In a Cloud Spanner database, Cloud Spanner automatically creates an index for each table's primary key column. For example, you don't need to do anything to index the primary key column of `Singers`, because it's automatically indexed for you.

You can also create *secondary indexes* for other columns. Adding a secondary index on a column makes it more efficient to look up data in that column. For example, if you need to quickly look up a set of `SingerId` values for a given range of `LastName` values, you should create a secondary index on `LastName`, so Cloud Spanner does not need to scan the entire table.

Cloud Spanner stores the following data in each secondary index:

- All key columns from the base table
- All columns that are included in the index
- All columns specified in the optional `STORING` clause (#storing-clause) of the index definition

Over time, Cloud Spanner analyzes your tables to ensure that your secondary indexes are used for the appropriate queries.

The most efficient time to add a secondary index is when you create the table. To create a table and its indexes at the same time, send the DDL statements for the new table and the new indexes in a single request to Cloud Spanner.

In Cloud Spanner, you can also add a new secondary index to an existing table while the database continues to serve traffic. Like any other schema changes in Cloud Spanner, adding an index to an existing database does not require taking the database offline and does not lock entire columns or tables.

Whenever a new index is added to an existing table, Cloud Spanner automatically *backfills*, or populates, the index to reflect an up-to-date view of the data being indexed. Cloud Spanner manages this backfill process for you, and it uses additional resources during the index backfill.

Index creation can take from several minutes to many hours. Because index creation is a schema update, it is bound by the same performance constraints (/spanner/docs/schema-updates#performance) as any other schema update. The time needed to create a secondary index depends on several factors:

- The size of the data set
- The number of nodes in the instance
- The load on the instance

Be aware that using the commit timestamp (/spanner/docs/commit-timestamp) column as the first part of the secondary index can create hotspots (/spanner/docs/schema-design#primary-key-prevent-hotspots) and reduce write performance.

If you are adding many secondary indexes to a database, follow the guidance for large schema updates (/spanner/docs/schema-updates#large-updates) when you create the indexes.

Use the `CREATE INDEX` (/spanner/docs/data-definition-language#create_index) statement to define a secondary index in your schema. Here are some examples:

To index all `Singers` in the database by their first and last name:

To create an index of all `Songs` in the database by the value of `SongName`:

To index only the songs for a particular singer, interleave the index in the table `Singers`:

To index only the songs on a particular album:

To index by descending order of `SongName`:

Note that the `DESC` annotation above applies only to `SongName`. To index by descending order of other index keys, annotate them with `DESC` as well: `SingerId DESC, AlbumId DESC`.

You can use the Cloud SDK to cancel index creation. To retrieve a list of schema-update operations for a Cloud Spanner database, use the `gcloud spanner operations list` (/sdk/gcloud/reference/spanner/operations/list) command, and include the `--filter` option:

Find the `OPERATION_ID` for the operation you want to cancel, then use the `gcloud spanner operations cancel` (/sdk/gcloud/reference/spanner/operations/cancel) command to cancel it:

To view information about existing indexes in a database, you can use the Google Cloud Console or the `gcloud` command-line tool:

The following sections explain how to specify an index in a SQL statement and with the read interface for Cloud Spanner. The examples in these sections assume that you added a `MarketingBudget` column to the `Albums` table and created an index called `AlbumsByAlbumTitle`:

When you use SQL to query a Cloud Spanner table, Cloud Spanner automatically uses any indexes that are likely to make the query more efficient. As a result, you typically don't need to specify an index for SQL queries.

In a few cases, though, Cloud Spanner might choose an index that causes query latency to increase. If you've followed the troubleshooting steps for performance regressions (/spanner/docs/troubleshooting-performance-regressions) and confirmed that it makes sense to try a different index for the query, you can specify the index as part of your query.

To specify an index in a SQL statement, use `FORCE_INDEX` (/spanner/docs/query-syntax#table-hints) to provide an *index directive*. Index directives use the following syntax:

You can also use an index directive to tell Cloud Spanner to scan the base table instead of using an index:

The following example shows a SQL query that specifies an index:

An index directive might force Cloud Spanner's query processor to read additional columns that are required by the query but not stored in the index. The query processor retrieves these columns by

joining the index and the base table. To avoid this extra join, use a `STORING` clause (#storing-clause) to store the additional columns in the index.

For instance, in the example shown above, the `MarketingBudget` column is not stored in the index, but the SQL query selects this column. As a result, Cloud Spanner must look up the `MarketingBudget` column in the base table, then join it with data from the index, to return the query results.

Cloud Spanner raises an error if the index directive has any of the following issues:

- The index does not exist.
- The index is on a different base table.
- The query is missing a required `NULL` filtering expression (#null-indexing-disable) for a `NULL_FILTERED` (#null-indexing) index.

The following examples show how to write and execute queries that fetch the values of `AlbumId`, `AlbumTitle`, and `MarketingBudget` using the index `AlbumsByAlbumTitle`:

When you use the read interface to Cloud Spanner, and you want Cloud Spanner to use an index, you must specify the index. The read interface does not select the index automatically.

In addition, your index must contain all of the data that appears in the query results, excluding columns that are part of the primary key. This restriction exists because the read interface does not support joins between the index and the base table. If you need to include other columns in the query results, you have a few options:

- Use a `STORING` clause (#storing-clause) to store the additional columns in the index.
- Query without including the additional columns, then use the primary keys to send another query that reads the additional columns.

Cloud Spanner returns values from the index in ascending sort order by index key. To retrieve values in descending order, complete these steps:

- Annotate the index key with `DESC`. For example:

  The `DESC` annotation applies to a single index key. If the index includes more than one key, and you want the query results to appear in descending order based on all keys, include a `DESC` annotation for each key.
- If the read specifies a key range, ensure that the key range is also in descending order. In other words, the value of the start key must be greater than the value of the end key.

The following example shows how to retrieve the values of `AlbumId` and `AlbumTitle` using the index `AlbumsByAlbumTitle`:

Optionally, you can use the `STORING` clause to store a copy of a column in the index. This type of index provides advantages for queries and read calls using the index, at the cost of using extra storage:

- SQL queries that use the index and select columns stored in the `STORING` clause do not require an extra join to the base table.
- Read calls that use the index can read columns stored in the `STORING` clause.

For example, suppose you created an alternate version of `AlbumsByAlbumTitle` that stores a copy of the `MarketingBudget` column in the index (note the `STORING` clause in bold):

With the old `AlbumsByAlbumTitle` index, Cloud Spanner must join the index with the base table, then retrieve the column from the base table. With the new `AlbumsByAlbumTitle2` index, Cloud Spanner reads the column directly from the index, which is more efficient.

If you use the read interface instead of SQL, the new `AlbumsByAlbumTitle2` index also lets you read the `MarketingBudget` column directly:

By default, Cloud Spanner indexes `NULL` values. For example, recall the definition of the index `SingersByFirstLastName` on the table `Singers`:

All rows of `Singers` are indexed even if either `FirstName` or `LastName`, or both, are `NULL`.

Singers

SingerId	FirstName	LastName
11	"Milo"	"Matt"
12	NULL	"Wyatt"
13	"Kena"	NULL
14	NULL	NULL

SingersByFirstLastName

FirstName	LastName	SingerId
NULL	***NULL***	***14***
NULL	***"Wyatt"***	***12***
"Kena"	***NULL***	***13***
"Milo"	"Matt"	11

Note: Rows with ***bolded, italicized*** values would not be present in a NULL_FILTERED index

(/spanner/docs/images/indexing_nulls.svg)

When `NULL` values are indexed, you can perform efficient SQL queries and reads over data that includes `NULL` values. For example, use this SQL query statement to find all `Singers` with a `NULL` `FirstName`:

Cloud Spanner sorts `NULL` as the smallest value for any given type. For a column in ascending (`ASC`) order, `NULL` values sort first. For a column in descending (`DESC`) order, `NULL` values sort last.

To disable the indexing of nulls, add the `NULL_FILTERED` keyword to the index definition. `NULL_FILTERED` indexes are particularly useful for indexing sparse columns, where most rows contain a `NULL` value. In these cases, the `NULL_FILTERED` index can be considerably smaller and more efficient to maintain than a normal index that includes `NULL` values.

Here's an alternate definition of `SingersByFirstLastName` that does not index `NULL` values:

The `NULL_FILTERED` keyword applies to all index key columns. You cannot specify `NULL` filtering on a per-column basis.

Making an index `NULL_FILTERED` prevents Cloud Spanner from using it for some queries. For example, Cloud Spanner does not use the index for this query, because the index omits any `Singers` rows for which `LastName` is `NULL`; as a result, using the index would prevent the query from returning the correct rows:

To enable Cloud Spanner to use the index, you must rewrite the query so it excludes the rows that are also excluded from the index:

Indexes can be declared `UNIQUE`. `UNIQUE` indexes add a constraint to the data being indexed that prohibits duplicate entries for a given index key. This constraint is enforced by Cloud Spanner at transaction commit time. Specifically, any transaction that would cause multiple index entries for the same key to exist will fail to commit.

If a table contains non-`UNIQUE` data in it to begin with, attempting to create a `UNIQUE` index on it will fail.

A `UNIQUE NULL_FILTERED` index does not enforce index key uniqueness when at least one of the index's key parts is NULL.

For example, suppose that you created the following table and index:

The following two rows in `ExampleTable` have the same values for the secondary index keys `Key1`, `Key2` and `Col1`:

Because `Key2` is `NULL` and the index is `NULL_FILTERED`, the rows will not be present in the index `ExampleIndex`. Because they are not inserted into the index, the index will not reject them for violating uniqueness on `(Key1, Key2, Col1)`.

If you want the index to enforce the uniqueness of values of the tuple (`Key1`, `Key2`, `Col1`), then you must annotate `Key2` with `NOT NULL` in the table definition or create the index without `NULL_FILTERED`.

If a key column contains **`NULL`** values, don't create a **`UNIQUE NULL_FILTERED`** index that contains only that key colu
d, add another column to the index that does not contain **`NULL`** values.

Use the `DROP INDEX` (/spanner/docs/data-definition-language#drop-index) statement to drop a secondary index from your schema.

To drop the index named `SingersByFirstLastName`:

- Learn about SQL best practices for Cloud Spanner (/spanner/docs/sql-best-practices).

- Understand query execution plans for Cloud Spanner (/spanner/docs/query-execution-plans).
- Find out how to troubleshoot performance regressions in SQL queries (/spanner/docs/troubleshooting-performance-regressions).