This page describes best practices for using Data Manipulation Language (DML) and Partitioned DML.

You execute DML statements inside read-write transactions. When Cloud Spanner reads data, it acquires shared read locks on limited portions of the row ranges that you read. Specifically, it acquires these locks only on the columns you access. The locks can include data that does not match the filter condition of the `WHERE` clause.

When Cloud Spanner modifies data using DML statements, it acquires exclusive locks on the specific data that you are modifying. In addition, it acquires shared locks in the same way as when you read data. If your request includes large row ranges, or an entire table, the shared locks might prevent other transactions from completing in parallel.

To modify data as efficiently as possible, use a `WHERE` clause that enables Cloud Spanner to read only the necessary rows. You can achieve this goal with a filter on the primary key, or on the key of a secondary index. The `WHERE` clause limits the scope of the shared locks and enables Cloud Spanner to process the update more efficiently.

For example, suppose that one of the musicians in the `Singers` table changes their first name, and you need to update the name in your database. You could execute the following DML statement, but it forces Cloud Spanner to scan the entire table and acquires shared locks that cover the entire table. As a result, Cloud Spanner must read more data than necessary, and concurrent transactions cannot modify the data in parallel:

To make the update more efficient, include the `SingerId` column in the `WHERE` clause. The `SingerId` column is the only primary key column for the `Singers` table:

Cloud Spanner buffers insertions, updates, and deletions performed using DML statements on the server-side, and the results are visible to subsequent SQL and DML statements within the same transaction. This behavior is different from the Mutation API (/spanner/docs/modify-mutation-api), where Cloud Spanner buffers the mutations on the client-side and sends the mutations server-side as part of the commit operation. As a result, mutations in the commit request are not visible to SQL or DML statements within the same transaction.

You might want to combine DML statements and mutations in the same transaction, because some operations are only supported in the Mutation API. An example is `insert_or_update` (/spanner/docs/reference/rpc/google.spanner.v1#google.spanner.v1.Mutation). If a transaction contains both DML statements and mutations in the commit request, Cloud Spanner executes the DML statements before the mutations. To avoid having to account for the order of execution in your client library code, you should use either DML statements or the mutations in a single transaction, but not both. If you use both, you should buffer writes only at the very end of the transaction.

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.

Partitioned DML uses one or more transactions that might run and commit at different times. If you use the date (/spanner/docs/functions-and-operators#date-functions) or timestamp (/spanner/docs/functions-and-operators#timestamp-functions) functions, the modified rows might contain different values.