Terminology

Throughout this document, the following terms are used:

- **Destination table** - the BigQuery table that the user would like data from Kafka to be published to via the connector
- **Intermediate table** - a BigQuery table that the connector may stream data to before it is made available in the destination table via a MERGE operation
- **Merge flush** - a operation that loads data from an intermediate table into a destination table
- **upsert/delete** (case insensitive) - upsert and/or delete

Public-Facing Changes

Configuration properties

The following configuration properties will be added to the connector:

Name	Type	Default	Importance	Behavior
upsertEnabled	BOOLEAN	false	HIGH	Enable upsert functionality on the connector through the use of record keys, intermediate tables, and periodic merge flushes. Row-matching will be performed based on the contents of record keys.
deleteEnabled	BOOLEAN	false	HIGH	Enable delete functionality on the connector through the use of record keys, intermediate tables, and periodic merge flushes. A delete will be performed when a record with a null value is read.

mergeIntervalMs	LONG	60000*	MEDIUM	How often (in milliseconds) to perform a merge flush, if upsert/delete is enabled. If set to -1, periodic time-based merge flushing will be disabled.
mergeRecordsThreshold	LONG	-1	MEDIUM	How many records to write per batch per table before triggering a merge flush, if upsert/delete is enabled. If set to -1, record count-based merge flushing will be disabled.

intermediateTableSuffix	STRING	"_tmp"	LOW	A suffix that will be combined with an underscore ("_") and appended to the names of destination tables to create the names for the corresponding intermediate tables. Multiple intermediate tables may be created for a single destination table, but their names will always start with the name of the destination table, followed by an underscore, followed by this suffix, and possibly followed by an additional suffix.

*** -** We may want to consider a higher default here, depending on how long we expect a merge to take, whether we will have to wait for data to exit the BigQuery streaming buffer, and any potential impact on the connector's performance.

Behavior

When upsert/delete is enabled, the connector will no longer stream data directly into destination tables. Instead, it will stream data into intermediate tables (the naming process for which is configurable in order to accommodate security and/or collision prevention concerns) and periodically (with configurable frequency) perform a MERGE operation (or a "merge flush") that loads data from the intermediate table into the destination table. After each merge flush, the intermediate table will be dropped.

In order to support this feature, the connector will require all incoming messages to have a non-null map or struct as their key, the fields of which will be used as the row-matching predicate for the MERGE.

Users will also have to specify a value for the existing kafkaKeyFieldName property if upsert/delete is enabled for the connector. If not, the connector will signal an error during configuration validation and the Kafka Connect framework will reject the configuration for the connector.

Proposed Implementation

Intermediate tables

Schema

The schema of the intermediate table will not be identical to the schema of the destination table; although this may seem strange at first, it's necessary in order to support deletes in cases where the destination table has non-null, non-key fields. Instead, the intermediate table's schema will consist of the key of the record wrapped in a struct named key, a nullable partitionTime TIMESTAMP-type field, a required batchNumber INT64-type field, and an additional nullable STRUCT column that will contain the row content for the table derived from the value of the Kafka record.

For example, if the destination table were defined with this schema (assuming a kafkaKeyFieldName of “customKeyField”):

```
CREATE TABLE `kcbq`.`upsertDeleteDemo` (
    customKeyField STRUCT<
        k1 INT64 NOT NULL,
        k2 STRING
    >,
    f1 STRING,
    f2 STRING NOT NULL
);
```

Then, assuming a temporary table suffix of “tmp” (disregarding the proposed UUID and epoch for readability’s sake), an intermediate table might be defined with this schema:

```
CREATE TABLE `kcbq`.`upsertDeleteDemo_tmp` (
    key STRUCT<
        k1 INT64 NOT NULL,
        k2 STRING
    >,
    value STRUCT<
        f1 STRING,
        f2 STRING NOT NULL
    >,
    partitionTime TIMESTAMP,
    i INT64 NOT NULL,
    batchNumber INT64 NOT NULL
);
```

This design accomplishes a few things:

