Cloud Spanner's Table Interleaving: A Query Optimization Feature

By Christoph Bussler and Anand Jain

Christoph Bussler
Apr 16 · 11 min read

This blog discusses Cloud Spanner table interleaving from a developer's perspective: what are interleaving tables, when to use this schema design concept, some edge cases and gotchas.

tl;dr: interleaving tables are a query performance optimization tool for specific join query patterns. Interleaving tables are not a data modeling concept even though they appear as such at first glance during schema design. Interleaved indexes fall into the category of performance optimization as well.

Cloud Spanner's use of distributed persistent data management

Cloud Spanner is Google Cloud's "fully managed relational database with unlimited scale, strong consistency, and up to 99.999% availability."

One underlying fundamental design decision to achieve unlimited scale as a relational database is Cloud Spanner's reliance on distributed persistent data storage in combination with distributed query processing: queries are executed across distributed processing nodes accessing distributed persistent storage.

The storage underlying Cloud Spanner can scale "forever" since it allows adding capacity without architectural or implementation imposed limitations. The unit of capacity that can be added (or removed) is 2 TB and it is accomplished by adding (or removing) Cloud Spanner nodes to a Cloud Spanner instance. To illustrate the possible scale, this presentation characterizes a Cloud Spanner instance with PBs of relational data. Data is

stored in databases and one Cloud Spanner instance can manage several databases (currently up to 100 databases).

The distributed nature of storage as well as table sizes and possible hotspots results in automatic horizontal partitioning of tables by primary key. A table must have a primary key and the rows of a table are sorted by primary key order on storage. A horizontal partition is called a split and a split is a subset of the rows of a table. Load-based splitting is one of the processes that Cloud Spanner executes in the background.

Splits of a table might be stored in the same storage area, or in different storage areas. When a table grows the splits will be in different storage locations, and if capacity is added more storage locations will be available for a database to store splits. The safe assumption is that a table's splits are stored in different locations in the general case.

If data is queried from a single table that resides in different splits, the query execution will have to find and to access the storage locations that store the relevant splits and return the data. The data might be in the same split, in different splits, and in the same or different storage locations. If the data is in different storage locations then distributed storage access takes place.

The same is true for a join between two tables: several splits and several storage locations of each table can be involved. Accessing different storage locations incurs more latency than a single storage location because of their distribution.

Wouldn't it be nice to have a way to force data collocation when certain rows of different tables are always queried together based on the application's query patterns? For example, in a parent-child 1:n relationship it might be that the parent row is always or most of the time joined with its children. In this case having both in the same split and the same storage location would be the best storage layout as the result data set would not have to be gathered from distributed storage locations.

This optimization, collocating rows of parent-child tables, is available in Cloud Spanner and called table interleaving.

The reminder of this blog discusses this runtime optimization feature, its relationship to schema design, some of the best practices, caveats and anti-patterns.

Schema feature: table interleaving

The concept of table interleaving is expressed as a syntax terminal `INTERLEAVE IN PARENT` when specifying tables. For example,

```
CREATE TABLE Singers (
  SingerId INT64 NOT NULL,
  FirstName STRING(1024),
  LastName STRING(1024),
  SingerInfo BYTES(MAX),
  ) PRIMARY KEY (SingerId);

CREATE TABLE Albums (
  SingerId INT64 NOT NULL,
  AlbumId INT64 NOT NULL,
  AlbumTitle STRING(MAX),
  ) PRIMARY KEY (SingerId, AlbumId),
    INTERLEAVE IN PARENT Singers ON DELETE CASCADE;
```

In the above example, the table `Albums` is interleaved in the table `Singers`.

What does this mean? Before going into detail, observe that the primary key of `Singers` is a leading part of the primary key of `Albums`. This establishes a 1:n relationship between a parent table (here `Singers)` with the child table (here `Albums`).

Cloud Spanner based on this specification and the value of the primary keys knows which children rows (might be zero, one or more) belong to which parent row in the 1:n relationship. This now directs the storage management subsystem to collocate the children rows of a parent row into the same split, and hence into the same storage location. A parent row and its children row are stored in the same location without any distribution. Two different parent rows and their respective children might reside in different splits and in different storage locations; but a parent and its children will be in the same split.

Here is a diagram illustrating this layout:

Singers(1)	"Marc"	"Richards"	<Bytes>	
Albums(1, 1)				"Total Junk"
Albums(1, 2)				"Go, Go, Go"

Albums(1, 2)				Go, Go, Go
Singers(2)	"Catalina"	"Smith"	<Bytes>	
Albums(2, 1)				"Green"
Albums(2, 2)				"Forever Hold Your Peace"
Albums(2, 3)				"Terrified"

A query retrieving the children rows for each parent in the above example looks like

```
SELECT s.FirstName,
       s.LastName,
       a.AlbumTitle
FROM Singers s JOIN Albums a ON s.SingerId = a.SingerId;
```

This query returns one row for each child row of a parent row.

As you can see, the fact that the tables are interleaved is not visible in the query itself. If the two tables were not declared on a schema level to be interleaved but would be independent tables, the query would look exactly the same. This observation will be important later.

When the above query executes, it finds all children rows of a parent collocated with the parent and the query execution will not have to access different distributed storage locations. This is the performance benefit: a join on collocated data in one storage location.

This benefit is not only available for one level of relationship, but up to 7 levels, meaning, each child row can be a parent to further children rows (grand-children) in another table.

