As described in Query execution plans (/spanner/docs/query-execution-plans), Cloud Spanner's SQL compiler transforms a SQL statement into a query execution plan, which is used to obtain the results of the query. This page describes best practices for constructing SQL statements to help Cloud Spanner find efficient execution plans.

The example SQL statements shown in this page use the sample schema below:

For the complete SQL reference, refer to Statement syntax (/spanner/docs/query-syntax), Functions and operators (/spanner/docs/functions-and-operators), and Lexical structure and syntax (/spanner/docs/lexical).

Parameterized queries are a technique of query execution that separates a query string from query parameter values. For example, suppose your application needs to retrieve singers who have released albums with certain titles in a given year. You might write a SQL statement like the following example to retrieve all of the albums titled "Love" that were released in 2017:

In another query, you might change the value of the album title to "Peace":

If your application needs to execute many queries that are similar to this one, in which only a literal value changes in subsequent queries, you should use a parameter placeholder for that value. The resulting parametric query can be cached and reused, which reduces compilation costs.

For example, the rewritten query below replaces `Love` with a parameter named `title`:

Notes about query parameter usage:

- A parameter reference in the query uses the `@` character followed by the parameter name, which can contain any combination of letters, numbers, and underscores.
- Parameters can appear anywhere that a literal value is expected.
- The same parameter name can be used more than once in a single SQL statement.
- Specify the query parameter and the value to bind it to in the `params` field of the `ExecuteSQL` or `ExecuteStreamingSQL` request API (/spanner/docs/reference/rpc/google.spanner.v1#google.spanner.v1.ExecuteSqlRequest).
- Learn more about query parameter syntax in SQL Lexical Structure and Syntax (/spanner/docs/lexical#query-parameters).

The rewritten query above does not use a placeholder for the date value **`2017-01-01`**, because that value is constan
quent query invocations. Leaving the constant as a literal in this case is beneficial because it can improve the specific
plan that's chosen.

In summary, query parameters benefit query execution in the following ways:

- Pre-optimized plans: Queries that use parameters can be executed faster on each invocation because the parameterization makes it easier for Cloud Spanner to cache the execution plan.

- Simplified query composition: you do not need to escape string values when providing them in query parameters. Query parameters also reduce the risk of syntax errors.
- Security: query parameters make your queries more secure by protecting you from various SQL injection attacks. This protection is especially important for queries that you construct from user input.

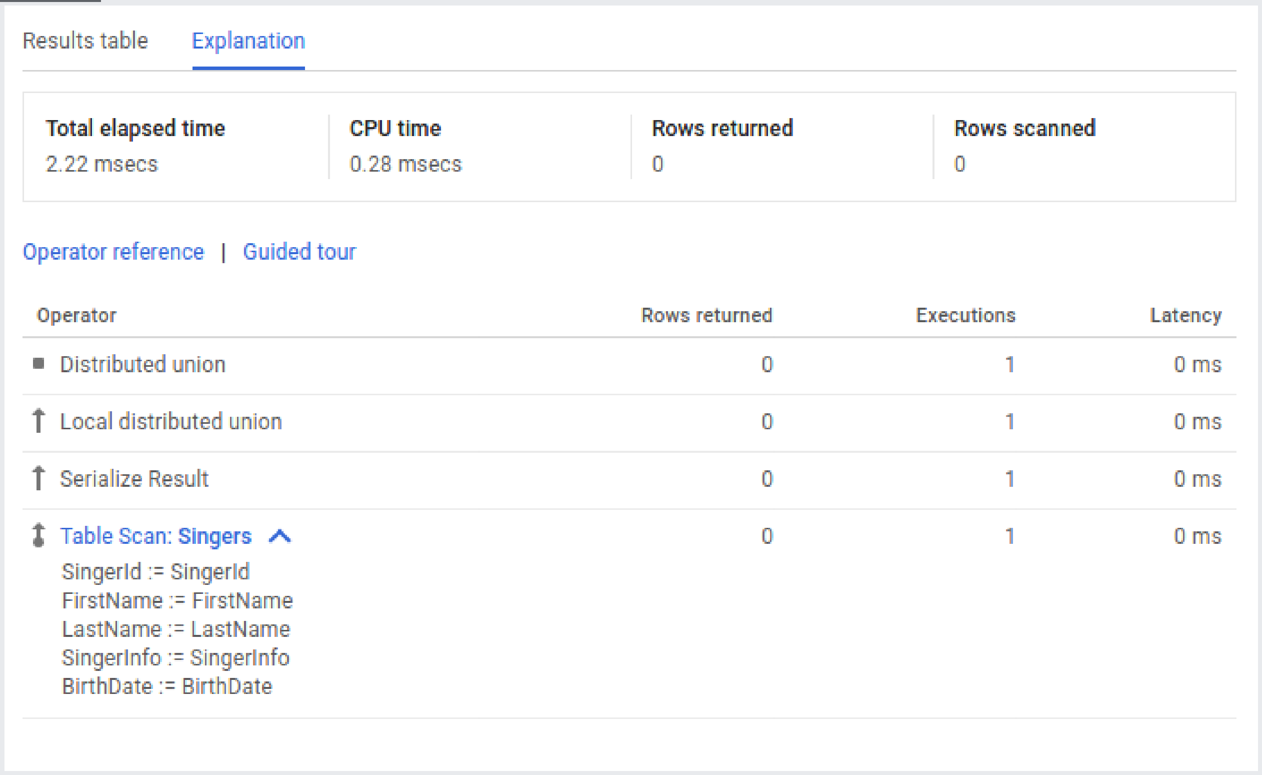
Cloud Spanner enables you to query databases using declarative SQL statements that specify what data you want to retrieve. If you want to also understand how Cloud Spanner obtains the results, you should use query execution plans. A query execution plan displays the cost associated with each step of the query. Using those costs, you can debug query performance issues and optimize your query.

You can retrieve query execution plans through the Cloud Console or the client libraries (/spanner/docs/reference/libraries).

To get a query plan using the Cloud Console:

1. Open the Cloud Spanner instances page.

   Go to Cloud Spanner instances (https://console.cloud.google.com/spanner/instances)
2. Click the name of the Cloud Spanner instance and database you want to query.
3. Click **Query**.
4. Type the query in the text field, then click **Run query**.
5. Click **Explanation**.
   The Cloud Console displays a visual execution plan for your query:



Results table | Explanation

Total elapsed time	CPU time	Rows returned	Rows scanned
2.22 msecs	0.28 msecs	0	0

Operator reference | Guided tour

Operator	Rows returned	Executions	Latency
Distributed union	0	1	0 ms
Local distributed union	0	1	0 ms
Serialize Result	0	1	0 ms

Table Scan: Singers
SingerId := SingerId
FirstName := FirstName
LastName := LastName
SingerInfo := SingerInfo
BirthDate := BirthDate
	0	1	0 ms

For the complete query plan reference, refer to Query execution plans (/spanner/docs/query-execution-plans).

Like other relational databases, Cloud Spanner offers secondary indexes, which you can use to retrieve data using either a SQL statement or using Cloud Spanner's read interface. The more common way to fetch data from an index is to use the SQL query interface. Using a secondary index in a SQL query enables you to specify *how* you want Cloud Spanner to obtain the results. Specifying a secondary index can speed up query execution.

For example, suppose you wanted to fetch the IDs of all the singers with a particular last name. One way to write such a SQL query is:

This query would return the results that you expect, but it might take a long time to return the results. The timing would depend on the number of rows in the `Singers` table and how many satisfy the predicate `WHERE s.LastName = 'Smith'`. If there is no secondary index that contains the `LastName` column to read from, the query plan would read the entire `Singers` table to find rows that match the predicate. Reading the entire table is called a *full table scan*, and a full table scan is an expensive way to obtain the results if the table contains only a small percentage of `Singers` with that last name.

You can improve the performance of this query by defining a secondary index on the last name column:

Because the secondary index `SingersByLastName` contains the indexed table column `LastName` and the primary key column `SingerId`, Cloud Spanner can fetch all the data from the much smaller index table instead of scanning the full `Singers` table.

In this scenario, Cloud Spanner would likely automatically use the secondary index `SingersByLastName` when executing the query. However, it is best to explicitly tell Cloud Spanner to use that index by specifying an index directive (/spanner/docs/secondary-indexes#index_directive) in the `FROM` clause:

Now suppose you also wanted to fetch the singer's first name in addition to the ID. Even though the `FirstName` column is not contained in the index, you should still specify the index directive as before:

You still get a performance benefit from using the index because Cloud Spanner doesn't need to do a full table scan when executing the query plan. Instead, it selects the subset of rows that satisfy the predicate from the `SingersByLastName` index, then does a lookup from the base table `Singers` to fetch the first name for only that subset of rows.

If you want to avoid Cloud Spanner from having to fetch any rows from the base table at all, you can optionally store a copy of the `FirstName` column in the index itself:

Using a `STORING` (/spanner/docs/secondary-indexes#storing_clause) clause like this costs extra storage but it provides the following advantages for queries and read calls using the index:

- 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.

The preceding examples illustrate how secondary indexes can speed up queries when the rows chosen by the `WHERE` clause of a query can be quickly identified using the secondary index. Another scenario in which secondary indexes can offer performance benefits is for certain queries that return ordered results. For example, suppose you wanted to fetch all album titles and their release dates and return them in ascending order of release date and descending order by album title. You could write a SQL query like this:

Without a secondary index, this query requires a potentially expensive sorting step in the execution plan. You could speed up query execution by defining this secondary index:

Then rewrite the query to use the secondary index:

Note that this query and index definition meet both of the following criteria:

- The column list in the `ORDER BY` clause is a prefix of the index key list.
- All columns in the table used in the query are covered by the index.

Because both of these conditions are satisfied, the resulting query plan removes the sorting step and executes faster.

While secondary indexes can speed up common queries, be aware that adding secondary indexes can add latency to your commit operations, because each secondary index typically requires involving an extra node in each commit. For most workloads, having a few secondary indexes is fine. However, you should consider whether you care more about read or write latency, and consider which operations are most critical for your workload. You should also benchmark your workload to ensure that it is performing as you expect.

For the complete reference on secondary indexes, refer to Secondary indexes (/spanner/docs/secondary-indexes).

A common use of SQL query is to read multiple rows from Cloud Spanner based on a list of known keys.

These are the best practices for writing efficient queries when fetching data by a range of keys:

- If the list of keys is sparse and not adjacent, use query parameters and `UNNEST` to construct your query.

  For example, if your key list is `{1, 5, 1000}`, write the query like this:

  Notes:

  - The array UNNEST (/spanner/docs/query-syntax#unnest) operator flattens an input array into rows of elements.
  - `@KeyList` is a query parameter, which can speed up your query as discussed in the preceding best practice (#use_query_parameters_to_speed_up_frequently_executed_queries).

- If the list of keys is adjacent and within a range, specify the lower bound and higher bound of the key range in the `WHERE` clause.

  For example, if your key list is `{1,2,3,4,5}`, construct the query like this:

Where `@min` and `@max` are query parameters that are bound to the values 1 and 5, respectively.

Note that this query is only more efficient if the keys in the key range are adjacent. In other words, if your key list is `{1, 5, 1000}`, you should not specify the lower and higher bounds like in the preceding query because the resulting query would scan through every value between 1 and 1000.

Join operations can be expensive. This is because `JOIN`s can significantly increase the number of rows your query needs to scan, which results in slower queries. In addition to the techniques you're accustomed to using in other relational databases to optimize join queries, here are some best practices for a more efficient JOIN when using Cloud Spanner SQL:

- If possible, join data in interleaved tables by primary key. For example:

  The rows in the interleaved table `Albums` are guaranteed to be physically stored in the same splits as the parent row in `Singers`, as discussed in Schema and Data Model (/spanner/docs/schema-and-data-model). Therefore, `JOIN`s can be completed locally without sending lots of data across the network.

- Use the join directive if you want to force the order of the `JOIN`. For example:

  The join directive `@{FORCE_JOIN_ORDER=TRUE}` tells Cloud Spanner to use the join order specified in the query (that is, `Singers JOIN Albums`, not `Albums JOIN Singers`). The returned results are the same regardless of the order that Cloud Spanner chooses. However, you might want to use this join directive if you notice in the query plan that Cloud Spanner has changed the join order and caused undesirable results like larger intermediate results or has missed opportunities for seeking rows.

- Use a join directive to choose a `JOIN type`. Choosing the right join algorithm for your query can improve latency, memory consumption, or both. This query demonstrates the syntax for using a JOIN directive (/spanner/docs/query-syntax#join-types) to choose a `HASH JOIN`:

- If you're using a `HASH JOIN` or `APPLY JOIN` and if you have a `WHERE` clause that is highly selective on one side of your `JOIN`, put the table that produces the smallest number of rows as the first table in the `FROM` clause of the join. This is because currently in `HASH JOIN`, Cloud Spanner always picks the left-hand side table as build and the right-hand side table as probe. Similarly, for `APPLY JOIN`, Cloud Spanner picks left-hand side as outer and right-hand side table as inner. See more information about these join types: Hash join (/spanner/docs/query-execution-operators#hash-join) and Apply join (/spanner/docs/query-execution-operators#cross-apply).

Read-write transactions (/spanner/docs/transactions#read-write_transactions) allow a sequence of zero or more reads or SQL queries, and can include a set of mutations, before a call to commit. In order to maintain the consistency of your data, Cloud Spanner acquires locks when reading and writing rows in your tables and indexes (read more details about locking in Life of Reads and Writes (/spanner/docs/whitepapers/life-of-reads-and-writes)).

Because of the way locking works in Cloud Spanner, performing a read or SQL query that reads a large number of rows (for example `SELECT * FROM Singers`) means that no other transactions can write to the rows you have read until your transaction is either committed or aborted. Furthermore, because your transaction is processing a large number of rows, it is likely to take longer than a transaction that reads a much smaller range of rows (for example `SELECT LastName FROM Singers WHERE SingerId = 7`), which further exacerbates the problem and reduces system throughput.

Hence, you should try to avoid large reads (for example: full table scans or massive join operations) inside of your transactions, unless you are willing to accept lower write throughput. In some cases, the following pattern can yield better results:

1. Do your large read inside a read-only transaction (/spanner/docs/transactions#read-only_transactions). (Note that read-only transactions do not use locks and hence allow for higher aggregate throughput.)
2. [Optional] If you need to do any processing on the data you just read, do it.
3. Start a read-write transaction.
4. Verify that the critical rows you care about have not changed values since the time that you performed the read-only transaction in step 1.
    a. If the rows have changed, roll back your transaction and start again at step 1.
    b. If everything looks okay, commit your mutations.

One way to ensure that you are avoiding large reads inside of read-write transactions is to look at the execution plans that are generated by your queries.

If you are expecting a certain ordering for the results of a `SELECT` query, you should explicitly include the `ORDER BY` clause. For example: If you want to list all singers in primary key order, use this query:

Note that Cloud Spanner only guarantees result ordering if the `ORDER BY` clause is present in the query. In other words, consider this query without the `ORDER BY`:

Cloud Spanner does not guarantee that the results of this query will be in primary key order. Furthermore, the ordering of results could change at any time and is not guaranteed to be consistent from invocation to invocation.

Because Cloud Spanner does not evaluate parameterized `LIKE` patterns until execution time, Cloud Spanner must read all rows and evaluate them against the `LIKE` expression to filter out rows that do not match.

In cases where a `LIKE` pattern looks for matches that are at the beginning of a value and the column is indexed, use `STARTS_WITH` instead of `LIKE`. This allows Cloud Spanner to more effectively optimize the query execution plan.

Not recommended:

This example assumes **`@like_clause`** is bound to **`'Love%'`**.

Recommended:

This example assumes **`@prefix`** is bound to **`'Love'`**. This query is more efficient than the previous query, and it will r
if there is an index defined on **`Albums.AlbumTitle`**.