This article explains how to migrate your Online Transactional Processing (OLTP) database from MySQL to Cloud Spanner (/spanner).

Spanner uses certain concepts differently from other enterprise database management tools, so you might need to adjust your application's architecture to take full advantage of its capabilities. You might also need to supplement Spanner with other services from Google Cloud to meet your needs.

Spanner does not support running user code in the database level, so as part of the migration, business logic implemented by database-level stored procedures and triggers must be moved into the application.

Spanner does not implement a sequence generator, and as explained below, using monotonically increasing numbers as primary keys is an anti-pattern in Spanner. An alternative mechanism of generating a unique primary key is to use a random UUID (https://wikipedia.org/wiki/Universally_unique_identifier#Version_4_(random)).

If you require sequences for external reasons, you must implement them in the application layer.

Spanner supports only database-level access controls using the Cloud Identity and Access Management (IAM) access permissions and roles. There are predefined roles that can grant read-write or read-only access to the database. If you require finer-grained permissions, you must implement them in the application layer. In a normal scenario, only the application should be allowed to read/write to the database.

If you need to expose your database to users for reporting, and wish to use fine-grained security permissions (for example, table/view level permissions), we recommend that you export your database to BigQuery (/bigquery/docs/loading-data-cloud-storage-avro).

Spanner supports a limited set of data validation constraints in the database layer. If you need more complex data constraints, you must implement them in the application layer.

The following table discusses the types of constraints commonly found in MySQL databases, and how to implement them with Spanner.

Constraint	Implementation with Spanner
Not null	`NOT NULL` column constraint
Unique	Secondary index with `UNIQUE` constraint
`Foreign key` (for normal tables)	Implemented in the application layer
Foreign key `ON DELETE/ON UPDATE` actions	Only possible for interleaved tables; otherwise, implemented in the application layer
Value checks and validation through `CHECK` constraints or triggers	Implemented in the application layer

MySQL and Spanner support different sets of data types. The following table lists the MySQL data types and their equivalent in Spanner. For detailed definitions of each Spanner data type, see Data types (/spanner/docs/data-types).

You might have to further transform your data as described in the Notes column to make MySQL data fit in your Spanner database. For example, you can store a large `BLOB` as an object in a Cloud Storage bucket rather than in the database, and then store the URI reference to the Cloud Storage object in the database as a `STRING`.

MySQL data type	Spanner equivalent	Notes
`INTEGER`, `INT`, `BIGINT` `MEDIUMINT`, `SMALLINT`	`INT64`	
`TINYINT`, `BOOL`, `BOOLEAN`	`BOOL`, `INT64`	`TINYINT(1)` values are used to represent boolean values of 'true' (nonzero) or 'false' (0).
`FLOAT`, `DOUBLE`	`FLOAT64`	

MySQL data type	Spanner equivalent	Notes
DECIMAL, NUMERIC	FLOAT64, INT64, STRING	The NUMERIC data type supports up to 65 digits of precision, while the FLOAT64 Spanner data type supports up to 16 digits of precision. See Storing arbitrary precision numeric data (/spanner/docs/storing-numeric-data) for alternative mechanisms.
BIT	BYTES	
DATE	DATE	Both Spanner and MySQL use the 'yyyy-mm-dd' format for dates, so no transformation is necessary. SQL functions are provided to convert dates to a formatted string.
DATETIME, TIMESTAMP	TIMESTAMP	Spanner stores time independent of time zone. If you need to store a time zone, you must use a separate STRING column. SQL functions are provided to convert timestamps to a formatted string using time zones.
CHAR, VARCHAR	STRING	Note: Spanner uses Unicode strings throughout. VARCHAR supports a maximum length of 65,535 bytes, while Spanner supports up to 2,621,440 characters.
BINARY, VARBINARY, BLOB, TINYBLOB	BYTES	Small objects (less than 10 MiB) can be stored as BYTES. Consider using alternative Google Cloud offerings such as Cloud Storage to store larger objects
TEXT, TINYTEXT, ENUM	STRING	Small TEXT values (less than 10 MiB) can be stored as STRING. Consider using alternative Google Cloud offerings such as Cloud Storage to support larger TEXT values.
ENUM	STRING	Validation of ENUM values must be performed in the application
SET	ARRAY<STRING>	Validation of SET element values must be performed in the application
LONGBLOB, MEDIUMBLOB	BYTES or STRING containing URI to object.	Small objects (less than 10 MiB) can be stored as BYTES. Consider using alternative Google Cloud offerings such as Cloud Storage to store larger objects.
LONGTEXT, MEDIUMTEXT	STRING (either containing data or URI to external object)	Small objects (less than 2,621,440 characters) can be stored as STRING. Consider using alternative Google Cloud offerings such as Cloud Storage to store larger objects

