Inserting, updating, and deleting data using mutations

This page describes how to insert, update, and delete data using mutations. Although you can commit mutations by using gRPC (https://cloud.google.com/spanner/docs/reference/rpc/google.spanner.v1#google.spanner.v1.CommitRequest) or REST (https://cloud.google.com/spanner/docs/reference/rest/v1/projects.instances.databases.sessions/commit), it is more common to access the APIs is through the client libraries.

This page shows the basic tasks of insert, update, and delete. You can find more examples in the Getting started tutorials (https://cloud.google.com/spanner/docs/tutorials).

Inserting new rows in a table

C# GO JAVA NODE.JS MORE

You can insert data using the `connection.CreateInsertCommand()` (https://googleapis.github.io/google-cloud-dotnet/docs/Google.Cloud.Spanner.Data/api/Google.Cloud.Spanner.Data.SpannerConnection.html#Google_Cloud_Spanner_Data_SpannerConnection_CreateInsertCommand_System_String_Google_Cloud_Spanner_Data_SpannerParameterCollection_) method, which creates a new `SpannerCommand` to insert rows into a table. The `SpannerCommand.ExecuteNonQueryAsync()` (https://googleapis.github.io/google-cloud-dotnet/docs/Google.Cloud.Spanner.Data/api/Google.Cloud.Spanner.Data.SpannerCommand.html#Google_Cloud_Spanner_Data_SpannerCommand_ExecuteNonQueryAsync_System_Threading_CancellationToken_) method adds new rows to the table.

Note: You can run multiple transactions in parallel using a single `SpannerConnection` object. When running additional transactions, you must ensure that the `SpannerConnection` object is in the `Open` state before you execute additional transaction commands by calling the `SpannerCommand.ExecuteNonQueryAsync()`

(https://googleapis.github.io/google-cloud-dotnet/docs/Google.Cloud.Spanner.Data/api/Google.Cloud.Spanner.Data.SpannerConnection.html#Google_Cloud_Spanner_Data_SpannerConnection_OpenAsync_System_Threading_CancellationToken_)
method, as seen in the following example.

This code shows how to insert data:

spanner/api/Spanner/Program.cs
(https://github.com/GoogleCloudPlatform/dotnet-docs-samples/blob/master/spanner/api/Spanner/Program.cs)

_ECLOUDPLATFORM/DOTNET-DOCS-SAMPLES/BLOB/MASTER/SPANNER/API/SPANNER/PROGRAM.CS)

```
public class Singer
{
    public int SingerId { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

public class Album
{
    public int SingerId { get; set; }
    public int AlbumId { get; set; }
    public string AlbumTitle { get; set; }
}
public static async Task InsertSampleDataAsync(
    string projectId, string instanceId, string databaseId)
{
    const int firstSingerId = 1;
    const int secondSingerId = 2;
    string connectionString =
    $"Data Source=projects/{projectId}/instances/{instanceId}"
    + $"/databases/{databaseId}";
    List<Singer> singers = new List<Singer> {
        new Singer {SingerId = firstSingerId, FirstName = "Marc",
            LastName = "Richards"},
        new Singer {SingerId = secondSingerId, FirstName = "Catalina",
            LastName = "Smith"},
        new Singer {SingerId = 3, FirstName = "Alice",
            LastName = "Trentor"},
        new Singer {SingerId = 4, FirstName = "Lea",
            LastName = "Martin"},
        new Singer {SingerId = 5, FirstName = "David",
            LastName = "Lomond"},
```

```
};
List<Album> albums = new List<Album> {
    new Album {SingerId = firstSingerId, AlbumId = 1,
        AlbumTitle = "Total Junk"},
    new Album {SingerId = firstSingerId, AlbumId = 2,
        AlbumTitle = "Go, Go, Go"},
    new Album {SingerId = secondSingerId, AlbumId = 1,
        AlbumTitle = "Green"},
    new Album {SingerId = secondSingerId, AlbumId = 2,
        AlbumTitle = "Forever Hold your Peace"},
    new Album {SingerId = secondSingerId, AlbumId = 3,
        AlbumTitle = "Terrified"},
};
// Create connection to Cloud Spanner.
using (var connection = new SpannerConnection(connectionString))
{
    await connection.OpenAsync();

    // Insert rows into the Singers table.
    var cmd = connection.CreateInsertCommand("Singers",
        new SpannerParameterCollection {
            {"SingerId", SpannerDbType.Int64},
            {"FirstName", SpannerDbType.String},
            {"LastName", SpannerDbType.String}
    });
    await Task.WhenAll(singers.Select(singer =>
    {
        cmd.Parameters["SingerId"].Value = singer.SingerId;
        cmd.Parameters["FirstName"].Value = singer.FirstName;
        cmd.Parameters["LastName"].Value = singer.LastName;
        return cmd.ExecuteNonQueryAsync();
    }));

    // Insert rows into the Albums table.
    cmd = connection.CreateInsertCommand("Albums",
        new SpannerParameterCollection {
            {"SingerId", SpannerDbType.Int64},
            {"AlbumId", SpannerDbType.Int64},
            {"AlbumTitle", SpannerDbType.String}
    });
    await Task.WhenAll(albums.Select(album =>
    {
        cmd.Parameters["SingerId"].Value = album.SingerId;
        cmd.Parameters["AlbumId"].Value = album.AlbumId;
        cmd.Parameters["AlbumTitle"].Value = album.AlbumTitle;
```

```
            return cmd.ExecuteNonQueryAsync();
        }));
        Console.WriteLine("Inserted data.");
    }
}
```

