

---

# Cloud Spanner - Defining Schemas and Understanding Query Plans

---

**Objective:** To gain hands-on experience in defining schemas in Google Cloud Spanner and analyzing query plans for query optimization in a managed relational database environment.

**Introduction:** Google Cloud Spanner is a fully managed, horizontally scalable, relational database service that offers global transactions and strong consistency. Defining efficient schemas and understanding query plans are essential for maintaining performance and scalability in your Cloud Spanner instances.

## Task 1. Load data into tables

---

The **banking-ops-db** was created with empty tables. Follow the steps below to load data into three of the tables (**Portfolio**, **Category**, and **Product**).

1. From the Cloud Console, open the navigation menu (☰) > **View All Products**, under **Databases** click **Spanner**.
2. The instance name is **banking-ops-instance**, click on the name to explore the databases.
3. The associated database is named **banking-ops-db**. Click on the name, scroll down to **Tables**, and you will see there are four tables already in place.
4. On the left pane of the Console, click **Spanner Studio**. Then click the **+ New SQL Editor Tab** button in the right frame

Filter Enter property name or value										
<input type="checkbox"/>	Name ↑	ID	Edition	Configuration	Processing units ?	Nodes ?	Scaling mode	Storage utilization ?	Labels ?	Tags ?
<input type="checkbox"/>	<a href="#">banking-ops-instance</a>	banking-ops-instance	Standard	us-central1 (Iowa)	1,000	1	Manual allocation	<div></div> 0 B / 10 TB		—

## Databases

[+ CREATE DATABASE](#)[REFRESH](#)

<div><div>Filter</div><div>Filter databases</div><div>?</div><div>III</div></div>					
<input type="checkbox"/>	Name <span>↑</span>	Dialect <span>?</span>	CPU utilization	Size	Backup
<input type="checkbox"/>	<a href="#">banking-ops-db</a>	Google Standard SQL	1.38%	0 B	

[+ CREATE TABLE](#)

<div><div>Filter</div><div>Filter tables</div><div>?</div></div>					
Name <span>↑</span>	Schema	Indexes	Interleaved in	Watched by	
<a href="#">Campaigns</a>	Default	—	—		
<a href="#">Category</a>	Default	1	Portfolio		
<a href="#">Portfolio</a>	Default	—	—		
<a href="#">Product</a>	Default	—	Category		

1. This takes you to the **Query** page. Paste the insert statements below as a single block to load the **Portfolio** table. Spanner will execute each in succession. Click **Run**:

```
insert into Portfolio (PortfolioId, Name, ShortName, PortfolioInfo) values (1, "Banking", "Bnkg", "All Banking Business");
```

```
insert into Portfolio (PortfolioId, Name, ShortName, PortfolioInfo) values (2, "Asset Growth", "AsstGrwth", "All Asset Focused Products");
```

```
insert into Portfolio (PortfolioId, Name, ShortName, PortfolioInfo) values (3, "Insurance", "Ins", "All Insurance Focused Products");
```

The screenshot shows a BigQuery console interface. At the top, there's a header with a home icon, the text "Untitled query", a plus icon, a "GEMINI" dropdown, and a "VIEW IN BIGQUERY" link. Below the header is a toolbar with buttons: "RUN" (with a play icon), "SAVE", "FORMAT", "CLEAR", and "DOCUMENTATION" (with an external link icon). A green checkmark and the word "Valid" are on the right. The main area contains three SQL insert statements for a table named "Portfolio". The first statement is highlighted with a red squiggly line under "insert". Below the code, a red error banner states: "Statement failed: Row [1] in table Portfolio already exists". At the bottom, there are tabs for "RESULTS" and "EXPLANATION" (with a question mark icon), and icons for download, full screen, and a dropdown arrow.

```

1 insert into Portfolio (PortfolioId, Name, ShortName, PortfolioInfo) values (1, "Banking", "Bnkg", "All Banking Business");
2 insert into Portfolio (PortfolioId, Name, ShortName, PortfolioInfo) values (2, "Asset Growth", "AsstGrwth", "All Asset
  Focused Products");
3 insert into Portfolio (PortfolioId, Name, ShortName, PortfolioInfo) values (3, "Insurance", "Ins", "All Insurance Focused
  Products");

```

Statement failed: Row [1] in table Portfolio already exists

RESULTS      EXPLANATION ?

