


Cloud Spanner - Defining Schemas and Understanding Query Plans

Activate Cloud Shell

Cloud Shell is a virtual machine that is loaded with development tools. It offers a persistent 5GB home directory and runs on the Google Cloud. Cloud Shell provides command-line access to your Google Cloud resources.

1. Click **Activate Cloud Shell**  at the top of the Google Cloud console.
2. Click through the following windows:
 - Continue through the Cloud Shell information window.
 - Authorize Cloud Shell to use your credentials to make Google Cloud API calls.

When you are connected, you are already authenticated, and the project is set to your **Project_ID**, `qwiklabs-gcp-03-55d52bde554a`. The output contains a line that declares the **Project_ID** for this session:

```
Your Cloud Platform project in this session is set to quiklabs-gcp-03-55d52bde554a
```

`gcloud` is the command-line tool for Google Cloud. It comes pre-installed on Cloud Shell and supports tab-completion.

3. (Optional) You can list the active account name with this command:

```
gcloud auth list
```

Copied!

```
content_copy
```

4. Click **Authorize**.

Output:

```
ACTIVE: *
ACCOUNT: student-00-cebe0817c14d@quiklabs.net

To set the active account, run:
$ gcloud config set account `ACCOUNT`
```

5. (Optional) You can list the project ID with this command:

```
gcloud config list project
```

Output:

```
[core]  
project = qwiklabs-gcp-03-55d52bde554a
```

Note: For full documentation of `gcloud`, in Google Cloud, refer to [the gcloud CLI overview guide](#).

Cloud Spanner instance

In order to allow you to move more quickly through this lab a Cloud Spanner instance, database, and tables were automatically created for you.

Here are some details for your reference:

Item	Name	Details
Cloud Spanner Instance	banking-ops-instance	This is the project-level instance
Cloud Spanner Database	banking-ops-db	This is the instance specific database
Table	Portfolio	Contains top-level bank offerings
Table	Category	Contains second-tier bank offering groupings
Table	Product	Contains specific line-item bank offerings
Table	Campaigns	Contains details on marketing initiatives

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**).

6. From the Cloud Console, open the navigation menu (☰) > **View All Products**, under **Databases** click **Spanner**.
7. The instance name is **banking-ops-instance**, click on the name to explore the databases.
8. 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.
9. On the left pane of the Console, click **Spanner Studio**. Then click the + **New SQL Editor Tab** button in the right frame.
10. 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 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.

6. Click **Clear** in the top portion of the page.
7. 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");
```

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,Pr
oductClass) values (1,1,1,"Checking Account","ChkAcct","Banking
LOB");
insert into Product
(ProductId,CategoryId,PortfolioId,ProductName,ProductAssetCode,Pr
oductClass) values (2,2,2,"Mutual Fund Consumer
Goods","MFundCG","Investment LOB");
insert into Product
(ProductId,CategoryId,PortfolioId,ProductName,ProductAssetCode,Pr
oductClass) values (3,3,2,"Annuity Early
Retirement","AnnuFixed","Investment LOB");
insert into Product
(ProductId,CategoryId,PortfolioId,ProductName,ProductAssetCode,Pr
oductClass) values (4,4,3,"Term Life
Insurance","TermLife","Insurance LOB");
insert into Product
(ProductId,CategoryId,PortfolioId,ProductName,ProductAssetCode,Pr
oductClass) values (5,1,1,"Savings Account","SavAcct","Banking
LOB");
insert into Product
(ProductId,CategoryId,PortfolioId,ProductName,ProductAssetCode,Pr
oductClass) values (6,1,1,"Personal Loan","PersLn","Banking
LOB");
insert into Product
(ProductId,CategoryId,PortfolioId,ProductName,ProductAssetCode,Pr
oductClass) values (7,1,1,"Auto Loan","AutLn","Banking LOB");
insert into Product
(ProductId,CategoryId,PortfolioId,ProductName,ProductAssetCode,Pr
oductClass) values (8,4,3,"Permanent Life
Insurance","PermLife","Insurance LOB");
insert into Product
(ProductId,CategoryId,PortfolioId,ProductName,ProductAssetCode,Pr
oductClass) values (9,2,2,"US Savings
Bonds","USSavBond","Investment LOB");
```

12. 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 **Product** table now has nine rows.