MySQL data type	Spanner equivalent	Notes
`JSON`	`STRING` (either containing data or URI to external object)	Small JSON strings (less than 2,621,440 characters) can be stored as `STRING`. Consider using alternative Google Cloud offerings such as Cloud Storage to store larger objects.
`GEOMETRY`, `POINT`, `LINESTRING`, `POLYGON`, `MULTIPOINT`, `MULTIPOLYGON`, `GEOMETRYCOLLECTION`		Spanner does not support Geospatial data types. You must store this data using standard data types, and implement any searching/filtering logic in the application layer.

An overall timeline of your migration process would be:

- Convert your schema and data model.
- Translate any SQL queries.
- Migrate your application to use Spanner in addition to MySQL.
- Bulk export your data from MySQL and import your data into Spanner using Dataflow.
- Maintain consistency between both databases during your migration.
- Migrate your application away from MySQL.

You convert your existing schema to a Spanner schema (/spanner/docs/schema-and-data-model) to store your data. In order to make application modifications simpler, make sure the converted schema matches the existing MySQL schema as closely as possible. However, because of the differences in features, some changes might be necessary.

Using best practices in schema design (/spanner/docs/schema-design) can help you increase throughput and reduce hot spots in your Spanner database.

Every table that must store more than one row must have a primary key consisting of one or more columns of the table. Your table's primary key uniquely identifies each row in a table, and because the

table rows are sorted by primary key, the table itself acts as a primary index.

It's best to avoid designating columns that monotonically increase or decrease as the first part of the primary key (examples include sequences or timestamps), because this can lead to hot spots caused by inserts occurring at the end of your keyspace. A hot spot is a concentration of operations on a single node, which lowers the write throughput to the node's capacity instead of benefiting from load-balancing all writes among the spanner nodes.

Use the following techniques to generate unique primary key values and reduce the risk of hot spots:

- Swap the order of keys (/spanner/docs/schema-design#fix_swap_key_order) so that the column that contains the monotonically increasing or decreasing value is not the first key part.
- Hash the unique key and spread the writes across logical shards (/spanner/docs/schema-design#fix_hash_the_key) by creating a column that contains the hash of the unique key, and then use the hash column (or the hash column and the unique key columns together) as the primary key. This approach helps avoid hot spots because new rows are spread more evenly across the keyspace.
- Use a Universally Unique Identifier (UUID) (/spanner/docs/schema-design#uuid_primary_key) as defined by RFC 41122 (https://tools.ietf.org/html/rfc4122) as the primary key. We recommend using version 4 UUID because it uses random values in the bit sequence.
- Bit-reverse sequential values (/spanner/docs/schema-design#bit_reverse_primary_key) to distribute high-order bits of subsequent numbers roughly equally over the entire number space.

After you designate your primary key for your table, you cannot change it later without deleting and recreating the table. For more information on how to designate your primary key, see Schema and data model - primary keys (/spanner/docs/schema-and-data-model#primary_keys).

