via **Hive Warehouse Connector (HWC)**, it will indeed use **Direct Reader Mode** if explicitly specified in your Spark configuration.

Overview of Direct Reader Mode:

- **Direct Reader Mode** in Spark allows reading data from Hive tables without going through the traditional HDFS-based file system operations. This mode is efficient for reading large amounts of data, particularly from Hive managed tables that don't need to be altered (i.e., non-ACID tables) and improves performance by skipping unnecessary scans or data processing layers.
- It's particularly useful when working with **non-ACID** Hive tables (e.g., ORC, Parquet, etc.) where transactional support and updates are not required.

Key Concepts Of Spark Configuration:

To enable Direct Reader Mode in Spark, you need to ensure the following configuration is set when running Spark queries via HWC:

CONFIG: spark.conf.set("spark.datasource.hive.warehouse.read.mode", "DIRECT_READER_V2")

This tells Spark to use the **Direct Reader Mode V2** when accessing Hive tables, and it will use the optimized connector for reading the data.

What Happens for Non-ACID Hive Tables?

1. **If Direct Reader Mode is Configured**

- When using **Direct Reader Mode** and connecting to **non-ACID tables**, Spark will leverage the Hive Warehouse Connector to directly read from Hive tables efficiently, bypassing some traditional overhead, such as using MapReduce or full table scans.
- This mode should be explicitly configured, and once it's set, it will remain in effect for all the Hive queries in that session unless you override the configuration.

2. **If Direct Reader Mode is NOT Configured**

- If you do **not** explicitly configure Direct Reader Mode, Spark may use default behavior, which could involve full table scans using the traditional Hive interaction model, resulting in slower performance compared to Direct Reader Mode.
- Without this configuration, Spark may also rely on other methods like **HiveContext** for reading Hive tables, which will not utilize the optimized direct reading mechanism provided by HWC.

Default Behavior if Not Specified

By default, if **Direct Reader Mode** is not specified, Spark will **not** use Direct Reader Mode for non-ACID tables. Instead, it will fall back to the traditional **HiveContext** or **SparkSession** integration, which involves using the default Hive execution engine, possibly leading to performance degradation on large datasets.

Best Practice for Cloudera Environment

When working with **non-ACID Hive tables** in a **Cloudera** environment using **HWC** (Hive Warehouse Connector), it's highly recommended to explicitly set the **spark.datasource.hive.warehouse.read.mode to DIRECT_READER_V2** for optimal performance, especially if you're processing large amounts of data and don't require transactional support.