6. The lower page of the screen shows the results of inserting the data one row at a time. A green checkmark also appears on each row of inserted data. The **Portfolio** table now has three rows.
7. Click **Clear** in the top portion of the page.
8. Paste the insert statements below as a single block to load the **Category** table. Click **Run**:
 

```

insert into Category (CategoryId,PortfolioId,CategoryName) values (1,1,"Cash");

insert into Category (CategoryId,PortfolioId,CategoryName) values (2,2,"Investments - Short Return");

insert into Category (CategoryId,PortfolioId,CategoryName) values (3,2,"Annuities");

insert into Category (CategoryId,PortfolioId,CategoryName) values (4,3,"Life Insurance");

```

The screenshot shows a BigQuery console interface, similar to the one above. The header and toolbar are the same. The main area contains four SQL insert statements for a table named "Category". The first statement is highlighted with a red squiggly line under "insert". Below the code, a red error banner states: "Statement failed: Row [1,1] in table Category already exists". At the bottom, there are tabs for "RESULTS" and "EXPLANATION" (with a question mark icon), and icons for download, full screen, and a dropdown arrow.

```

1 insert into Category (CategoryId,PortfolioId,CategoryName) values (1,1,"Cash");
2 insert into Category (CategoryId,PortfolioId,CategoryName) values (2,2,"Investments - Short Return");
3 insert into Category (CategoryId,PortfolioId,CategoryName) values (3,2,"Annuities");
4 insert into Category (CategoryId,PortfolioId,CategoryName) values (4,3,"Life Insurance");

```

Statement failed: Row [1,1] in table Category already exists

RESULTS      EXPLANATION ?

9. The lower page of the screen shows the results of inserting the data one row at a time. A green checkmark also appears on each row of inserted data. The **Category** table now has four rows.

10. Click **Clear** in the top portion of the page.

11. Paste the insert statements below as a single block to load the **Product** table. Click **Run**:

insert into Product

(ProductId,CategoryId,PortfolioId,ProductName,ProductAssetCode,ProductClass) values  
(1,1,1,"Checking Account","ChkAcct","Banking LOB");

insert into Product

(ProductId,CategoryId,PortfolioId,ProductName,ProductAssetCode,ProductClass) values  
(2,2,2,"Mutual Fund Consumer Goods","MFundCG","Investment LOB");

insert into Product

(ProductId,CategoryId,PortfolioId,ProductName,ProductAssetCode,ProductClass) values  
(3,3,2,"Annuity Early Retirement","AnnuFixed","Investment LOB");

insert into Product

(ProductId,CategoryId,PortfolioId,ProductName,ProductAssetCode,ProductClass) values  
(4,4,3,"Term Life Insurance","TermLife","Insurance LOB");

insert into Product

(ProductId,CategoryId,PortfolioId,ProductName,ProductAssetCode,ProductClass) values  
(5,1,1,"Savings Account","SavAcct","Banking LOB");

insert into Product

(ProductId,CategoryId,PortfolioId,ProductName,ProductAssetCode,ProductClass) values  
(6,1,1,"Personal Loan","PersLn","Banking LOB");

insert into Product

(ProductId,CategoryId,PortfolioId,ProductName,ProductAssetCode,ProductClass) values  
(7,1,1,"Auto Loan","AutLn","Banking LOB");

insert into Product

(ProductId,CategoryId,PortfolioId,ProductName,ProductAssetCode,ProductClass) values  
(8,4,3,"Permanent Life Insurance","PermLife","Insurance LOB");

insert into Product

(ProductId,CategoryId,PortfolioId,ProductName,ProductAssetCode,ProductClass) values  
(9,2,2,"US Savings Bonds","USSavBond","Investment LOB");

## Task 2. Use pre-built Python client library code to load data

---

You will be using the client libraries written in Python for the next several steps.

