Streaming data into BigQuery

cloud.google.com/bigquery/streaming-data-into-bigquery

Instead of using a job to load data into BigQuery, you can choose to stream your data into BigQuery one record at a time by using the `tabledata.insertAll` method. This approach enables querying data without the delay of running a load job.

This document discusses several important trade-offs to consider before choosing an approach, including streaming quotas, data availability, and data consistency.

Note: For new projects, consider using the BigQuery Storage Write API (Preview) for streaming instead of the `tabledata.insertAll` method. The Storage Write API has lower pricing than the `insertAll` API and more robust features, including exactly-once delivery semantics. For more information, see Using BigQuery Storage Write API

Before you begin

1. Ensure that you have write access to the dataset that contains your destination table. The table must exist before you begin writing data to it unless you are using template tables. For more information on template tables, see Creating tables automatically using template tables.

2. Check the quota policy for streaming data.

3. Make sure that billing is enabled for your Cloud project. Learn how to confirm that billing is enabled for your project.

Streaming is not available via the free tier. If you attempt to use streaming without enabling billing, you receive the following error: `BigQuery: Streaming insert is not allowed in the free tier.`

Required permissions

At a minimum, to stream data into BigQuery, you must be granted the `bigquery.tables.updateData` permission. If you use a template table to create the table automatically, you must also have the `bigquery.tables.create` permission.

The following predefined Identity and Access Management (IAM) roles include both `bigquery.tables.updateData` and `bigquery.tables.create` permissions:

- `bigquery.dataEditor`
- `bigquery.dataOwner`
- `bigquery.admin`

For more information about IAM roles and permissions in BigQuery, see Predefined roles and permissions.

Checking for data availability

Streamed data is available for real-time analysis within a few seconds of the first streaming insertion into a table. In rare circumstances (such as an outage), data in the streaming buffer may be temporarily unavailable. When data is unavailable, queries continue to run successfully, but they skip some of the data that is still in the streaming buffer. These queries will contain a warning in the `errors` field of `bigquery.jobs.getQueryResults`, in the response to `bigquery.jobs.query` or in the `status.errors` field of `bigquery.jobs.get`.

Data can take up to 90 minutes to become available for copy operations. Also, when streaming to a partitioned table, data in the streaming buffer has a NULL value for the `_PARTITIONTIME` pseudo column. To see whether data is available for copy, check the `tables.get` response for a section named `streamingBuffer`. If that section is absent, your data should be available for copy, and should have a non-null value for the `_PARTITIONTIME` pseudo column. Additionally, the `streamingBuffer.oldestEntryTime` field can be leveraged to identify the age of records in the streaming buffer.

Best effort de-duplication

When you supply `insertId` for an inserted row, BigQuery uses this ID to support best effort de-duplication for up to one minute. This means that if you try to stream the same row with the same `insertId` more than once within that time period into the same table, BigQuery *may* de-duplicate the multiple occurrences of that row, retaining only one of those occurrences.

This is generally meant for retry scenarios in a distributed system where there's no way to determine the state of a streaming insert under certain error conditions, such as network errors between your system and BigQuery or internal errors within BigQuery. If you retry an insert, use the same `insertId` for the same set of rows so that BigQuery can attempt to de-duplicate your data. For more information, see troubleshooting streaming inserts.

De-duplication offered by BigQuery is best effort, and it should not be relied upon as a mechanism to guarantee the absence of duplicates in your data. Additionally, BigQuery might degrade the quality of best effort de-duplication at any time in order to guarantee higher reliability and availability for your data.

If you have strict de-duplication requirements for your data, Google Cloud Datastore is an alternative service that supports transactions.

Disabling best effort de-duplication

You can disable best effort de-duplication by not populating the `insertId` field for each row inserted. When you do not populate `insertId`, you get higher streaming ingest quotas in certain regions. This is the recommended way to get higher streaming ingest quota limits. For more information, see Quotas and limits.

Apache Beam and Dataflow

To disable best effort de-duplication when you use Apache Beam's BigQuery I/O connector for Java, use the `ignoreInsertIds()` method.

Streaming into time-partitioned tables

When you stream data to a time-partitioned table, each partition has a streaming buffer. The streaming buffer is retained when you perform a load, query, or copy job that overwrites a partition by setting the `writeDisposition` property to `WRITE_TRUNCATE`. If you want to remove the streaming buffer, verify that the streaming buffer is empty by calling `tables.get` on the partition.

