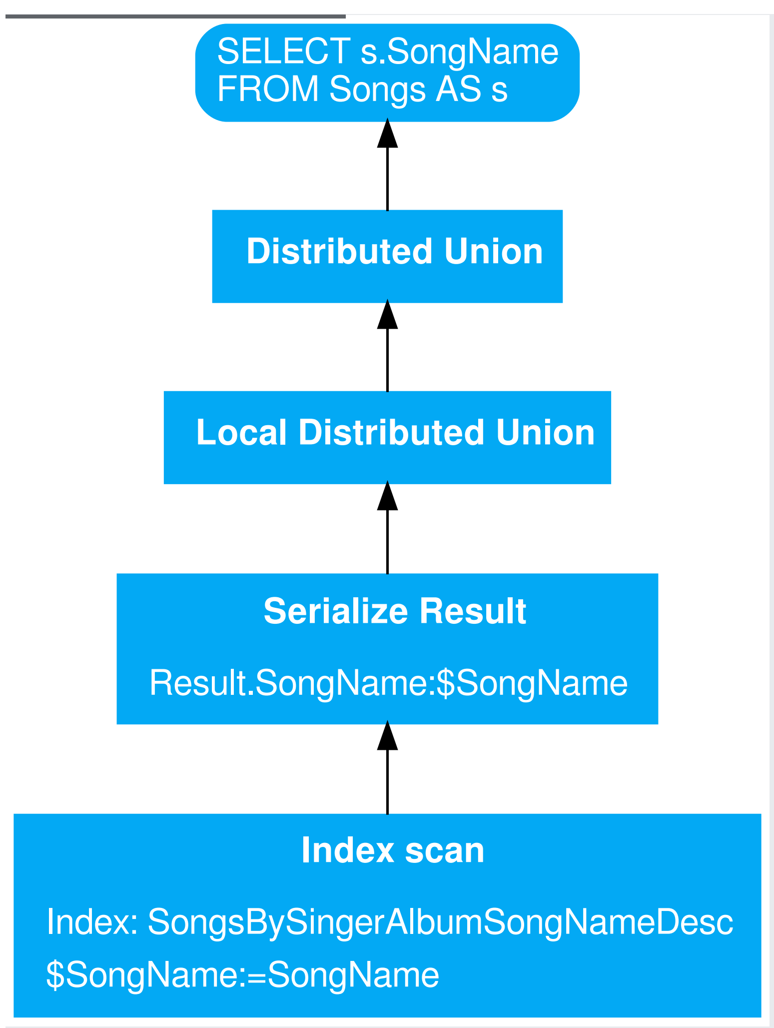
This page provides concepts about query execution plans and how they are used by Cloud Spanner to perform queries in a distributed environment. To learn how to retrieve an execution plan for a specific query using the Cloud Console, see Understanding how Cloud Spanner executes queries (/spanner/docs/sql-best-practices#how-execute-queries).

Cloud Spanner uses declarative SQL statements to query its databases. SQL statements define *what* the user wants without specifying *how* to obtain the results. A *query execution plan* is the set of steps for how the results are obtained. For a given SQL statement, there may be multiple ways to obtain the results. The Cloud Spanner query compiler evaluates the different ways to produce a query execution plan that is considered the most efficient. Cloud Spanner then uses the execution plan to retrieve the results.

Conceptually, an execution plan is a tree of relational operators. Each operator reads rows from its input(s) and produces output rows. The result of the operator at the root of the execution is returned as the result of the SQL query.

As an example, this query:

results in a query execution plan that can be visualized as:



The queries and execution plans on this page are based on the following database schema:

You can use the following Data Manipulation Language (DML) statements to add data to these tables:

You can run queries and retrieve execution plans even if the tables have no data.

Obtaining efficient execution plans is challenging because Cloud Spanner divides data into *splits* (/spanner/docs/schema-and-data-model#database-splits). Splits can move independently from each other and get assigned to different servers, which could be in different physical locations. To evaluate execution plans over the distributed data, Cloud Spanner uses execution based on:

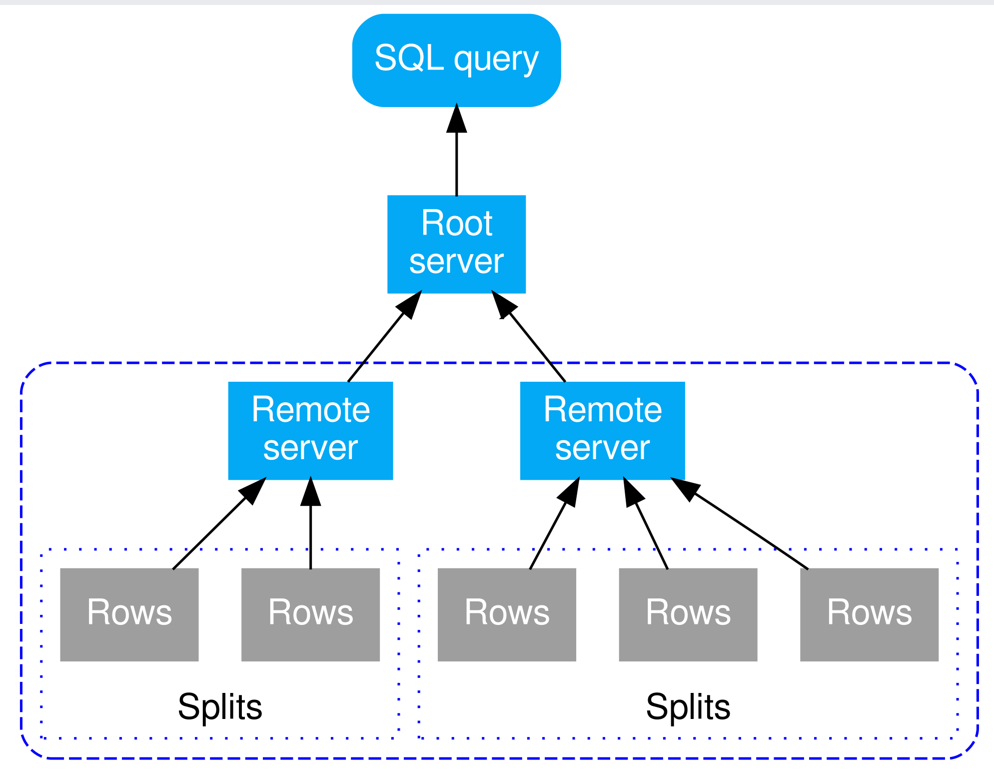
- local execution of *subplans* in servers that contain the data
- orchestration and aggregation of multiple remote executions with aggressive distribution pruning

Cloud Spanner uses the primitive operator `distributed union` (/spanner/docs/query-execution-operators#distributed-union), along with its variants `distributed cross apply` (/spanner/docs/query-execution-operators#distributed-cross-apply) and `distributed outer apply` (/spanner/docs/query-execution-operators#distributed-outer-apply), to enable this model.

A SQL query in Cloud Spanner is first compiled into an execution plan, then it is sent to an initial *root* server for execution. The root server is chosen so as to minimize the number of hops to reach the data being queried. The root server then:

- initiates remote execution of subplans (if necessary)
- waits for results from the remote executions
- handles any remaining local execution steps such as aggregating results
- returns results for the query

Remote servers that receive a subplan act as a "root" server for their subplan, following the same model as the topmost root server. The result is a tree of remote executions. Conceptually, query execution flows from top to bottom, and query results are returned from bottom to top.The following diagram shows this pattern:



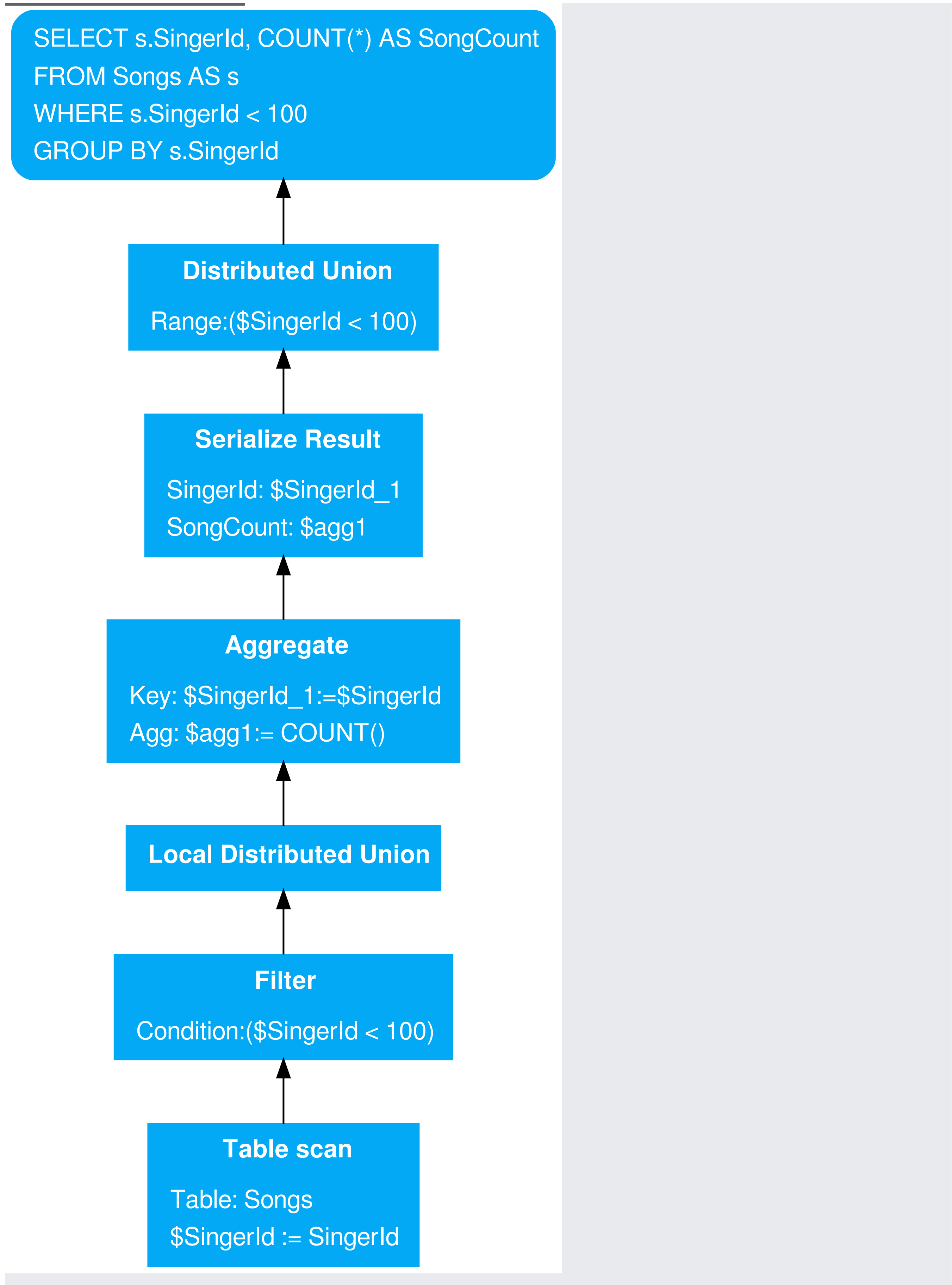
The following examples illustrate this pattern in more detail.

An aggregate query implements `GROUP BY` queries.

For example, using this query:

These are the results:

Conceptually, this is the execution plan:


SELECT s.SingerId, COUNT(*) AS SongCount
FROM Songs AS s
WHERE s.SingerId < 100
GROUP BY s.SingerId
Distributed Union
Range:($SingerId < 100)
Serialize Result
SingerId: $SingerId_1
SongCount: $agg1
Aggregate
Key: $SingerId_1:=$SingerId
Agg: $agg1:= COUNT()
Local Distributed Union
Filter
Condition:($SingerId < 100)
Table scan
Table: Songs
$SingerId := SingerId

Cloud Spanner sends the execution plan to a root server that coordinates the query execution and performs the remote distribution of subplans.

This execution plan starts with a distributed union (/spanner/docs/query-execution-operators#distributed-union), which distributes subplans to remote servers whose splits satisfy `SingerId < 100`. Local distributed unions, shown later in the plan, represent execution on the remote servers. Each local distributed union evaluates a subquery independently on splits of the `Songs` table, subject to the filter `SingerId < 100`. The local distributed unions return results to an aggregate (/spanner/docs/query-execution-operators#aggregate) operator. The aggregate operator performs the `COUNT` aggregation by `SingerId` and returns results to a serialize result (/spanner/docs/query-execution-operators#serialize_result) operator. The serialize result operator serializes the results into rows that contain the song count by `SingerId`. The distributed union then unions all results together and returns the query results.

You can learn more about aggregates at aggregate operator (/spanner/docs/query-execution-operators#aggregate).

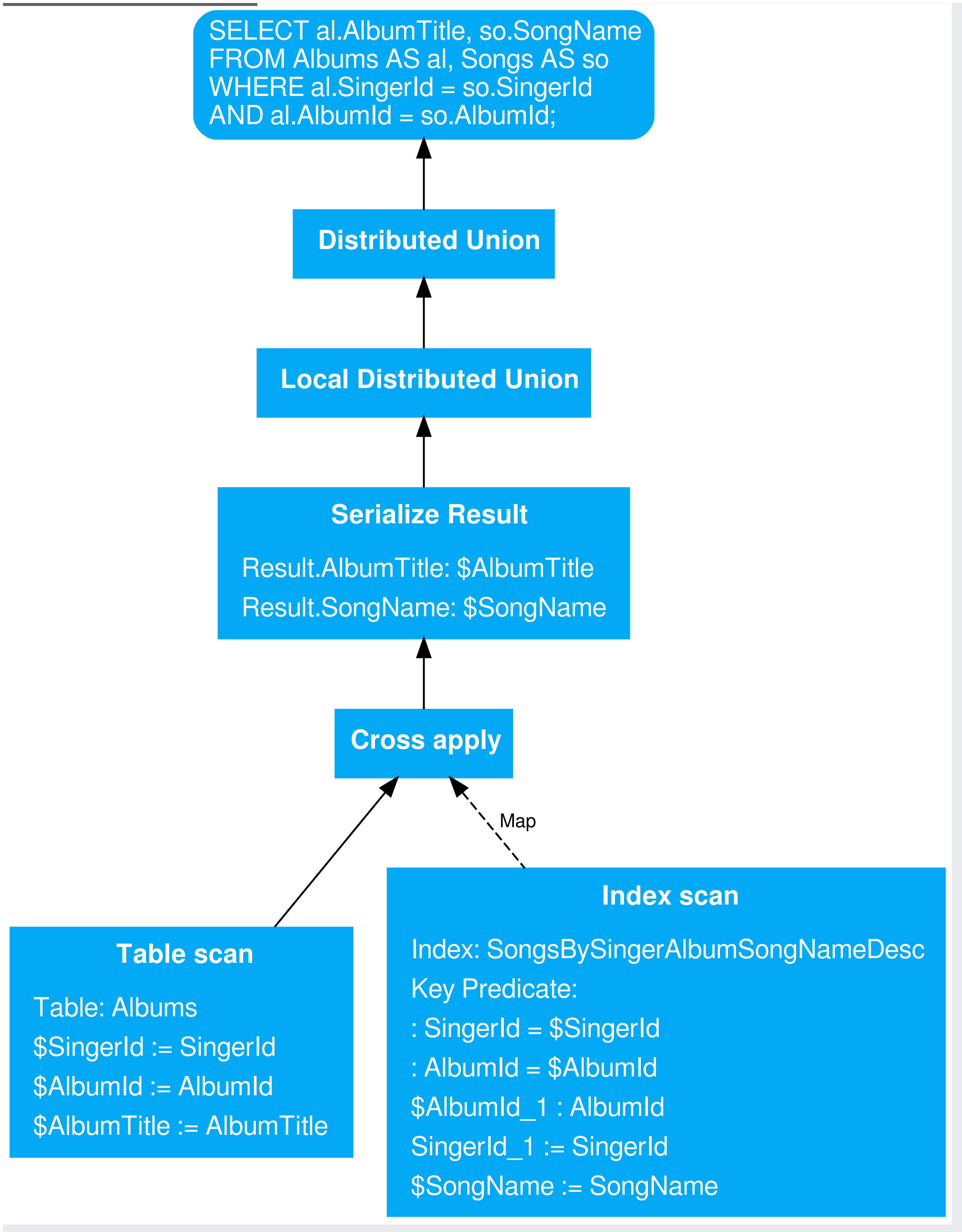
Interleaved (/spanner/docs/schema-and-data-model#creating-interleaved-tables) tables are physically stored with their rows of related tables co-located. A *co-located join* is a join between interleaved tables. Co-located joins can offer performance benefits over joins that require indexes or back joins.

For example, using this query:

(This query assumes that `Songs` is interleaved in `Albums`.)

These are the results:

This is the execution plan:



This execution plan starts with a distributed union (/spanner/docs/query-execution-operators#distributed-union), which distributes subplans to remote servers that have splits of the table `Albums`. Because `Songs` is an interleaved table of `Albums`, each remote

server is able to execute the entire subplan on each remote server without requiring a join to a different server.

The subplans contain a cross apply (/spanner/docs/query-execution-operators#cross-apply). Each cross apply performs a table scan (/spanner/docs/query-execution-operators#scan) on table `Albums` to retrieve `SingerId`, `AlbumId`, and `AlbumTitle`. The cross apply then maps output from the table scan to output from an index scan on index `SongsBySingerAlbumSongNameDesc`, subject to a filter (/spanner/docs/query-execution-operators#filter) of the `SingerId` in the index matching the `SingerId` from the table scan output. Each cross apply sends its results to a serialize result (/spanner/docs/query-execution-operators#serialize_result) operator which serializes the `AlbumTitle` and `SongName` data and returns results to the local distributed unions. The distributed union aggregates results from the local distributed unions and returns them as the query result.

The example above used a join on two tables, one interleaved in the other. Execution plans are more complex and less efficient when two tables, or a table and an index, are not interleaved.

Consider an index created with the following command:

Use this index in this query:

These are the results:

This is the execution plan:

Back join query execution plan

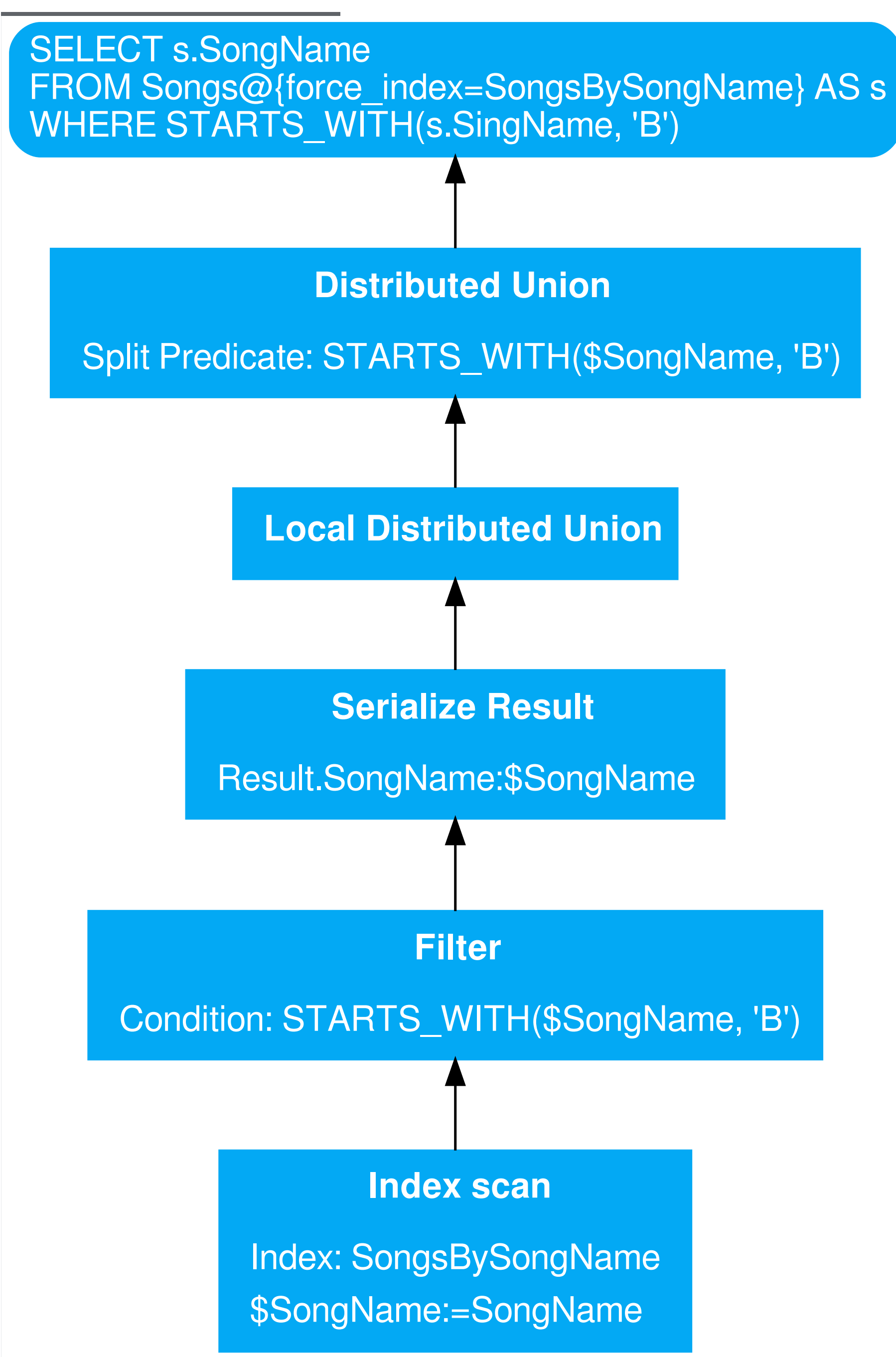
The resulting execution plan is complicated because the index `SongsBySongName` does not contain column `Duration`. To obtain the `Duration` value, Cloud Spanner needs to *back join* the indexed results to the table `Songs`. This is a join but it is not co-located because the `Songs` table and the global index `SongsBySongName` are not interleaved. The resulting execution plan is more complex than the co-located join example because Cloud Spanner performs optimizations to speed up the execution if data isn't co-located.

The top operator is a distributed cross apply (/spanner/docs/query-execution-operators#distributed-cross-apply). This input side of this operator are batches of rows from the index `SongsBySongName` that satisfy the predicate `STARTS_WITH(s.SongName, "B")`. The distributed cross apply then maps these batches to remote servers whose splits contain the `Duration` data. The remote servers use a table scan to retrieve the `Duration` column. The table scan uses the filter `Condition:($Songs_key_TrackId' = $batched_Songs_key_TrackId)`, which joins `TrackId` from the `Songs` table to `TrackId` of the rows that were batched from the index `SongsBySongName`.

The results are aggregated into the final query answer. In turn, the input side of the distributed cross apply contains a distributed union/local distributed union pair to evaluate rows from the index that satisfy the `STARTS_WITH` predicate.

Consider a slightly different query that doesn't select the `s.Duration` column:

This query is able to fully leverage the index as shown in this execution plan:



The execution plan doesn't require a back join because all the columns requested by the query are present in the index.

- Learn about Query execution operators (/spanner/docs/query-execution-operators)