1. Open the **Cloud Shell** and paste the commands below to create and change into a new directory to hold the required files.

```
mkdir python-helper
```

```
cd python-helper
```

2. Next download two files. One is used to setup the environment. The other is the lab code.

```
wget https://storage.googleapis.com/cloud-training/OCBL373/requirements.txt
```

```
wget https://storage.googleapis.com/cloud-training/OCBL373/snippets.py
```

```
Your Cloud Platform project in this session is set to qwiklabs-gcp-02-6df8f17f6cc9.
Use 'gcloud config set project [PROJECT_ID]' to change to a different project.
student_03_3559808396a9@cloudshell:~ (qwiklabs-gcp-02-6df8f17f6cc9)$ mkdir python-helper
cd python-helper
mkdir: cannot create directory 'python-helper': File exists
student_03_3559808396a9@cloudshell:~/python-helper (qwiklabs-gcp-02-6df8f17f6cc9)$ wget https://storage.googleapis.com/cloud-training/OCBL373/requirements.txt
wget https://storage.googleapis.com/cloud-training/OCBL373/snippets.py
--2025-06-27 09:41:09-- https://storage.googleapis.com/cloud-training/OCBL373/requirements.txt
Resolving storage.googleapis.com (storage.googleapis.com)... 64.233.170.207, 74.125.200.207, 74.125.130.207, ...
Connecting to storage.googleapis.com (storage.googleapis.com)|64.233.170.207|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 66 [text/plain]
Saving to: 'requirements.txt.1'

requirements.txt.1      100%[=====>] 66 --.-KB/s  in 0s

2025-06-27 09:41:11 (21.8 MB/s) - 'requirements.txt.1' saved [66/66]

--2025-06-27 09:41:11-- https://storage.googleapis.com/cloud-training/OCBL373/snippets.py
Resolving storage.googleapis.com (storage.googleapis.com)... 64.233.170.207, 74.125.200.207, 74.125.130.207, ...
Connecting to storage.googleapis.com (storage.googleapis.com)|64.233.170.207|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 81532 (80K) [text/x-python-script]
Saving to: 'snippets.py.1'

snippets.py.1          100%[=====>] 79.62K 351KB/s  in 0.2s

2025-06-27 09:41:12 (351 KB/s) - 'snippets.py.1' saved [81532/81532]

student_03_3559808396a9@cloudshell:~/python-helper (qwiklabs-gcp-02-6df8f17f6cc9)$
```

3. Create an isolated Python environment and install dependencies for the Cloud Spanner client.

```
pip install -r requirements.txt
```

```
pip install setuptools
```

The **snippets.py** is a consolidated file with multiple Cloud Spanner DDL, DML, and DCL functions that you are going to use as a helper during this lab. Execute **snippets.py** using the **insert\_data** argument to populate the **Campaigns** table

```
python snippets.py banking-ops-instance --database-id banking-ops-db insert_data
```

```

student_03_3559808396a9@cloudshell:~/python-helper (qwklabs-gcp-02-6df8f17f6cc9)$ python snippets.py banking-ops-instance --database-id banking-ops-c
Traceback (most recent call last):
  File "/home/student_03_3559808396a9/python-helper/snippets.py", line 2204, in <module>
    insert_data(args.instance_id, args.database_id)
  File "/home/student_03_3559808396a9/python-helper/snippets.py", line 347, in insert_data
    with database.batch() as batch:
  File "/home/student_03_3559808396a9/.local/lib/python3.12/site-packages/google/cloud/spanner_v1/database.py", line 828, in __exit__
    self.batch.commit()
  File "/home/student_03_3559808396a9/.local/lib/python3.12/site-packages/google/cloud/spanner_v1/batch.py", line 182, in commit
    response = api.commit(
    ~~~~~
  File "/home/student_03_3559808396a9/.local/lib/python3.12/site-packages/google/cloud/spanner_v1/services/spanner/client.py", line 1704, in commit
    response = rpc(
    ~~~~~
  File "/usr/local/lib/python3.12/dist-packages/google/api_core/gapic_v1/method.py", line 131, in __call__
    return wrapped_func(*args, **kwargs)
    ~~~~~
  File "/usr/local/lib/python3.12/dist-packages/google/api_core/retry/retry_unary.py", line 293, in retry_wrapped_func
    return retry_target(
    ~~~~~
  File "/usr/local/lib/python3.12/dist-packages/google/api_core/retry/retry_unary.py", line 153, in retry_target
    _retry_error_helper(
  File "/usr/local/lib/python3.12/dist-packages/google/api_core/retry/retry_base.py", line 212, in _retry_error_helper
    raise final_exc from source_exc
  File "/usr/local/lib/python3.12/dist-packages/google/api_core/retry/retry_unary.py", line 144, in retry_target
    result = target()
    ~~~~~
  File "/usr/local/lib/python3.12/dist-packages/google/api_core/grpc_helpers.py", line 78, in error_remapped_callable
    raise exceptions.from_grpc_error(exc) from exc

