This page provides guidance on migrating a PostgreSQL database to Cloud Spanner. It describes several aspects of a PostgreSQL to Cloud Spanner migration:

- Mapping a PostgreSQL schema to a Cloud Spanner schema.
- Creating a Cloud Spanner instance, database, and schema.
- Refactoring the application to work with your Cloud Spanner database.
- Migrating your data.
- Verifying the new system and moving it to production status.

This page also provides some example schemas using tables from the MusicBrainz (https://musicbrainz.org/doc/MusicBrainz_Database) PostgreSQL database.

Your first step in moving a database from PostgreSQL to Cloud Spanner is to determine what schema changes you must make. Use `pg_dump` (https://www.postgresql.org/docs/current/static/app-pgdump.html) to create Data Definition Language (DDL) statements that define the objects in your PostgreSQL database, and then modify the statements as described in the following sections. After you update the DLL statements, use the DDL statements to create your database in a Cloud Spanner instance.

The following table describes how PostgreSQL data types (https://www.postgresql.org/docs/current/static/datatype.html) map to Cloud Spanner data types. Update the data types in your DDL statements from PostgreSQL data types to Cloud Spanner data types.

PostgreSQL	Cloud Spanner
`Bigint` `int8`	`INT64`

PostgreSQL	Cloud Spanner
`Bigserial` `serial8` ★	`INT64` **Note:** There is no auto-increment capability in Cloud Spanner.
`bit [ (n) ]`	`ARRAY<BOOL>`
`bit varying [ (n) ]` `varbit [ (n) ]`	`ARRAY<BOOL>`
`Boolean` `bool`	`BOOL`
`box`	`ARRAY<FLOAT64>`
`bytea`	`BYTES`
`character [ (n) ]` `char [ (n) ]`	`STRING`
`character varying [ (n) ]` `varchar [ (n) ]`	`STRING`
`cidr`	`STRING`, using standard CIDR (https://en.wikipedia.org/wiki/Classless_Inter-Domain_Routing) notation.
`circle`	`ARRAY<FLOAT64>`
`date`	`DATE`
`double precision` `float8`	`FLOAT64`
`inet`	`STRING`

`Integer`
`int`
`int4`
	`INT64`

PostgreSQL	Cloud Spanner
`interval[ fields ] [ (p) ]`	`INT64` if storing the value in milliseconds, or `STRING` if storing the value in an application-defined interval format.
`json`	`STRING`
`jsonb`	`BYTES`
`line`	`ARRAY<FLOAT64>`
`lseg`	`ARRAY<FLOAT64>`
`macaddr`	`STRING`, using standard MAC address (https://en.wikipedia.org/wiki/MAC_address) notation.
`money`	`INT64`, or `STRING` for arbitrary precision numbers (/spanner/docs/storing-numeric-data).
`numeric [ (p, s) ]` `decimal [ (p, s) ]`	`INT64`, or `STRING` for arbitrary precision numbers (/spanner/docs/storing-numeric-data).
`path`	`ARRAY<FLOAT64>`
`pg_lsn`	This data type is PostgreSQL-specific, so there isn't a Cloud Spanner equivalent.
`point`	`ARRAY<FLOAT64>`
`polygon`	`ARRAY<FLOAT64>`
`Real` `float4`	`FLOAT64`
`Smallint` `int2`	`INT64`
`Smallserial` `serial2`	`INT64`
`Serial` `serial4`	`INT64`
`text`	`STRING`
`time [ (p) ] [ without time zone ]`	`STRING`, using `HH:MM:SS.sss` notation.

PostgreSQL	Cloud Spanner
	`STRING`, using `HH:MM:SS.sss+ZZZZ` notation. Alternately, this can be broken up into two columns, one of type `TIMESTAMP` and another one holding the timezone.

`time [ (p) ] with time zone`
 	
`timetz`	

`timestamp [ (p) ] [ without time zone ]`	No equivalent. You may store as a `STRING` or `TIMESTAMP` at your discretion.

`timestamp [ (p) ] with time zone`
 	
`timestamptz`
	`TIMESTAMP`

`tsquery`	No equivalent. Define a storage mechanism in your application instead.
`tsvector`	No equivalent. Define a storage mechanism in your application instead.
`txid_snapshot`	No equivalent. Define a storage mechanism in your application instead.
`uuid`	`STRING` or `BYTES`
`xml`	`STRING`

For tables in your Cloud Spanner database that you frequently append to, avoid using primary keys that monotonically increase or decrease, as this approach causes hotspots during writes. Instead, modify the DDL `CREATE TABLE` statements so that they use supported primary key strategies (/spanner/docs/schema-design#choosing_a_primary_key_to_prevent_hotspots). Careful schema design is important, because you can't add or remove a primary key column after you create a table.

During migration, you might need to keep some existing monotonically increasing integer keys. If you need to keep these kinds of keys on a frequently updated table with a lot of operations on these keys, you can avoid creating hotspots by prefixing the existing key with a pseudo-random number. This technique causes Cloud Spanner to redistribute the rows. See What DBAs need to know about Cloud Spanner, part 1: Keys and indexes (https://cloudplatform.googleblog.com/2018/06/What-DBAs-need-to-know-about-Cloud-Spanner-part-1-Keys-and-indexes.html) for more information on using this approach.