Here is an example DDL statement creating a table for a database of music tracks:

Spanner has a feature where you can define two tables as having a 1-many, parent-child relationship (/spanner/docs/schema-and-data-model#parent-child_table_relationships). This feature interleaves the child data rows next to their parent row in storage, effectively pre-joining the table and improving data retrieval efficiency when the parent and children are queried together.

The child table's primary key must start with the primary key column(s) of the parent table. From the child row's perspective, the parent row primary key is referred to as a foreign key. You can define up to 6 levels of parent-child relationships.

You can define on-delete actions (/spanner/docs/schema-and-data-model#creating-interleaved-tables) for child tables to determine what happens when the parent row is deleted: either all child rows are deleted, or the parent row deletion is blocked while child rows exist.

Here is an example of creating an Albums table interleaved in the parent Singers table defined earlier:

You can also create secondary indexes (/spanner/docs/secondary-indexes) to index data within the table outside of the primary key. Spanner implements secondary indexes in the same way as tables, so the column values to be used as index keys will have the same constraints (/spanner/docs/schema-and-data-model#primary_keys) as the primary keys of tables. This also means that indexes have the same consistency guarantees as Spanner tables.

Value lookups using secondary indexes are effectively the same as a query with a table join. You can improve the performance of queries using indexes by storing copies the original table's column values in the secondary index using the `STORING` clause, making it a covering index (https://wikipedia.org/wiki/Database_index#Covering_index).

Spanner's query optimizer only automatically uses a secondary index when the index itself stores all the columns being queried (a covered query). To force the use of an index when querying columns in the original table, you must use a FORCE INDEX directive (/spanner/docs/secondary-indexes#index_directive) in the SQL statement, for example:

Indexes can be used to enforce unique values within a table column, by defining a `UNIQUE` index (/spanner/docs/secondary-indexes#unique_indexes) on that column. Adding duplicate values will be prevented by the index.

Here is an example DDL statement creating a secondary index for the Albums table:

If you create additional indexes after your data is loaded, populating the index might take some time. We recommend that you limit the rate at which you add them to an average of three per day. For more guidance on creating secondary indexes, see Secondary indexes (/spanner/docs/secondary-indexes). For more information on the limitations on index creation, see Schema updates (/spanner/docs/schema-updates#large-updates).

Spanner uses the ANSI 2011 dialect of SQL with extensions (/spanner/docs/query-syntax), and has many functions and operators (/spanner/docs/functions-and-operators) to help translate and aggregate your data. Any SQL queries using MySQL-specific dialect, functions, and types will need to be converted to be compatible with Spanner.

Although Spanner doesn't support structured data as column definitions, you can use structured data in SQL queries using `ARRAY<>` and `STRUCT<>` types. For example, you could write a query that returns all Albums for an artist using an `ARRAY` of `STRUCT`s (taking advantage of the pre-joined data). For more information see the Notes about subqueries (/spanner/docs/query-syntax#notes-about-subqueries) section of the documentation.

SQL queries can be profiled using the Spanner Query interface in the Cloud Console to execute the query. In general, queries that perform full table scans on large tables are very expensive, and should be used sparingly. For more information on optimizing SQL queries, see the SQL best practices (/spanner/docs/sql-best-practices) documentation.

Spanner provides a set of Client libraries (/spanner/docs/reference/libraries) for various languages, and the ability to read and write data using Spanner-specific API calls, as well as by using SQL queries (/spanner/docs/query-syntax) and Data Modification Language (DML) (/spanner/docs/dml-syntax) statements. Using API calls might be faster for some queries, such as direct row reads by key, because the SQL statement doesn't have to be translated.