```

```

student_03_3559808396a9@cloudshell:~/python-helper (qwklabs-gcp-02-6df8f17f6cc9)$ python snippets.py banking-ops-instance --database-id banking-ops-db query_data
CampaignId: 1, PortfolioId: 1, CampaignStartDate: 2025-06-27, CampaignEndDate: 2025-06-27, CampaignName: New Account Reward, CampaignBudget: 15000
CampaignId: 2, PortfolioId: 2, CampaignStartDate: 2025-06-27, CampaignEndDate: 2025-06-27, CampaignName: Intro to Investments, CampaignBudget: 5000
CampaignId: 3, PortfolioId: 2, CampaignStartDate: 2025-06-27, CampaignEndDate: 2025-06-27, CampaignName: Youth Checking Accounts, CampaignBudget: 25000
CampaignId: 4, PortfolioId: 3, CampaignStartDate: 2025-06-27, CampaignEndDate: 2025-06-27, CampaignName: Protect Your Family, CampaignBudget: 10000
student_03_3559808396a9@cloudshell:~/python-helper (qwklabs-gcp-02-6df8f17f6cc9)$

```

### Task 3. Query data with client libraries

The **query\_data()** function in **snippets.py** can be used to query your database. In this case you use it to confirm the data loaded into the **Campaigns** table. You will not change any code, the section is shown here for your reference.

1. Execute **snippets.py** using the **query\_data** argument to query the **Campaigns** table.

```
python snippets.py banking-ops-instance --database-id banking-ops-db query_data
```

```

student_03_3559808396a9@cloudshell:~/python-helper (qwklabs-gcp-02-6df8f17f6cc9)$ python snippets.py banking-ops-instance --database-id banking-ops-db query_data
CampaignId: 1, PortfolioId: 1, CampaignStartDate: 2025-06-27, CampaignEndDate: 2025-06-27, CampaignName: New Account Reward, CampaignBudget: 15000
CampaignId: 2, PortfolioId: 2, CampaignStartDate: 2025-06-27, CampaignEndDate: 2025-06-27, CampaignName: Intro to Investments, CampaignBudget: 5000
CampaignId: 3, PortfolioId: 2, CampaignStartDate: 2025-06-27, CampaignEndDate: 2025-06-27, CampaignName: Youth Checking Accounts, CampaignBudget: 25000
CampaignId: 4, PortfolioId: 3, CampaignStartDate: 2025-06-27, CampaignEndDate: 2025-06-27, CampaignName: Protect Your Family, CampaignBudget: 10000
student_03_3559808396a9@cloudshell:~/python-helper (qwklabs-gcp-02-6df8f17f6cc9)$

```

### Task 4. Updating the database schema

As part of your DBA responsibilities you are required to add a new column called **MarketingBudget** to the **Category** table. Adding a new column to an existing table requires an update to your database schema. Cloud Spanner supports schema updates to a database while the database continues to serve traffic. Schema updates do not require taking the database offline and they do not lock entire tables or columns; you can continue reading and writing data to the database during the schema update.

Adding a column using Python

The **update\_ddl()** method of the **Database** class is used to modify the schema.

Use the **add\_column()** function in **snippets.py** which implements that method. You will not change any code, the section is shown here for your reference.

1. Execute **snippets.py** using the **add\_column** argument.

```
python snippets.py banking-ops-instance --database-id banking-ops-db add_column
```

```
student_03_3559808396a9@cloudshell:~/python-helper (qwiklabs-gcp-02-6df8f17f6cc9)$ python snippets.py banking-ops-instance --database-id banking-ops-db add_column
Waiting for operation to complete...
Traceback (most recent call last):
  File "/home/student_03_3559808396a9/python-helper/snippets.py", line 2214, in <module>
    add_column(args.instance_id, args.database_id)
  File "/home/student_03_3559808396a9/python-helper/snippets.py", line 639, in add_column
    operation.result(OPTION_TIMEOUT_SECONDS)
  File "/usr/local/lib/python3.12/dist-packages/google/api_core/future/polling.py", line 261, in result
    raise self._exception
google.api_core.exceptions.FailedPrecondition: 400 Duplicate column name Category.MarketingBudget. 9: Duplicate column name Category.MarketingBudget.
student_03_3559808396a9@cloudshell:~/python-helper (qwiklabs-gcp-02-6df8f17f6cc9)$
```

Other options to add a column to an existing table include the following:

### Issuing a DDL command via the gcloud CLI.

1. Click the table name in the Database listing.
2. Click **Write DDL** in the top right corner of the page.
3. Paste the appropriate DDL in the **DDL Templates** box.
4. Click **Submit**.