Cloud Spanner doesn't have foreign key constraints or triggers. If you rely on these features, you must move this functionality to your application.

When there is a parent-child relationship between tables, and you want the records in those tables co-located for faster access, you can create interleaved tables (/spanner/docs/schema-and-data-model#creating-interleaved-tables). When you use an interleaved table, you can choose to enforce referential integrity to delete rows in the child table when the related row in the parent table is deleted. You can't delete the parent row if there are child rows and you used the `ON DELETE NO ACTION` (/spanner/docs/data-definition-language#create_table) clause. You also can't add a child row if the parent row doesn't exist.

To find the foreign keys on your PostgreSQL tables, query the `information_schema.table_constraints` (https://www.postgresql.org/docs/current/static/infoschema-table-constraints.html) view using a `WHERE constraint_type = 'FOREIGN KEY'` clause.

Update the `CREATE TABLE` statements so that they create interleaved tables as appropriate.

PostgreSQL b-tree indexes (https://www.postgresql.org/docs/10/static/indexes-types.html) are similar to secondary indexes (/spanner/docs/secondary-indexes) in Cloud Spanner. In a Cloud Spanner database you use secondary indexes to index commonly searched columns for better performance, and to replace any unique constraints specified in your tables. For example, if your PostgreSQL DDL has this statement:

You would use this statement in your Cloud Spanner DDL:

You can find the indexes for any of your PostgreSQL tables by running the `\di` (https://www.postgresql.org/docs/10/static/app-psql.html) meta-command in `psql`.

After you determine the indexes that you need, add `CREATE INDEX` (/spanner/docs/data-definition-language#create_index) statements to create them. Follow the guidance at Creating indexes (/spanner/docs/schema-design#creating-indexes).

Cloud Spanner implements indexes as tables, so indexing monotonically increasing columns (like those containing `TIMESTAMP` data) can cause a hotspot. See What DBAs need to know about Cloud Spanner, part 1: Keys and indexes (https://cloudplatform.googleblog.com/2018/06/What-DBAs-need-to-know-about-Cloud-Spanner-part-1-Keys-and-indexes.html) for more information on methods to avoid hotspots.

You must create the functionality of the following objects in your application logic:

- Views
- Triggers
- Stored procedures
- User-defined functions (UDFs)
- Columns that use `serial` data types as sequence generators

Keep the following tips in mind when migrating this functionality into application logic:

- You must migrate any SQL statements you use from the PostgreSQL SQL dialect to the Cloud Spanner SQL dialect (/spanner/docs/query-syntax).
- If you use cursors (https://www.postgresql.org/docs/current/static/plpgsql-cursors.html), you can rework the query to use offsets and limits (/spanner/docs/query-syntax#limit-clause-and-offset-clause).

After you update your DDL statements to conform to Cloud Spanner schema requirements, use it to create your database in Cloud Spanner.

1. Create a Cloud Spanner instance (/spanner/docs/create-manage-instances#creating_an_instance). Follow the guidance in Instances (/spanner/docs/instances) to determine the correct regional configuration and number of nodes to support your performance goals.
2. Create the database by using either the Google Cloud Console or the `gcloud` (/spanner/docs/gcloud-spanner) command-line tool:

After you create the database, follow the instructions in Applying IAM Roles (/spanner/docs/grant-permissions) to create user accounts and grant permissions to the Cloud Spanner instance and database.

In addition to the code needed to replace the preceding database objects (#other-database-objects), you must add application logic to handle the following functionality:

- Hashing primary keys for writes, for tables that have high write rates to sequential keys.
- Validating data, to replace constraints that you could not migrate from the PostgreSQL schema.
- Referential integrity checks not already covered by table interleaving or application logic, including functionality handled by triggers in the PostgreSQL schema.

We recommend using the following process when refactoring:

1. Find all of your application code that accesses the database, and refactor it into a single module or library. That way, you know exactly what code accesses to the database, and therefore exactly what code needs to be modified.
2. Write code that performs reads and writes on the Cloud Spanner instance, providing parallel functionality to the original code that reads and writes to PostgreSQL. During writes, update the entire row, not just the columns that have been changed, to ensure that the data in Cloud Spanner is identical to that in PostgreSQL.
3. Write code that replaces the functionality of the database objects and functions that aren't available in Cloud Spanner.