- There is no schema information lost between the intermediate and the destination table; if and only if a row can be written to the intermediate table with the appropriate shuffling to populate the key struct and wrap its content in the value struct, it can also be written to the destination table. This ensures fail-fast behavior in the case of either schema evolution or malformed records.
- The row content derived from the value of the Kafka records read is nullable, which allows us to prepare to delete rows by setting the row struct to NULL
- Stores all relevant information in a single table, as opposed to splitting to-be-upserted and to-be-deleted rows across different tables
- The partitionTime field can be used for overwriting the ingestion time of rows during a merge flush so that they are sent to the correct partition
- The i field can be used for deduplication of rows that share the same key when upsert is enabled (only the row with the highest value for that column will be considered for each key, and the value will be monotonically increasing for every row that the connector processes for a given table)
- The batchNumber field can be used to track groups of rows as they're merged, which enables offset tracking and efficient cleanup of the intermediate table over its lifetime by deleting rows from it that correspond to old batches

Construction

Intermediate tables will be created with the name ${dest}_${suffix}_${task}_${uuid}_${epoch}, where ${dest} is the name of the destination table, ${suffix} is the user-configured temporary table suffix, ${task} is the ID of the task, ${uuid} is a UUID generated at startup, and ${epoch} is the epoch time in milliseconds at task startup.

It is imperative that no table with this name exists before the task starts writing. If one does exist and contains an incompatible schema, the task would fail to write a single row. Even if the user understood this and remembered to delete that table before starting the connector, BigQuery has an unfortunate bug at the moment where data written to deleted-then-immediately-recreated tables can be silently dropped during streaming inserts. As this would allow for the possibility of data loss, it is imperative that this scenario be avoided if at all possible; utilizing a UUID in conjunction with the epoch time at task startup should make it extraordinarily unlikely that any tables with that name exist currently or existed at any point.

After every merge flush, an attempt will be made to clear out old rows from the intermediate table by deleting rows with old batch numbers. The _PARTITIONTIME field of each row will be queried to ensure that only rows that are no longer in the streaming buffer are deleted; otherwise, the delete operation will fail. All intermediate tables will be created with ingestion-time partitioning by default in order to support this _PARTITIONTIME column-based strategy.

Intermediate tables will be automatically deleted by the task during shutdown. If the worker is killed suddenly or deletion requests fail for some reason, they will have to be deleted by users manually.

MERGE query construction

This is what a MERGE query for the intermediate and destination tables defined above would consist of, assuming both upsert and delete are enabled, and a batch number of 2:

```
MERGE `kcbq`.`upsertDeleteDemo`
  USING (
    SELECT * FROM (
      SELECT ARRAY_AGG(
        x ORDER BY i DESC LIMIT 1
      )[OFFSET(0)] src
      FROM `kcbq`.`upsertDeleteDemo_tmp` x
      WHERE batchNumber=2
      GROUP BY key.k1, key.k2
    )
  )
  ON `upsertDeleteDemo`.key=`src`.key
  WHEN MATCHED AND `src`.value IS NOT NULL
    THEN UPDATE SET f1=`src`.value.f1, f2=`src`.value.f2
  WHEN MATCHED AND `src`.value IS NULL
    THEN DELETE
  WHEN NOT MATCHED AND `src`.value IS NOT NULL
    THEN INSERT (key, _PARTITIONTIME, f1, f2)
    VALUES (
      `src`.key,
      CAST(CAST(DATE(`src`.rowTime) AS DATE) AS TIMESTAMP),
      `src`.value.f1,
      `src`.value.f2
    );
```

(Similar queries can be constructed for upsert-only and delete-only.)

For anyone not familiar with SQL or BigQuery's particular dialect of SQL, the query can be explained in English as:

- Construct a temporary table named src whose content is the result of selecting all rows with a batch number of 2 from `kcbq`.`upsertDeleteDemo_tmp`, then deduplicating to have a single row for each unique key, favoring the rows with the latest value for the i column as a tie-breaker
- For every row in the src table:
    - If that row's key column matches a row in the table upsertDeleteDemo in the dataset kcbq:

    - If the value column from the src table is null, delete the matched row from the table upsertDeleteDemo
    - Otherwise, update all of the non-key columns in the table upsertDeleteDemo to match the values in the row from the src table
- Otherwise, insert the key column and all of the values in the value column in the row from the src table into the upsertDeleteDemo table, using the rounded-off timestamp value of \`src\`.\`rowTime\` to determine which partition of the destination table each row should be sent to; if the destination table is not partitioned, then the MERGE query will not include the _PARTITIONTIME column

The manual deduplication of the intermediate table is necessary as the MERGE operation will fail if multiple rows are found in the source table with the same values for their keys.