Updating rows in a table

Suppose that sales of `Albums(1, 1)` are lower than expected. As a result, you want to move $200,000 from the marketing budget of `Albums(2, 2)` to `Albums(1, 1)`, but only if the money is available in the budget of `Albums(2, 2)`.

Because you need to read the data in the tables to determine whether to write new values, you should use a read-write transaction (https://cloud.google.com/spanner/docs/transactions#read-write_transactions) to perform the reads and writes atomically.

C# GO JAVA NODE.JS MORE

For .NET Standard 2.0 (or .NET 4.5) and above, you can use the .NET framework's `TransactionScope()` (https://msdn.microsoft.com/en-us/library/system.transactions.transactionscope) to run a transaction. For all supported versions of .NET, you can create a transaction by setting the result of `SpannerConnection.BeginTransactionAsync` as the `Transaction` property of `SpannerCommand`.

Here are the two ways to run the transaction:

.NET STANDARD 2.0 .NET STANDARD 1.5

spanner/api/Spanner/Program.cs (https://github.com/GoogleCloudPlatform/dotnet-docs-samples/blob/master/spanner/api/Spanner/Program.cs)

ECLOUDPLATFORM/DOTNET-DOCS-SAMPLES/BLOB/MASTER/SPANNER/API/SPANNER/PROGRAM.CS)

```
public static async Task ReadWriteWithTransactionAsync(
    string projectId,
    string instanceId,
    string databaseId)
{
    // This sample transfers 200,000 from the MarketingBudget
```

```
// field of the second Album to the first Album. Make sure to run
// the addColumn and writeDataToNewColumn samples first,
// in that order.

string connectionString =
$"Data Source=projects/{projectId}/instances/{instanceId}"
+ $"/databases/{databaseId}";

using (TransactionScope scope = new TransactionScope(
    TransactionScopeAsyncFlowOption.Enabled))
{
    decimal transferAmount = 200000;
    decimal secondBudget = 0;
    decimal firstBudget = 0;

    // Create connection to Cloud Spanner.
    using (var connection =
        new SpannerConnection(connectionString))
    {
        // Create statement to select the second album's data.
        var cmdLookup = connection.CreateSelectCommand(
        "SELECT * FROM Albums WHERE SingerId = 2 AND AlbumId = 2");
        // Excecute the select query.
        using (var reader = await cmdLookup.ExecuteReaderAsync())
        {
            while (await reader.ReadAsync())
            {
                // Read the second album's budget.
                secondBudget =
                  reader.GetFieldValue<decimal>("MarketingBudget");
                // Confirm second Album's budget is sufficient and
                // if not raise an exception. Raising an exception
                // will automatically roll back the transaction.
                if (secondBudget < transferAmount)
                {
                    throw new Exception("The second album's "
                        + $"budget {secondBudget} "
                        + "is less than the "
                        + "amount to transfer.");
                }
            }
        }
        // Read the first album's budget.
        cmdLookup = connection.CreateSelectCommand(
        "SELECT * FROM Albums WHERE SingerId = 1 and AlbumId = 1");
```

```
            using (var reader = await cmdLookup.ExecuteReaderAsync())
            {
                while (await reader.ReadAsync())
                {
                    firstBudget =
                      reader.GetFieldValue<decimal>("MarketingBudget");
                }
            }

            // Specify update command parameters.
            var cmd = connection.CreateUpdateCommand("Albums",
                new SpannerParameterCollection {
                {"SingerId", SpannerDbType.Int64},
                {"AlbumId", SpannerDbType.Int64},
                {"MarketingBudget", SpannerDbType.Int64},
            });
            // Update second album to remove the transfer amount.
            secondBudget -= transferAmount;
            cmd.Parameters["SingerId"].Value = 2;
            cmd.Parameters["AlbumId"].Value = 2;
            cmd.Parameters["MarketingBudget"].Value = secondBudget;
            await cmd.ExecuteNonQueryAsync();
            // Update first album to add the transfer amount.
            firstBudget += transferAmount;
            cmd.Parameters["SingerId"].Value = 1;
            cmd.Parameters["AlbumId"].Value = 1;
            cmd.Parameters["MarketingBudget"].Value = firstBudget;
            await cmd.ExecuteNonQueryAsync();
            scope.Complete();
            Console.WriteLine("Transaction complete.");
        }
    }
}
```

