RFC - 14 : JDBC incremental puller

Proposers

- Sagar Sumit

Approvers

- Vinoth Chandar **APPROVED**

Status

Current state: *IN PROGRESS*

Discussion thread:

JIRA: *here*

Released: <Hudi Version>

Abstract

To provide an alternate scalable ingestor to ingest data incrementally through JDBC and handle reconciliation.

Background

This approach would completely avoid the need to have a kafka cluster just to stream data at rest. Also present JDBC connect does not scale well for huge tables because of no distributed way of fetching partial data from a single table, leading to a single task per table. Unlike Sqoop which is a scalable incremental puller, we are going to avoid intermediate states and avoid extraneous data lifting action to the DAG.

Implementation

Motivation :

1. Supporting data sources which does not support bin logs(SAP HANA, Tibero, Teradata) but support sql.
2. Reducing resource wastage for batch based sync. For batch based sync, it is an overkill to stream data at rest using kafka.
3. Avoiding maintaining the kafka ecosystem and directly bringing data from sources.
4. JDBC connect is sequential in nature. One table can be loaded using a single task using JDBC connect.

We have identified major components for incremental JDBC puller.

1. A component to figure the optimal number of partitions to fetch data from source with an upper limit on the number of tasks and fetch size (rows per table).
2. Query builder based on incremental logic configured by the user. Strategy types can be
   a. Incrementing column
   b. Timestamp column
   c. Timestamp and incrementing columns
   d. Customer query-based
   e. Full refresh
3. Component to execute part operations independently with retry mechanism (spark map).
4. Component to handle schema evolution and database-specific type conversions.

The implementation will be divided into two phases. In the first phase, we want to add direct support to DeltaStreamer for incrementally pulling data from JDBC source. In this phase, we will not implement a component to figure the optimal number of partitions as mentioned in #1 in the preceding paragraph. Instead, we will rely on the `sourceLimit` to set the JDBC fetch size, which will determine how many rows to fetch per round trip during incremental fetching. In the second phase, we tackle the problem of intelligently figuring out the optimal number of partitions and fetch data in a distributed way from a single table.

Low Level Design (Phase 1)

Classes

- `JdbcSource: This class extends RowSource and implements fetchNextBatch(Option<String> lastCkptStr, long sourceLimit)`

to read data from an RDBMS table.

- `SqlQueryBuilder: A simple utility class to build sql queries fluently. It will have the following APIs:`
    - select(String... columns)
    - from(String... tables)
    - where(String predicate)
    - orderBy(String... columns)
    - limit(long count)

Configurations

- `hoodie.datasource.jdbc.url (REQUIRED)`

The JDBC URL to connect to.

- `hoodie.datasource.jdbc.user (REQUIRED)`
- `hoodie.datasource.jdbc.password`
- `hoodie.datasource.jdbc.password.file`

The username and password for the connection. The password can also be present in a file, in which case, provide the path to the file in hoodie.datasource.jdbc.password.file.

- `hoodie.datasource.jdbc.driver.class (REQUIRED)`

Driver class used for JDBC connection.

- `hoodie.datasource.jdbc.table.name`

  (REQUIRED)

Name of the table from which data will be pulled.

- `hoodie.datasource.jdbc.incremental.pull (REQUIRED)`

A boolean to indicate whether to do an incremental pull from JDBC source.

- `hoodie.datasource.jdbc.table.incremental.column.name`

If pulling incrementally, then the name of the incremental column. It could be a column with long values or a timestamp column.

- `hoodie.datasource.jdbc.storage.level`

Persistence level of the DataSet. It helps to avoid unnecessary table scans. By default, it will be set to *MEMORY_AND_DISK_SER.*

- `hoodie.datasource.jdbc.extra.options`

Any other options that match with Spark jdbc configurations.

Different modes of fetching

JdbcSource will support two modes of fetching:

- Full fetch: This is essentially a full table scan. This is done only when the last checkpoint is not known or there was some error during incremental fetch.
- Incremental fetch: This is done only when the last checkpoint is known. Records with the checkpoint column having value greater than the last checkpoint are fetched. In case of any error during incremental fetch, it will fall back to full fetch.

Incremental fetch supports a limit on the number of records to fetch in one round trip. The records are sorted by the incremental column in ascending order and then the limit is applied. This could lead to potential loss of records in certain scenarios as listed below:

Scenario 1: Suppose ‘ckpt’ is the incremental column and the last checkpoint (last_ckpt) is 10, while the fetch size is 100. Further suppose there were more than 120 records written to the table before the next interval of incremental fetch started. Then a simple query as below

```
select * from table_name where ckpt >= last_ckpt order by ckpt limit 100;
```

would fetch only the records with 10 < ckpt <= 110 (assuming ckpt is incrementing by 1). That means we lost the records with 110 < ckpt <= 120.

To handle this scenario, we run the query iteratively until no more records are left to pull.

```
fetchSize := 100

dataset := new DataSet()

resultData := new DataSet()

do {

  dataset := records with ckpt >= last_ckpt order by ckpt limit fetchSize

  last_ckpt := max(ckpt) in dataset

  resultData.union(dataset)

} while (!dataset.isEmpty());
```

NOTE: The above algorithm could run indefinitely in case of a source with high write throughput. So, we should have an upper limit based on sync interval or max seen checkpoint before starting the sync. For phase 1, as discussed in the comments, instead of running iteratively, we are going to just fetch once.

Scenario 2: Suppose ‘ckpt’ is the incremental column and the last checkpoint (last_ckpt) is 10. Further suppose there was a long running transaction in the database which wrote a record with ckpt=8. In this case, incremental fetch will pull all the records with ckpt > 10 and miss the one with ckpt=8. This scenario is **not handled** in Phase 1. Probably, we would need some kind of sweeper to look for such records in the background.

Scenario 3: A record is deleted in the source table. This would make the downstream table inconsistent with the source table. This scenario is also **not handled** in Phase 1. A workaround is to do insert_overwrite_table at regular interval. Another suggestion is to join with hudi dataset and find non-intersecting records, which could be expensive. In order to support such kind of CDC, we need to evaluate the pros and cons of both approaches, which we will take up in Phase 2.

Scenario 4: If the user chooses an `auto_increment` column as the incremental column, how do we handle updates? Auto-increment increments the value only in case of inserts. Updates are **not handled** in Phase 1. A workaround is to use `_updated_at` column as well in case the source table has such a column. Else, we might have to do batch sync at regular cadence for such use-cases.

From the above, we note that the current one-size-fits-all approach does not solve all the problems. We need to evaluate different data reconciliation strategies. However, Phase 1 brings immediate benefit to the users and also gives us a chance to get feedback while we evaluate along the way.

Rollout/Adoption Plan

- There won’t be any impact for existing users. This is just a new feature.
- New configurations will be added to the documentation.
- Add one JdbcSource example in the demo.

Test Plan

- Add a new unit test class for the JdbcSource.
- Run integration tests around RowSource and DeltaStreamer to cover JdbcSource.