Since BigQuery does not allow grouping by STRUCT columns, the values of the key field must be explicitly referenced as “key.k1” and “key.k2” instead of simply “key”. In order to handle keys whose schemas contain nested structs, some recursive logic will be necessary here.

Delivery guarantees and failure recovery

Data delivery guarantees for the connector should actually become stronger with upsert/delete enabled. Currently, no offsets are committed by a task until it has successfully written all records to Bigquery that it received since the last offset commit. If a task unexpectedly dies before committing offsets, or is only able to write some of its records to BigQuery, it will resume from the last-committed offsets and produce duplicate rows in BigQuery; this gives the connector **at-least-once** delivery guarantees. This is usually not an issue given that BigQuery idiom is to perform deduplication via custom query logic when reading data from a table. However, with upsert/delete enabled and only record count-based merge flushing, all writes will be deterministic, idempotent, and therefore effectively **exactly-once**. With time-based merge flushing, the delivery guarantees of the connector may become somewhat weaker, as it will be possible that a failed offset commit will cause old data to be rewritten to the destination table, temporarily overwriting newer data already present. However, as the connector progresses, the new data will again appear in the destination table in a sort of eventually-consistent fashion.

This idempotence allows us to be conservative about committing offsets. We can delay commits until we have successfully completed a merge flush, confirming that all relevant records have been successfully written to the destination table. If a merge flush fails partway through, or a task is unable to confirm that it has succeeded before being shut down by the framework, we can leave the offsets for those records uncommitted, and simply re-write them to BigQuery with no fear of creating duplicate records.

The current behavior of the connector is to asynchronously stream records into tables as they are received in SinkTask::put, and block on the completion of those writes in SinkTask::flush. We could do something similar with upsert/delete enabled, where records would be streamed asynchronously into intermediate tables during SinkTask::put and then SinkTask::flush would block on a merge flush for all of these; however, the frequency of merge flushes is likely to need to be tuned carefully by users to accommodate their specific needs for latency and throughput, and a large merge flush interval would block the connector unduly during this period. Since the framework's offset flush interval for connectors is only configurable on a per-worker level and not a per-connector level, we also can't expect users to tune the offset flush interval of their entire worker (and all connectors on it) to match the merge flush interval for their BigQuery sink connector. So, instead, we can track which records have been successfully written to the destination table via merge flush and adjust the offsets returned in SinkTask::preCommit accordingly.

If a task is started after being shut down ungracefully, no special state tracking or failure recovery logic will be required. It can simply begin reading from the last-committed offsets in Kafka for all topic-partitions it is meant to consume from.

Merge flushes

When a merge flush is triggered, the task will follow these steps in order:

1. Increment the batch number for the to-be-flushed table, so that new rows can and will still be written to that table during the merge flush, but with a new batch number
2. Ensure that all previous merge flushes on the to-be-flushed table have completed successfully
3. Ensure that all rows with the batch number for this flush have been streamed successfully into the intermediate table
4. Wait for a brief period (a few seconds) to ensure that those rows are readable from the streaming buffer
5. Execute the merge flush query from the intermediate table into the destination table
6. Mark the offsets for the records that were just merge flushed as ready to commit, so that they can be returned to the framework during the next call to SinkTask::preCommit
7. Request an offset commit from the framework via SinkTaskContext::requestCommit
8. Clean up the intermediate table by deleting from it all rows with a batch number less than or equal to that of the batch that was just flushed that are not still in the streaming buffer

Key schema changes

If the schema for a record key changes, the connector will attempt to update the schema of the destination table so that its key column can accurately correspond to the schema of incoming Kafka records. This will succeed if and only if the schema changes are backwards compatible; for BigQuery, this means that new nullable columns can be added, and existing columns can be modified from required to nullable. Other operations, such as changing a column's name or type, removing a column, or changing the mode of a column except from required to nullable, are not permitted. If this update fails, the task will fail and report that an incompatible schema change has occurred.

Interactions with existing features

Table schema updates

Automatic table schema updates should be as feasible with upsert/delete enabled as they currently are with it disabled.