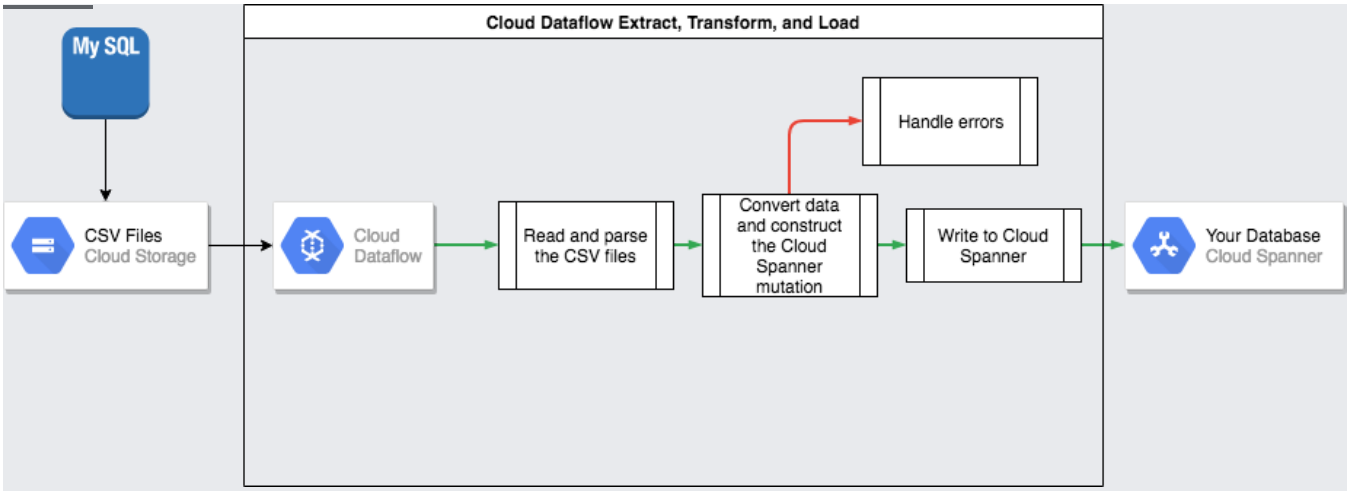
You can also use the Java Database Connectivity (JDBC) driver (/spanner/docs/partners/drivers) to connect to Spanner, leveraging existing tooling and infrastructure that does not have native integration.

As part of the migration process, features not available in Spanner as mentioned above must be implemented in the application. For example, a trigger to verify data values and update a related table would need to be implemented in the application using a read/write transaction to read the existing row, verify the constraint, then write the updated rows to both tables.

Spanner offers read/write and read-only transactions (/spanner/docs/transactions), which ensure external consistency of your data. Additionally, read transactions can have Timestamp bounds (/spanner/docs/timestamp-bounds) applied, where you are reading a consistent version of the data either:

- at an exact time in the past (up to 1 hour ago).
- in the future (where the read will block until that time arrives).
- with an acceptable amount of bounded staleness, which will return a consistent view up to some time in the past without needing to check that later data is available on another replica. This can give performance benefits at the expense of possibly stale data.

To transfer your data from MySQL to Spanner, you must export your MySQL database to a portable file format—for example, XML—and then import that data into Spanner using Dataflow.



The `mysqldump` tool included with MySQL is able to export the entire database into well-formed XML files (https://dev.mysql.com/doc/refman/8.0/en/mysqldump.html#option_mysqldump_xml). Alternatively, you can use the `SELECT ... INTO OUTFILE` (https://dev.mysql.com/doc/refman/8.0/en/select-into.html) SQL statement to create CSV files for each table. However, this approach has the disadvantage that only one table can be exported at a time, which means that you must either pause your application or quiesce your database so that the database remains in a consistent state for export.

After exporting these data files, we recommend that you upload them to a Cloud Storage (/storage) bucket so that they are accessible for import.

Because database schemas likely differ between MySQL and Spanner, you might need to make some data conversions part of the import process. The easiest way to perform these data conversions and import the data into Spanner is by using Dataflow (/dataflow/). Dataflow is the Google Cloud distributed extract, transform, and load (ETL) service. It provides a platform for running data pipelines written using the Apache Beam SDK (https://beam.apache.org/get-started/beam-overview/) in order to read and process large amounts of data in parallel over multiple machines.

The Apache Beam SDK requires you to write a simple Java program to set read, transform and write the data. Beam connectors exist for Cloud Storage and Spanner, so the only code that you need to write is the data transform itself.

For an example of a simple pipeline that reads from CSV files and writes to Spanner, see the sample code repository

(https://github.com/GoogleCloudPlatform/java-docs-samples/blob/master/dataflow/spanner-io/src/main/java/com/example/dataflow/SpannerWrite.java)
.

If you use parent-child interleaved tables in your Spanner schema, take care in the import process that the parent row is created before the child row. The Spanner import pipeline code (https://github.com/GoogleCloudPlatform/DataflowTemplates/blob/master/src/main/java/com/google/cloud/teleport/spanner/ImportTransform.java) handles this by importing all data for root-level tables first, then all the level 1 child tables, then all the level 2 child tables, and so on.

You can use the Spanner import pipeline directly to bulk import your data (/spanner/docs/import-non-spanner), but this approach requires your data to exist in Avro files using the correct schema.

Many applications have availability requirements that make it impossible to keep the application offline for the time required to export and import your data. Therefore, while you are transferring your data to Spanner, your application continues to modify the existing database. So it is necessary to duplicate updates to the Spanner database while the application is running.

There are various methods of keeping your two databases in sync, including change data capture, and implementing simultaneous updates in the application.

MySQL doesn't have a native change data capture (https://wikipedia.org/wiki/Change_data_capture) (CDC) utility. However, there are various open source projects that can receive MySQL binlogs and convert them into a CDC stream. For example Maxwell's daemon (http://maxwells-daemon.io/) can provide a CDC stream for your database.

You can write an application that subscribes to this stream and applies the same modifications (after data conversion, of course) to your Spanner database.

An alternative method is to modify your application to perform writes to both databases. One database (initially MySQL) would be considered the source of truth, and after each database write,

the entire row is read, converted and written to the Spanner database. In this way, the application constantly overwrites the Spanner rows with the latest data.

When you're confident that all your data has been transferred correctly, you can switch the source of truth to the Spanner database. This mechanism provides a rollback path if issues are found when switching to Spanner.

As data streams into your Spanner database, you can periodically run a comparison between your Spanner data and your MySQL data to make sure that the data is consistent. You can validate consistency by querying both data sources and comparing the results.

You can use Dataflow to perform a detailed comparison over large data sets by using the Join transform (https://beam.apache.org/documentation/pipelines/design-your-pipeline/#multiple-sources). This transform takes 2 keyed data sets, and matches the values by key. The matched values can then be compared for equality. You can regularly run this verification until the level of consistency matches your business requirements.

When you are confident in the data migration, you can switch your application to using Spanner as the source of truth. If you continue writing back changes to the MySQL database, this keeps the MySQL database up to date, giving a rollback path should issues arise.

Finally, you can disable and remove the MySQL database update code and shut down the now-obsolete MySQL database.

You can optionally export your tables from Spanner to a Cloud Storage bucket using a Dataflow template to perform the export. The resulting folder contains a set of Avro files and JSON manifest files containing your exported tables. These files can serve various purposes, including:

- Backing up your database for data-retention policy compliance or disaster recovery.
- Importing the Avro file into other Google Cloud offerings such as BigQuery.

For more information on the export and import process, see Exporting databases (/spanner/docs/export) and Importing databases (/spanner/docs/import).

- Read about how to optimize your Spanner schema (/spanner/docs/whitepapers/optimizing-schema-design).
- Learn how to use Dataflow (/dataflow/docs/how-to) for more complex situations.
- Lean how to choose the best Google Cloud storage service (/storage-options/) for your application.
- Review other Spanner how-to guides (/spanner/docs/how-to).
- Try out other Google Cloud features for yourself. Have a look at our tutorials (/docs/tutorials).