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
```

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

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

4. 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
```

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.

```
def query_data(instance_id, database_id):  
    """Queries sample data from the database using SQL."""  
    spanner_client = spanner.Client()  
    instance = spanner_client.instance(instance_id)  
    database = instance.database(database_id)  
  
    with database.snapshot() as snapshot:  
        results = snapshot.execute_sql(  

```

```

        "SELECT
CampaignId,PortfolioId,CampaignStartDate,CampaignEndDate,CampaignName,C
ampaignBudget FROM Campaigns"
    )

    for row in results:
        print(u"CampaignId: {}, PortfolioId: {}, CampaignStartDate:
{}, CampaignEndDate: {}, CampaignName: {}, CampaignBudget:
{}".format(*row))

```

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
```

The result should look like the following

```

CampaignId: 1, PortfolioId: 1, CampaignStartDate: 2022-06-07,
CampaignEndDate: 2022-06-07, CampaignName: New Account Reward,
CampaignBudget: 15000
CampaignId: 2, PortfolioId: 2, CampaignStartDate: 2022-06-07,
CampaignEndDate: 2022-06-07, CampaignName: Intro to Investments,
CampaignBudget: 5000
CampaignId: 3, PortfolioId: 2, CampaignStartDate: 2022-06-07,
CampaignEndDate: 2022-06-07, CampaignName: Youth Checking Accounts,
CampaignBudget: 25000
CampaignId: 4, PortfolioId: 3, CampaignStartDate: 2022-06-07,
CampaignEndDate: 2022-06-07, CampaignName: Protect Your Family,
CampaignBudget: 10000

```

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.

```
def add_column(instance_id, database_id):
    """Adds a new column to the Albums table in the example
    database."""
    spanner_client = spanner.Client()
    instance = spanner_client.instance(instance_id)
    database = instance.database(database_id)

    operation = database.update_ddl(
        ["ALTER TABLE Category ADD COLUMN MarketingBudget INT64"]
    )

    print("Waiting for operation to complete...")
    operation.result(OPERATION_TIMEOUT_SECONDS)

    print("Added the MarketingBudget column.")
```

1. Execute `snippets.py` using the `add_column` argument.

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

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

Issuing a DDL command via the gcloud CLI.

Note: This option is shown as an alternate example. **Do not issue this command.**

The code sample below completes the same task you just executed via Python.

```
gcloud spanner databases ddl update banking-ops-db --instance=banking-ops-instance --ddl='ALTER TABLE Category ADD COLUMN MarketingBudget INT64;'
```

Issuing a DDL command in the Cloud Console.

Note: This option is shown as an alternate example. **Do not perform this action.**

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

Write data to the new column

The following code writes data to the new column. It sets **MarketingBudget** to 100000 for the row with a **CategoryId** of 1 and a **PortfolioId** of 1 and to 500000 for the row with a **CategoryId** of 3 and a **PortfolioId** of 2. You will not change any code, the section is shown here for your reference.

```
def update_data(instance_id, database_id):
    """Updates sample data in the database.

    This updates the `MarketingBudget` column which must be created
    before
    running this sample. You can add the column by running the
    `add_column`
    sample or by running this DDL statement against your database

    """
    spanner_client = spanner.Client()
    instance = spanner_client.instance(instance_id)
    database = instance.database(database_id)

    with database.batch() as batch:
        batch.update(
            table="Category",
            columns=("CategoryId", "PortfolioId", "MarketingBudget"),
            values=[(1, 1, 100000), (3, 2, 500000)],
        )

    print("Updated data.")
```


1. Execute **snippets.py** using the **update_data** argument.

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

Copied!

content_copy

2. 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
```

The result should be:

```
CategoryId: 1, PortfolioId: 1, MarketingBudget: 100000  
CategoryId: 2, PortfolioId: 2, MarketingBudget: None  
CategoryId: 3, PortfolioId: 2, MarketingBudget: 500000  
CategoryId: 4, PortfolioId: 3, MarketingBudget: None
```