Deleting rows in a table

Each client library provides multiple ways to delete rows:

- Delete all the rows in a table.
- Delete a single row by specifying the key column values for the row.

- Delete a group of rows by creating a key range.
- Delete rows in an interleaved table by deleting the parent rows, if the interleaved table includes `ON DELETE CASCADE` in its schema definition.

Note: If you want to delete a large amount of data, you should use Partitioned DML (https://cloud.google.com/spanner/docs/dml-partitioned), because Partitioned DML handles transaction limits and is optimized to handle large-scale deletions. For example, you usually can't delete all of the rows in a large table in a single transaction if the table has a secondary index defined on it. The secondary index causes the transaction to exceed the transaction size limits.

C# GO JAVA NODE.JS MORE

Delete rows using the `connection.CreateDeleteCommand()` (https://googleapis.github.io/google-cloud-dotnet/docs/Google.Cloud.Spanner.Data/api/Google.Cloud.Spanner.Data.SpannerConnection.html#Google_Cloud_Spanner_Data_SpannerConnection_CreateDeleteCommand_System_String_Google_Cloud_Spanner_Data_SpannerParameterCollection_) method, which creates a new `SpannerCommand` to delete rows. The `SpannerCommand.ExecuteNonQueryAsync()` (https://googleapis.github.io/google-cloud-dotnet/docs/Google.Cloud.Spanner.Data/api/Google.Cloud.Spanner.Data.SpannerCommand.html#Google_Cloud_Spanner_Data_SpannerCommand_ExecuteNonQueryAsync_System_Threading_CancellationToken_) method deletes the rows from the table.

This example deletes the rows in the `Singers` table individually. The rows in the `Albums` table are deleted because the `Albums` table is interleaved in the `Singers` table and is defined with `ON DELETE CASCADE`.

spanner/api/Spanner/Program.cs (https://github.com/GoogleCloudPlatform/dotnet-docs-samples/blob/master/spanner/api/Spanner/Program.cs)

_ECLOUDPLATFORM/DOTNET-DOCS-SAMPLES/BLOB/MASTER/SPANNER/API/SPANNER/PROGRAM.CS)

```
public static async Task DeleteSampleDataAsync(
    string projectId, string instanceId, string databaseId)
{
    const int firstSingerId = 1;
    const int secondSingerId = 2;
    string connectionString =
        $"Data Source=projects/{projectId}/instances/{instanceId}"
        + $"/databases/{databaseId}";
```

```
    List<Singer> singers = new List<Singer> {
        new Singer {SingerId = firstSingerId, FirstName = "Marc",
            LastName = "Richards"},
        new Singer {SingerId = secondSingerId, FirstName = "Catalina",
            LastName = "Smith"},
        new Singer {SingerId = 3, FirstName = "Alice",
            LastName = "Trentor"},
        new Singer {SingerId = 4, FirstName = "Lea",
            LastName = "Martin"},
        new Singer {SingerId = 5, FirstName = "David",
            LastName = "Lomond"},
    };
    // Create connection to Cloud Spanner.
    using (var connection = new SpannerConnection(connectionString))
    {
        await connection.OpenAsync();

        // Delete rows from the Singers table.
        var cmd = connection.CreateDeleteCommand("Singers",
            new SpannerParameterCollection {
                {"SingerId", SpannerDbType.Int64}
            });
        await Task.WhenAll(singers.Select(singer =>
        {
            cmd.Parameters["SingerId"].Value = singer.SingerId;
            return cmd.ExecuteNonQueryAsync();
        }));

        Console.WriteLine("Deleted data.");
    }
}
```


Last updated January 24, 2020.