Ingestion-time partitioning

When you stream to an ingestion-time partitioned table, BigQuery infers the destination partition from the current UTC time.

Newly arriving data is temporarily placed in the `__UNPARTITIONED__` partition while in the streaming buffer. When there's enough unpartitioned data, BigQuery partitions the data into the correct partition. A query can exclude data in the streaming buffer from a query by filtering out the `NULL` values from the `__UNPARTITIONED__` partition by using one of the pseudocolumns (`_PARTITIONTIME` or `_PARTITIONDATE` depending on your preferred data type).

If you are streaming data into a daily partitioned table, then you can override the date inference by supplying a partition decorator as part of the `insertAll` request. Include the decorator in the `tableId` parameter. For example, you can stream to the partition corresponding to 2021-03-01 for table `table1` using the partition decorator:

```
table1$20210301
```

When streaming using a partition decorator, you can stream to partitions within the last 31 days in the past and 16 days in the future relative to the current date, based on current UTC time. To write to partitions for dates outside these allowed bounds, use a load or query job instead, as described in Appending to and overwriting partitioned table data.

Streaming using a partition decorator is only supported for daily partitioned tables. It is not supported for hourly, monthly, or yearly partitioned tables.

For testing, you can use the `bq` command-line tool `bq insert` CLI command. For example, the following command streams a single row to a partition for the date January 1, 2017 (`$20170101`) into a partitioned table named `mydataset.mytable`:

```
echo '{"a":1, "b":2}' | bq insert 'mydataset.mytable$20170101'
```

Caution: The `bq insert` command is intended for testing only.

Time-unit column partitioning

You can stream data into a table partitioned on a `DATE`, `DATETIME`, or `TIMESTAMP` column that is between 5 years in the past and 1 year in the future. Data outside this range is rejected.

When the data is streamed, it is initially placed in the `__UNPARTITIONED__` partition. When there's enough unpartitioned data, BigQuery automatically repartitions the data, placing it into the appropriate partition.

Creating tables automatically using template tables

Template tables provide a mechanism to split a logical table into many smaller tables to create smaller sets of data (for example, by user ID). Template tables have a number of limitations described below. Instead, partitioned tables and clustered tables are the recommended ways to achieve this behavior.

To use a template table via the BigQuery API, add a `templateSuffix` parameter to your `insertAll` request. For the `bq` command-line tool, add the `template_suffix` flag to your `insert` command. If BigQuery detects a `templateSuffix` parameter or the `template_suffix` flag, it treats the targeted table as a base template, and creates a new table that shares the same schema as the targeted table and has a name that includes the specified suffix:

```
<targeted_table_name> + <templateSuffix>
```

By using a template table, you avoid the overhead of creating each table individually and specifying the schema for each table. You need only create a single template, and supply different suffixes so that BigQuery can create the new tables for you. BigQuery places the tables in the same project and dataset.

Tables created via template tables are usually available within a few seconds. On rare occasions they may take longer to become available.

Changing the template table schema

If you change a template table schema, all subsequently generated tables will use the updated schema. Previously generated tables will not be affected, unless the existing table still has a streaming buffer.

For existing tables that still have a streaming buffer, if you modify the template table schema in a backward compatible way, the schema of those actively streamed generated tables will also be updated. However, if you modify the template table schema in a non-backward compatible way, any buffered data that uses the old schema will be lost. Additionally, you will not be able to stream new data to existing generated tables that use the old, but now incompatible, schema.

After you change a template table schema, wait until the changes have propagated before you try to insert new data or query generated tables. Requests to insert new fields should succeed within a few minutes. Attempts to query the new fields might require a longer wait of up to 90 minutes.

If you want to change a generated table's schema, do not change the schema until streaming via the template table has ceased and the generated table's streaming statistics section is absent from the `tables.get()` response, which indicates that no data is buffered on the table.

Partitioned tables and clustered tables do not suffer from the above limitations and are the recommended mechanism.

Template table details

Template suffix value
The `templateSuffix` (or `--template_suffix`) value must contain only letters (a-z, A-Z), numbers (0-9), or underscores (_). The maximum combined length of the table name and the table suffix is 1024 characters.

Quota
Template tables are subject to similar streaming quota limitations as other tables. In addition, when you stream to template tables, disabling best effort de-duplication does not provide higher quotas.

Time to live
The generated table inherits its expiration time from the dataset. As with normal streaming data, generated tables cannot be copied immediately.

Deduplication
Deduplication only happens between uniform references to a destination table. For example, if you simultaneously stream to a generated table using both template tables and a regular `insertAll` command, no deduplication occurs between rows inserted by template tables and a regular `insertAll` command.

Views
The template table and the generated tables should not be views.

Example use cases

High volume event logging

If you have an app that collects a large amount of data in real-time, streaming inserts can be a good choice. Generally, these types of apps have the following criteria:

- **Not transactional.** High volume, continuously appended rows. The app can tolerate a rare possibility that duplication might occur or that data might be temporarily unavailable.
- **Aggregate analysis.** Queries generally are performed for trend analysis, as opposed to single or narrow record selection.

One example of high volume event logging is event tracking. Suppose you have a mobile app that tracks events. Your app, or mobile servers, could independently record user interactions or system errors and stream them into BigQuery. You could analyze this data to determine overall trends, such as areas of high interaction or problems, and monitor error conditions in real-time.

Manually removing duplicates

You can use the following manual process to ensure that no duplicate rows exist after you are done streaming.

1. Add the `insertId` as a column in your table schema and include the `insertId` value in the data for each row.
2. After streaming has stopped, perform the following query to check for duplicates:

   ```
   #standardSQL
   SELECT
     MAX(count) FROM(
     SELECT
       ID_COLUMN,
       count(*) as count
     FROM
       `TABLE_NAME`
     GROUP BY
       ID_COLUMN)
   ```

   If the result is greater than 1, duplicates exist.
3. To remove duplicates, perform the following query. You should specify a destination table, allow large results, and disable result flattening.

   ```
   #standardSQL
   SELECT
     * EXCEPT(row_number)
   FROM (
     SELECT
       *,
       ROW_NUMBER()
             OVER (PARTITION BY ID_COLUMN) row_number
     FROM
       `TABLE_NAME`)
   WHERE
     row_number = 1
   ```

Notes about the duplicate removal query:

- The safer strategy for the duplicate removal query is to target a new table. Alternatively, you can target the source table with write disposition `WRITE_TRUNCATE`.
- The duplicate removal query adds a `row_number` column with the value `1` to the end of the table schema. The query uses a SELECT * EXCEPT statement from standard SQL to exclude the `row_number` column from the destination table. The `#standardSQL` prefix enables standard SQL for this query. Alternatively, you can select by specific column names to omit this column.

- For querying live data with duplicates removed, you can also create a view over your table using the duplicate removal query. Be aware that query costs against the view will be calculated based on the columns selected in your view, which can result in large bytes scanned sizes.

Troubleshooting streaming inserts

For information about how to troubleshoot errors during streaming inserts, see Troubleshooting streaming inserts on the Troubleshooting Errors page.

Back to top

Streaming insert examples

Before trying this sample, follow the Python setup instructions in the BigQuery quickstart using client libraries. For more information, see the BigQuery Python API reference documentation.

View on GitHub Feedback

```
from google.cloud import

 bigquery


# Construct a BigQuery client object.
client = bigquery.Client()
# TODO(developer): Set table_id to the ID of table to append to.
# table_id = "your-project.your_dataset.your_table"rows_to_insert = [
    {u"full_name": u"Phred Phlyntstone", u"age": 32},
    {u"full_name": u"Wylma Phlyntstone", u"age": 29},
]errors = client.insert_rows_json(table_id, rows_to_insert)  # Make an API
request.
if errors == []:
    print("New rows have been added.")
else:
    print("Encountered errors while inserting rows: {}".format(errors))
```

You do not need to populate the `insertID` field when you insert rows. The following example shows how to avoid sending an `insertID` for each row when streaming.

Python
Before trying this sample, follow the Python setup instructions in the BigQuery quickstart using client libraries. For more information, see the BigQuery Python API reference documentation.

View on GitHub Feedback

```
from google.cloud import

 bigquery


# Construct a BigQuery client object.
client = bigquery.Client()
# TODO(developer): Set table_id to the ID of table to append to.
# table_id = "your-project.your_dataset.your_table"rows_to_insert = [
    {u"full_name": u"Phred Phlyntstone", u"age": 32},
    {u"full_name": u"Wylma Phlyntstone", u"age": 29},
]errors = client.insert_rows_json(
    table_id, rows_to_insert, row_ids=[None] * len(rows_to_insert)
)  # Make an API request.
if errors == []:
    print("New rows have been added.")
else:
    print("Encountered errors while inserting rows: {}".format(errors))
```

Back to top

Rate and review