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.

```
def add_index(instance_id, database_id):
    """Adds a simple index to the example database."""
    spanner_client = spanner.Client()
    instance = spanner_client.instance(instance_id)
    database = instance.database(database_id)

    operation = database.update_ddl(
        ["CREATE INDEX CategoryByCategoryName ON
        Category(CategoryName)"]
    )

    print("Waiting for operation to complete...")
    operation.result(OPTION_TIMEOUT_SECONDS)

    print("Added the CategoryByCategoryName index.")
```

1. Execute **snippets.py** using the **add_index** argument.

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

Read using the index

To read using the index, invoke a variation of the **read()** method with an index included. You will not change any code, the section is shown here for your reference.

```
def read_data_with_index(instance_id, database_id):
    """Reads sample data from the database using an index.

    """
    spanner_client = spanner.Client()
    instance = spanner_client.instance(instance_id)
    database = instance.database(database_id)

    with database.snapshot() as snapshot:
        keyset = spanner.KeySet(all_=True)
        results = snapshot.read(
            table="Category",
            columns=("CategoryId", "CategoryName"),
            keyset=keyset,
            index="CategoryByCategoryName",
        )

        for row in results:
            print("CategoryId: {}, CategoryName: {}".format(*row))
```

1. Execute **snippets.py** using the **read_data_with_index** argument.

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

The result should look like this:

```
CategoryId: 3, CategoryName: Annuities
CategoryId: 1, CategoryName: Cash
CategoryId: 2, CategoryName: Investments - Short Return
CategoryId: 4, CategoryName: Life Insurance
```

Add an index with a **STORING** clause

You might have noticed that the read example above did not include reading the **MarketingBudget** column. This is because Cloud Spanner's read interface does not support the ability to join an index with a data table to look up values that are not stored in the index.

To bypass this restriction, create an alternate definition of the **CategoryByCategoryName** index that stores a copy of **MarketingBudget** in the index.

Use the **update_ddl()** method of the Database class to add an index with a **STORING** clause. You will not change any code, the section is shown here for your reference.

```
def add_storing_index(instance_id, database_id):
    """Adds an storing index to the example database."""
    spanner_client = spanner.Client()
    instance = spanner_client.instance(instance_id)
    database = instance.database(database_id)

    operation = database.update_ddl(
        [
            "CREATE INDEX CategoryByCategoryName2 ON
Category(CategoryName) "
            "STORING (MarketingBudget) "
        ]
    )

    print("Waiting for operation to complete...")
    operation.result(OPERATION_TIMEOUT_SECONDS)

    print("Added the CategoryByCategoryName2 index.")
```

1. Execute **snippets.py** using the **add_storing_index** argument.

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

Now you can execute a read that fetches the **CategoryId**, **CategoryName**, and **MarketingBudget** columns while using the **CategoryByCategoryName2** index. You will not change any code, the section is shown here for your reference.

```
def read_data_with_storing_index(instance_id, database_id):
    """Reads sample data from the database using an index with a
    storing
    clause.

    """
    spanner_client = spanner.Client()
    instance = spanner_client.instance(instance_id)
    database = instance.database(database_id)

    with database.snapshot() as snapshot:
        keyset = spanner.KeySet(all_=True)
        results = snapshot.read(
            table="Category",
            columns=("CategoryId", "CategoryName", "MarketingBudget"),
            keyset=keyset,
            index="CategoryByCategoryName2",
        )

        for row in results:
            print(u"CategoryId: {}, CategoryName: {}, "
                  "MarketingBudget: {}".format(*row))
```

2. Execute **snippets.py** using the **read_data_with_storing_index** argument.

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

The result should be

```
CategoryId: 3, CategoryName: Annuities, MarketingBudget: 500000
CategoryId: 1, CategoryName: Cash, MarketingBudget: 100000
CategoryId: 2, CategoryName: Investments - Short Return,
MarketingBudget: None
CategoryId: 4, CategoryName: Life Insurance, MarketingBudget: None
```

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;
```