Running interactive and batch query jobs

cloud.google.com/bigquery/docs/running-queries

This document describes how to run *interactive* (on-demand) and *batch* query jobs.

Query results are saved to either a temporary or permanent table. You can choose whether to append or overwrite data in an existing table or whether to create a new table if none exists with the same name.

Required permissions

At a minimum, to run a query job, you must be granted `bigquery.jobs.create` permissions. In order for the query job to complete successfully, you must also be granted access to the tables or views referenced by the query. Access to the tables or views can be granted at the following levels, listed in order of range of resources allowed (largest to smallest):

- at a high level in the Google Cloud resource hierarchy such as the project, folder, or organization level
- at the dataset level
- at the table level

The following predefined IAM roles include `bigquery.jobs.create` permissions:

- `bigquery.user`
- `bigquery.jobUser`
- `bigquery.admin`

Note: Most users such as data analysts and data scientists should be granted the `bigquery.user` or `bigquery.jobUser` role to run query jobs. `bigquery.admin` is granted all BigQuery permissions. Alternatively, you can create a custom role that is granted all required permissions.
In addition, if a user has `bigquery.datasets.create` permissions, when that user creates a dataset, they are granted `bigquery.dataOwner` access to it. `bigquery.dataOwner` access gives the user the ability to query tables and views in the dataset.

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

Running interactive queries

By default, BigQuery runs interactive query jobs, which means that the query is executed as soon as possible. Interactive queries count toward your concurrent rate limit and your daily limit.

To run an interactive query that writes to a temporary table, follow these steps:

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()query = """
    SELECT name, SUM(number) as total_people
    FROM `bigquery-public-data.usa_names.usa_1910_2013`
    WHERE state = 'TX'
    GROUP BY name, state
    ORDER BY total_people DESC
    LIMIT 20
"""
query_job = client.query(query)  # Make an API request.
print("The query data:")
for row in query_job:
    # Row values can be accessed by field name or index.
    print("name={}, count={}".format(row[0], row["total_people"]))
```

Running batch queries

BigQuery also offers batch queries. BigQuery queues each batch query on your behalf, and starts the query as soon as idle resources are available in the BigQuery shared resource pool. This usually occurs within a few minutes. If BigQuery hasn't started the query within 24 hours, BigQuery changes the job priority to interactive.

Batch queries don't count towards your concurrent rate limit, which can make it easier to start many queries at once. Batch queries use the same resources as interactive (on-demand) queries. If you use flat-rate pricing batch queries and interactive queries share your allocated slots.

To run a batch query, follow these steps:

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()job_config = bigquery.QueryJobConfig(
    # Run at batch priority, which won't count toward concurrent rate limit.
    priority=bigquery.QueryPriority.BATCH
)sql = """
    SELECT corpus
    FROM `bigquery-public-data.samples.shakespeare`
    GROUP BY corpus;
"""
# Start the query, passing in the extra configuration.
query_job = client.query(sql, job_config=job_config)  # Make an API request.
# Check on the progress by getting the job's updated state. Once the state
# is `DONE`, the results are ready.
query_job = client.get_job(
    query_job.job_id, location=query_job.location
)  # Make an API request.
print("Job {} is currently in state {}".format(query_job.job_id, query_job.state))
```

Was this helpful?