When enabled, table schema updates are performed by tasks whenever an error response from BigQuery during a streaming insert is encountered that appears to be due to a mismatch between the schema of the to-be-inserted data and the schema of the table. (The exact logic for determining whether a row insert failed because of a schema mismatch is a little hacky but seems to have worked.)

With upsert/delete enabled, a similar approach can be taken for writes to the intermediate table: whenever a streaming insert fails and appears to have failed because of a schema mismatch, the task can attempt an update of the intermediate table's schema. If that succeeds, the task will also perform an update of the destination table's schema, and if either of these update attempts fail, the task will fail.

The task will also attempt to update the schema for the destination table whenever an intermediate table is created; this is to catch cases where the connector is stopped, the schema for data in a topic it's reading from is updated, then the connector is restarted.

Table creation

Automatic table creation should also still be possible with upsert/delete enabled.

The connector performs table creation with a similar technique to schema updates: when a streaming insert fails, and the cause appears to be the lack of that table in BigQuery, the connector attempts to create the table.

With upsert/delete enabled, this is a little trickier. We can't assume that just because the intermediate table we initially write to is missing, the destination table is also missing. Instead, we can first check to see if the destination table exists, and if not, attempt to create it. If the destination table already exists, we will want to verify that the schemas of the destination table and the intermediate table align appropriately and, if not, either attempt an update of the destination table's schema (if automatic schema updates are enabled), or fail the task. There is a possible race here where two tasks check to see if the destination table already exists, both see that it doesn't, then both attempt to create it; this should be acceptable as long as it can still be verified afterward that the schema of the destination table is compatible with the schema of the intermediate tables created by those tasks.

Table partitioning

Background

This part is a little tricky. BigQuery currently supports three basic types of table partitioning:

1. Ingestion-time partitioning, where tables are partitioned by the times at which rows are inserted
2. Date/timestamp partitioning, where tables are partitioned by the value of a data or timestamp column for each row
3. Integer range partitioning, where tables are partitioned by the value of an integer column for each row

Current behavior

The BigQuery connector has a few configuration properties that dictate how it interacts with table partitions. It currently only supports ingestion-time and date/timestamp partitioning, but it's possible that support for integer range partitioning may be added in the future.

When creating tables:

- If connector config contains a value for the timestampPartitionFieldName property, the connector will create tables with date/timestamp partitioning on that field
- Otherwise, the connector will create tables partitioned by ingestion time

When writing to tables:

- If the property bigQueryPartitionDecorator is set to true:
    - If the property useMessageTimeDatePartitioning is set to true, the connector will use BigQuery table decorator syntax (i.e., appending $yyyyMMdd to the end of the table) to write directly to the partitions for the timestamps of its sink records
    - Otherwise, the connector will use BigQuery table decorator syntax to target the partitions for the times that it reads sink records
- Otherwise, the connector writes to the table without specifying a particular partition

It should be noted that the timestampPartitionFieldName and bigQueryPartitionDecorator properties are mutually exclusive.

Behavior with upsert/delete

When creating destination tables:

- The connector will follow the same behavior with regards to the partitioning of the created table

When writing to tables:

- If the property bigQueryPartitionDecorator is set to true:
    - If the property useMessageTimeDatePartitioning is set to true, the timestamp for sink records will be used for the rowTime column
    - Otherwise, the current time will be used for the rowTime column
    - Either way, the partitionTime column from the intermediate table will be used to populate the _PARTITIONTIME pseudo-column for the destination table, which will cause each row to land in the expected partition. Ironically, the actual _PARTITIONTIME column from the intermediate table cannot be used here as it is NULL for rows still in the streaming buffer
- Otherwise, nothing special will happen as either the destination table won't be partitioned at all, or will be partitioned by its own row content

Inclusion of record key in row

The connector originally only supported writing the values of its sink records to BigQuery, but support was added later on to also write the data in the keys for those records as well.

If a value is specified for the kafkaKeyFieldName property, then all data from the record's key will be written to a column in BigQuery whose name is that value. If no value is specified, then the key data for all sink records is omitted from BigQuery.

With upsert/delete enabled, the kafkaKeyFieldName property will be required. If no value is specified, the connector will fail to start with a configuration error. This helps ensure that the data in the record key is already present in the schema for the destination table.