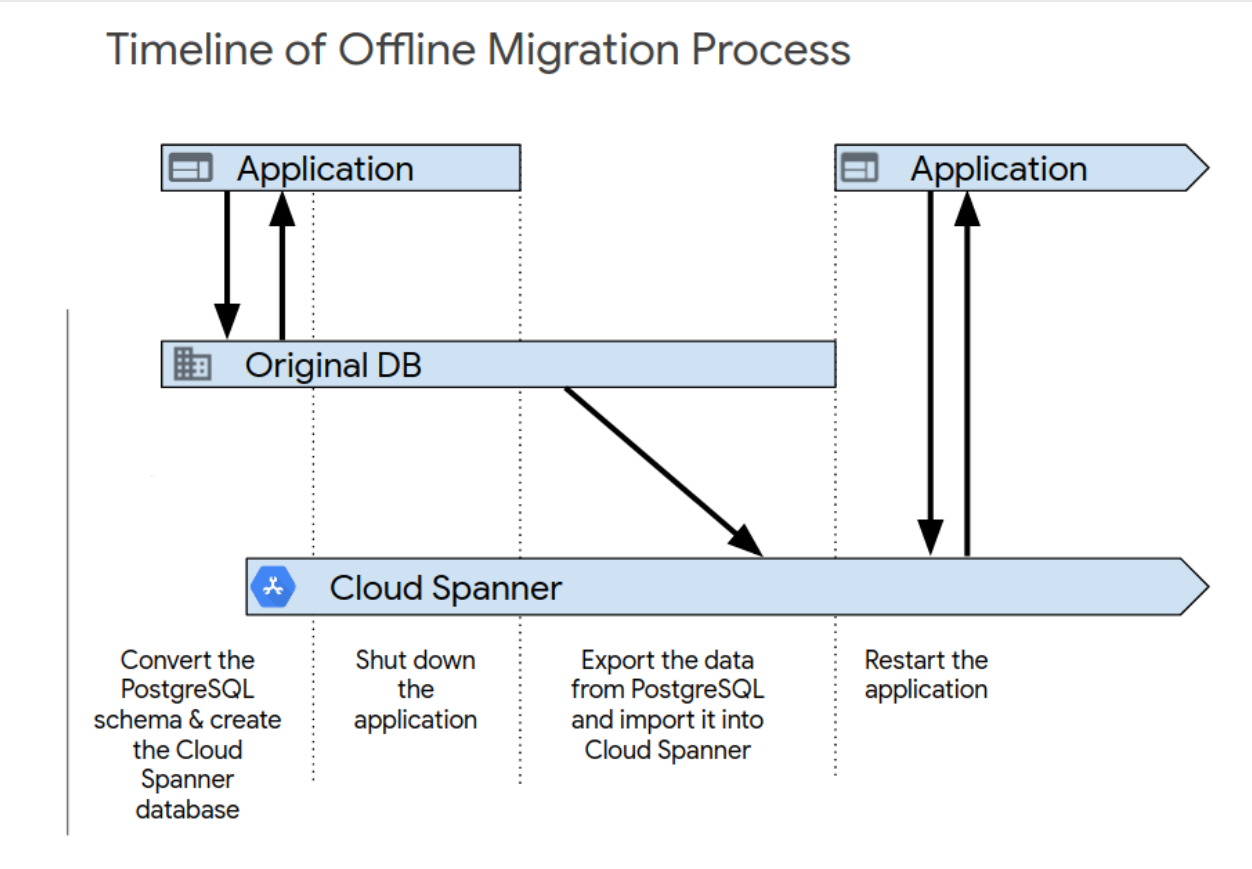
After you create your Cloud Spanner database and refactor your application code, you can migrate your data to Cloud Spanner.

1. Use the PostgreSQL `COPY` (https://www.postgresql.org/docs/10/static/sql-copy.html) command to dump data to .csv files.
2. Upload the .csv files to Cloud Storage.
    a. Create a Cloud Storage bucket (/storage/docs/creating-buckets).
    b. In the Cloud Storage console, click on the bucket name to open the bucket browser.
    c. Click **Upload Files**.
    d. Navigate to the directory containing the .csv files and select them.
    e. Click **Open**.
3. Create an application to import data into Cloud Spanner. This application could use Dataflow (/spanner/docs/dataflow-connector#writing-transforming) or it could use the client libraries (/spanner/docs/reference/libraries) directly. Make sure to follow the guidance in Bulk data loading best practices (/spanner/docs/bulk-loading) to get the best performance.

Test all application functions against the Cloud Spanner instance to verify that they work as expected. Run production-level workloads to ensure the performance meets your needs. Update the number of nodes (/spanner/docs/create-manage-instances#changing_the_number_of_nodes) as needed to meet your performance goals.

After you complete the initial application testing, turn up the new system using one of the following processes. Offline migration is the simplest way to migrate. However, this approach makes your application unavailable for a period of time, and it provides no rollback path if you find data issues later on. To perform an offline migration:

1. Delete all the data in the Cloud Spanner database.
2. Shut down the application that targets the PostgreSQL database.
3. Export all data from the PostgreSQL database and import it into the Cloud Spanner database as described in Migrating data (#migrating-data).
4. Start up the application that targets the Cloud Spanner database.



Live migration is possible and requires extensive changes to your application to support the migration.

These examples show the `CREATE TABLE` statements for several tables in the MusicBrainz (https://musicbrainz.org/) PostgreSQL database schema (https://musicbrainz.org/doc/MusicBrainz_Database/Schema). Each example includes both the PostgreSQL schema and the Cloud Spanner schema.

PostgreSQL version:

Cloud Spanner version:

PostgreSQL version:

Cloud Spanner version:

PostgreSQL version:

Cloud Spanner version: