This topic describes how to write a commit timestamp for each insert and update operation that you perform with Cloud Spanner. To use this feature, set the `allow_commit_timestamp` option on a `TIMESTAMP` column, then write the timestamp as part of each transaction.

The commit timestamp, based on TrueTime technology, is the time when a transaction is committed in the database. The `allow_commit_timestamp` column option allows you to atomically store the commit timestamp into a column. Using the commit timestamps stored in tables, you can determine the exact ordering of mutations and build features like changelogs.

To insert commit timestamps in your database, complete the following steps:

1. Create a `TIMESTAMP` column (#create-column) with the column option `allow_commit_timestamp` set to `true` in the schema definition.
2. If you are performing inserts or updates with DML, use the `PENDING_COMMIT_TIMESTAMP` function (#dml) to write the commit timestamp.

   If you are performing inserts or updates with mutations, use the placeholder string `spanner.commit_timestamp()` (#insert-row) (or the client library constant) on insertions or updates to your commit timestamp column.

When Cloud Spanner commits the transaction, the commit timestamp is written to the `LastUpdateTime` column. You could then use `LastUpdateTime` to create a history of updates to the `Performances` table.

Commit timestamp values are not guaranteed to be unique. Transactions that write to non-overlapping sets of fields might have the same timestamp. Transactions that write to overlapping sets of fields have unique timestamps.

Cloud Spanner commit timestamps have microsecond granularity, and they are converted to nanoseconds when stored in `TIMESTAMP` columns.

Use the `allow_commit_timestamp` option to add and remove support for commit timestamps:

- To create a new column, in a new or existing table, that supports commit timestamps.
- To alter an existing timestamp column to support commit timestamps.
- To remove commit timestamp support from a column.

You can use a commit timestamp column as a primary key column or as a non-key column. Primary keys can be defined as `ASC` or `DESC`.

- `ASC` (default) - Ascending keys are ideal for answering queries from a specific time forward.
- `DESC` - Descending keys keep the latest rows at the top of the table. They provide quick access to the latest records.

The `allow_commit_timestamp` option must be consistent across the primary keys of parent and child tables. If the option is not consistent across primary keys, Cloud Spanner returns an error. The only time the option can be inconsistent is when you are creating or updating the schema.

Using commit timestamps under the following scenarios creates hotspots (/spanner/docs/schema-design#primary-key-prevent-hotspots) which reduce data performance:

- Commit timestamp column as the first part of the primary key of a table:

- The first part of the primary key of a secondary index:

or

Hotspots (/spanner/docs/schema-design#primary-key-prevent-hotspots) reduce data performance, even with low write rates. There is no performance overhead if commit timestamps are enabled on non-key columns that are not indexed.

The following example creates a table with a column that supports commit timestamps.

Adding the option changes the timestamp column as follows:

- You can use the `spanner.commit_timestamp()` placeholder string (or a constant provided by the client library) for inserts and updates.
- The column can only contain values in the past. For more information, see Providing your own value for the timestamp (#provide-timestamp).

The option `allow_commit_timestamp` is case sensitive.

To add a commit timestamp column to an existing table, use the `ALTER TABLE` statement:

You can convert an existing timestamp column into a commit timestamp column, but doing so requires Cloud Spanner to validate that the existing timestamp values are in the past. For example:

You cannot change the data type or `NULL` annotation of a column in an `ALTER TABLE` statement that includes `SET OPTIONS`. For details, see Data Definition Language (/spanner/docs/data-definition-language).

If you want to remove commit timestamp support from a column, use the option `allow_commit_timestamp=null` in an `ALTER TABLE` statement. The commit timestamp behavior is removed, but the column is still a timestamp. Changing the option does not alter any other characteristics of the column, such as type or nullability (`NOT NULL`). For example:

You use the `PENDING_COMMIT_TIMESTAMP` (https://cloud.google.com/spanner/docs/functions-and-operators#timestamp-functions) function to write the

commit timestamp in a DML statement. Cloud Spanner selects the commit timestamp when the transaction commits.

After you call the `PENDING_COMMIT_TIMESTAMP` method, the table and any derived index is unreadable to any fut atements in the transaction. You must write commit timestamps as the last statement in a transaction to prevent th ility of trying to read the table. If you try to read the table, then Cloud Spanner returns an error.

The following DML statement updates the `LastUpdated` column in the `Singers` table with the commit timestamp:

When inserting a row, Cloud Spanner writes the commit timestamp value only if you include the column in the column list and pass the `spanner.commit_timestamp()` placeholder string (or client library constant) as its value. For example:

Commit timestamps can only be written to columns annotated with the `allow_commit_timestamp=true` option.

If you have mutations on rows in multiple tables, you must specify `spanner.commit_timestamp()` (or client library constant) for the commit timestamp column in each table.

When updating a row, Cloud Spanner writes the commit timestamp value only if you include the column in the column list and pass the `spanner.commit_timestamp()` placeholder string (or client library constant) as its value. You cannot update the primary key of a row. To update the primary key, delete the existing row and create a new row.

For example, to update a commit timestamp column named `LastUpdateTime`:

Commit timestamps can only be written to columns annotated with the `allow_commit_timestamp=true` option.

If you have mutations on rows in multiple tables, you must specify `spanner.commit_timestamp()` (or the client library constant) for the commit timestamp column in each table.

The following example queries the commit timestamp column of the table.

You can provide your own value for the commit timestamp column, instead of passing `spanner.commit_timestamp()` (or client library constant) as the column value. The value must be a timestamp in the past. This restriction ensures that writing timestamps is an inexpensive and fast operation. An easy way to confirm that a value is in the past is to compare it to the value returned by the `CURRENT_TIMESTAMP` SQL function. The server returns a `FailedPrecondition` error if a future timestamp is specified.

Suppose that you want to create a changelog of every mutation that happens to a table and then use that changelog for auditing. An example would be a table that stores the history of changes to word processing documents. The commit timestamp makes creating the changelog easier, because the timestamps can enforce ordering of the changelog entries. You could build a changelog that stores the history of changes to a given document using a schema like the following example:

To create a changelog, insert a new row in `DocumentHistory` in the same transaction in which you insert or update a row in `Document`. In the insertion of the new row in `DocumentHistory`, use the placeholder `spanner.commit_timestamp()` (or client library constant) to tell Cloud Spanner to write the commit timestamp into column `Ts`. Interleaving the `DocumentsHistory` table with the `Documents` table will allow for data locality and more efficient inserts and updates. However, it also adds the constraint that the parent and child rows must be deleted together. To keep the rows in `DocumentHistory` after rows in `Documents` are deleted, do not interleave the tables.

The size of a row (/spanner/docs/schema-design#limit_row_size) should be less than 4 GB for best performance. (The size of a row includes the top-level row and all of its interleaved child and index rows.) In this example, there is a limit to how many rows there can be in `DocumentHistory` for a particular document, because of the row size limit. If you expect the changelog in `DocumentHistory` to be large, you can design your app to delete the oldest rows in `DocumentHistory`. Alternatively, you can design your schema so that `DocumentHistory` is a top-level table instead of an interleaved table.

Use commit timestamps to create a change log with Go (https://cloud.google.com/community/tutorials/cloud-spanner-commit-timestamp-change-log).