#### ← Write DDL statements

Use Cloud Spanner's Data Definition Language (DDL) to define the schema in your instance. DDL statements let you alter a database; create, alter, or drop tables in a database; and create or drop indexes in a database. To write multiple statements, separate them with a semicolon. [Learn more](#)

Database dialect Google Standard SQL

DDL TEMPLATES ▾ SHORTCUTS

Press Alt+F1 for Accessibility Options.

- 1 ALTER TABLE Category
- 2 ADD COLUMN MarketingBudget INT64;

SUBMIT

CANCEL

1. Execute **snippets.py** using the **update\_data** argument.

```
python snippets.py banking-ops-instance --database-id banking-ops-db update_data
```

Query the table again to see the update. Execute **snippets.py** using the **query\_data\_with\_new\_column** argument

```
python snippets.py banking-ops-instance --database-id banking-ops-db
query_data_with_new_column
```

## Task 5. Add a Secondary Index

---

Suppose you wanted to fetch all rows of Categories that have CategoryNames values in a certain range. You could read all values from the CategoryName column using a SQL statement or a read call, and then discard the rows that don't meet the criteria, but doing this full table scan is expensive, especially for tables with a lot of rows. Instead you can speed up the retrieval of rows when searching by non-primary key columns by creating a secondary index on the table.

Adding a secondary index to an existing table requires a schema update. Like other schema updates, Cloud Spanner supports adding an index while the database continues to serve traffic. Cloud Spanner populates the index with data (also known as a "backfill") under the hood. Backfills might take several minutes to complete, but you don't have to take the database offline or avoid writing to certain tables or columns during this process.

Add a secondary index using the Python client library

Use the `add_index()` method to create a secondary index. You will not change any code, the section is shown here for your reference.

1. Execute **snippets.py** using the **add\_index** argument.

```
python snippets.py banking-ops-instance --database-id banking-ops-db add_index
```

## Task 6. Examine Query plans

---

In this section, you will explore Cloud Spanner **Query Plans**.

1. Return to the **Cloud Console**, it should still be on the **Query** tab of **Spanner Studio**. Clear any existing query, paste, and **Run** the following query:

```
SELECT Name, ShortName, CategoryName
FROM Portfolio
INNER JOIN Category
ON Portfolio.Portfolioid = Category.Portfolioid;
```



Google Cloud | qwiklabs-gcp-03-e9d1131fa43e | Search (/) for resources, docs, products, and more

All instances > INSTANCE banking-ops-instance: Overview > GOOGLE STANDARD SQL DATABASE banking-ops-db: Spanner Studio

Explorer

- Default
  - Tables 4
    - Campaigns
    - Category
    - Portfolio
    - Product
  - Change streams 0
  - Views 0
  - Roles 3
  - Models 0
- INFORMATION\_SCHEMA
- SPANNER\_SYS

Query 1

```

1 SELECT Name, ShortName, CategoryName
2 FROM Portfolio
3 INNER JOIN Category
4 ON Portfolio.PortfolioId = Category.PortfolioId;

```

RUN CLEAR QUERY FORMAT QUERY SHORTCUTS SQL query help

RESULTS EXPLANATION

All results > SELECT Name, ShortName, CategoryName FROM Portfolio INNER JOIN Category ON

Name	ShortName	CategoryName
Banking	Bnkg	Cash
Asset Growth	AsstGrwth	Investments - Short Return
Asset Growth	AsstGrwth	Annuities
Insurance	Ins	Life Insurance

## Life of a query

A SQL query in Cloud Spanner is first compiled into an execution plan, then it is sent to an initial root server for execution. The root server is chosen so as to minimize the number of hops to reach the data being queried. The root server then:

- Initiates remote execution of subplans (if necessary)
- Waits for results from the remote executions
- Handles any remaining local execution steps such as aggregating results
- Returns results for the query

Remote servers that receive a subplan act as the "root" server for their subplan, following the same model as the top-most root server. The result is a tree of remote executions.

Conceptually, query execution flows from top to bottom, and query results are returned from bottom to top. The following diagram shows this pattern

