The Cloud Spanner client libraries manage sessions automatically. This page is intended for creators of client libraries of the REST (/spanner/reference/rest/) or RPC (/spanner/reference/rpc/) APIs.

This page describes the advanced concept of sessions in Cloud Spanner, including best practices for sessions when creating a client library or when using the REST or RPC APIs.

A session represents a communication channel with the Cloud Spanner database service. A session is used to perform transactions that read, write, or modify data in a Cloud Spanner database. Each session applies to a single database.

Sessions can execute only one transaction (/spanner/docs/transactions) at a time. Standalone reads, writes, and queries use a transaction internally, and count toward the one transaction limit.

Creating a session is expensive. To avoid the performance cost each time a database operation is made, clients should keep a *session cache*, which is a pool of available sessions that are ready to use. The cache should store existing sessions and return the appropriate type of session when requested, as well as handle cleanup of unused sessions. For an example of how to implement a session cache, see the source code for one of the Cloud Spanner client libraries, such as the Go client library (https://github.com/GoogleCloudPlatform/google-cloud-go/blob/master/spanner/session.go) or the Java client library (https://github.com/googleapis/google-cloud-java/blob/master/google-cloud-clients/google-cloud-spanner/src/main/java/com/google/cloud/spanner/SessionPool.java).

Sessions are intended to be long-lived, so after a session is used for a database operation, the client should return the session to the cache for reuse.

The following describes best practices for implementing sessions in a client library for Cloud Spanner, or for using sessions with the REST (/spanner/reference/rest/) or RPC (/spanner/reference/rpc/) APIs.

To determine an optimal size of the session cache for a client process, set the lower bound to the number of expected concurrent transactions, and set the upper bound to an initial test number, such as 100. (For users working with the RPC API, we recommend having the cache store no more than 100 sessions, because 100 is the maximum number of concurrent sessions per gRPC channel.) If the upper bound is not adequate, increase it. Increasing the number of active sessions uses additional resources on the Cloud Spanner database service, so failing to clean up unused sessions can degrade performance or prevent you from using your Cloud Spanner database for up to an hour.

There are two ways to delete a session:

- A client can delete a session.
- The Cloud Spanner database service can delete a session when the session is idle for more than 1 hour.

Attempts to use a deleted session result in `NOT_FOUND` (/spanner/docs/reference/rpc/google.rpc#google.rpc.Code). If you encounter this error, create and use a new session, add the new session to the cache, and remove the deleted session from the cache.

The Cloud Spanner database service reserves the right to drop an unused session. If you definitely need to keep an idle session alive, for example, if a significant near-term increase in database use is expected, then you can prevent the session from being dropped. Perform an inexpensive operation such as executing the SQL query `SELECT 1` to keep the session alive. If you have an idle session that is not needed for near-term use, let Cloud Spanner drop the session, and then create a new session the next time a session is needed.

One scenario for keeping sessions alive is to handle regular peak demand on the database. If heavy database use occurs daily from 9:00 AM to 6:00 PM, you should keep some idle sessions alive during that time, since they are likely required for the peak usage. After 6:00 PM, you can let Cloud Spanner

drop idle sessions. Prior to 9:00 AM each day, create some new sessions so they will be ready for the expected demand.

Another scenario is if you have an application that uses Cloud Spanner but must avoid the connection overhead when it does. You can keep a set of sessions alive to avoid the connection overhead.

If you are creating a client library, do not expose sessions to the client library consumer. Provide the ability for the client to make database calls without the complexity of creating and maintaining sessions. For an example of a client library that hides the session details from the client library consumer, see the Cloud Spanner client library for Java.

Write transactions without replay protection may apply mutations more than once. If a mutation is not idempotent, a mutation that is applied more than once could result in a failure. For example, an insert may fail with `ALREADY_EXISTS` (/spanner/docs/reference/rpc/google.rpc#google.rpc.Code) even though the row did not exist prior to the write attempt. This could occur if the backend server committed the mutation but was unable to communicate the success to the client. In that event, the mutation could be retried, resulting in the `ALREADY_EXISTS` failure.

Here are possible ways to address this scenario when you implement your own client library or use the REST API:

- Structure your writes to be idempotent.
- Use writes with replay protection.
- Implement a method that performs "upsert" logic: insert if new or update if exists.
- Handle the error on behalf of the client.

For best performance, the connection that you use to host a session should remain stable. When the connection that hosts a session changes, Cloud Spanner might abort the active transaction on the session and cause a small amount of extra load on your database while it updates the session metadata. It is OK if a few connections change sporadically, but you should avoid situations that

would change a large number of connections at the same time. If you use a proxy between the client and Cloud Spanner, you should maintain connection stability for each session.

You can use the ListSessions command to monitor active sessions in your database from the command line (/spanner/docs/gcloud-spanner#manage_sessions), with the REST API (/spanner/docs/reference/rest/v1/projects.instances.databases.sessions/list), or with the RPC API (/spanner/docs/reference/rpc/google.spanner.v1#google.spanner.v1.Spanner.ListSessions). ListSessions shows the active sessions for a given database. This is useful if you need to find the cause of a session leak. (A session leak is an incident where sessions are being created but not returned to a session cache for reuse.)

ListSessions allows you to view metadata about your active sessions, including when a session was created and when a session was last used. Analyzing this data will point you in the right direction when troubleshooting sessions. If most active sessions don't have a recent `approximate_last_use_time`, this could indicate that sessions aren't being reused properly by your application. See the RPC API reference (/spanner/docs/reference/rpc/google.spanner.v1#google.spanner.v1.Session) for more information about the `approximate_last_use_time` field.

See the REST API reference (/spanner/docs/reference/rest/v1/projects.instances.databases.sessions/list), the RPC API reference (/spanner/docs/reference/rpc/google.spanner.v1#google.spanner.v1.Spanner.ListSessions), or the gcloud command-line tool reference (/spanner/docs/gcloud-spanner#manage_sessions) for more information on using ListSessions.

For most client libraries, Cloud Spanner reserves a portion of the sessions for read-write transactions, called the write-sessions fraction. If your app uses up all the read sessions, then Cloud Spanner uses the read-write sessions, even for read-only transactions. Read-write sessions require `spanner.databases.beginOrRollbackReadWriteTransaction`. If the user is in the spanner.databaseReader (/spanner/docs/iam#roles) IAM role, then the call fails and Cloud Spanner returns this error message:

For the client libraries that maintain a write-sessions fraction, you can set the write-sessions fraction.