An alternative query to retrieve the children of a parent row involves the data type `ARRAY` as follows

```
SELECT s.FirstName,
       s.LastName,
       ARRAY(SELECT AS STRUCT a1.AlbumTitle
             FROM Singers s1
```

```
                    JOIN Albums a1 ON s1.SingerId = a1.SingerId
               WHERE s.SingerId = s1.SingerId) Albums
FROM Singers s
WHERE EXISTS(SELECT a2.AlbumTitle
               FROM Albums a2
               WHERE a2.SingerId = s.SingerId);
```

This query returns one row for each singer and all the singer's albums in an array. The exists clause ensures that only those singers are returned that have an album. It demonstrates that a 1:n relationship can be queried and the child table entries represented as an array in case this is advantageous for the client application.

Alternative to table interleaving

Table interleaving forces the collocation of a parent and its children rows into the same storage location. An alternative design is the use of an array. Instead of table interleaving the children are stored in a column within the parent row. For example,

```
CREATE TABLE SingersAndAlbums (
  SingerId INT64 NOT NULL,
  FirstName STRING(1024),
  LastName STRING(1024),
  SingerInfo BYTES(MAX),
  Albums ARRAY<STRING(MAX)>,
  ) PRIMARY KEY (SingerId);
```

This design accomplishes almost the same goal from a storage perspective, however, there are not only advantages, but also disadvantages:

Some advantages are

- **Storage collocation**. Since the albums are stored as an array in a column they are automatically collocated on storage with the singer's data.
- **Convenient querying**. Selecting the albums of a singer is a non-join select statement.

Some disadvantages are

- **Array elements are scalar.** Array elements can only be scalar types. For the use case above, the album's id and title would have to be stored in a single string (with some delimiter or as a JSON object).
- **Array manipulation**. Array manipulation has specific functions and SQL syntax, as explained here.
- **Column value size limit 10 MB**. The limit of a column value size is 10 MB. If the data in an array can exceed that, an error occurs and an alternate design has to be found.
- **Complete array unit of read access**. Compared to interleaved rows which can be individually selected, an array is always read in its entirety.

While using array as column data types is not in general a disadvantage, for the use case above it would complicate the application logic that manages albums for singers.

Additional benefits of table interleaving

Table interleaving has additional benefits not further discussed below. One benefit is in context of deleting parent rows. It is possible to specify that child rows of a parent are deleted with the parent row automatically. The construct `ON CASCADE DELETE` is described here.

An implicit benefit is realized when inserting parents and their children in the same transaction. If both, a parent and its children are inserted in the same transaction, in most cases the insert happens in the same split. This requires less coordination compared to the case where parent and children rows are managed as separate tables. In this case it is possible that distributed coordination across storage regions is required if the rows are stored in different splits.

Performance optimization

The previous section has shown that a 1:n relationship can be modeled as separate tables as well as interleaved tables, and in both cases a query selecting a parent-child relationship by means of a join is exactly the same. The table interleaving forces collocation of parent and child rows, making joins between a parent and a child table better performing compared to non-interleaved tables as the data is collocated in the case of interleaving tables.

So why not simply use interleaving as much as possible from the very beginning when starting the table design?

Designing the tables of a schema and their foreign key relationships is a conceptual activity that is based on data semantics. Questions to be answered during a schema design are for example: are the tables (entities) correct for my area of interest? Are columns representing the properties of entities appropriately? Do the primary keys uniquely identify entities? Are the foreign key relationships representing dependencies correctly?

While certain query patterns are known during schema design, the specification of the queries is not at the forefront. At some point, however, query specification takes center stage in the design, and at this point optimization steps are taking place, like: defining indexes, denormalizing tables or replacing string data types with enumerations based on integers.

And later, the query execution frequency is taken into consideration in the sense that certain performance optimizations make sense for a subset of all queries, not for every single one of those. And at this point table interleaving becomes a performance improvement tool: will table interleaving be helpful in improving performance by causing collocation of data?

If table interleaving would be addressed during schema design, then it'll take on the role of a conceptual schema modeling concept (1:n, possibly part-of relationship). The consequence of collocating data might move into the background and when tested might cause a performance impediment.

Therefore, the recommendation is to not optimize premature based on expectation, but based on a proof-of-concept and measurement. Therefore, use table interleaving as a performance improvement tool, not as a schema design technique.

Best practice design process

The following process is the general best practice for table interleaving (caveats and edge cases follow afterward):

1. Design the database schema without interleaved tables in an initial schema design phase

2. During the schema design phase, keep in mind that tables representing 1:n relationships might have to be changed to interleaved tables later on for performance reasons. Design the primary keys appropriately if possible so that the primary key of the potential parent table is the leading part of the composite primary key of the potential child table

3. Design all queries (to the extent possible) that will operate on the schema

4. Review all joins and assess their nature. For the purpose of table interleaving, call out the joins that query 1:n relationships. Measure their execution performance and observe the execution frequency for those joins that query 1:n relationships

5. If there are 1:n joins between two tables for about 80+% of the time consider putting those tables into an interleaving relationship. Measure the query execution performance in both schemas (with and without interleaved tables) and determine if table interleaving provides a performance benefit

This best practice design process applies for those queries that are key to the application's core business logic and execute frequently. For a query that is only executed daily, weekly or monthly, it might not be important to consider such an optimization process.

Caveats and edge cases

There are a few caveats and edge cases to keep in mind while using table interleaving in Cloud Spanner:

- When interleaving tables, there is a soft limit of about 8GB for the size of all child rows combined. Up to that point all child rows are collocated with the parent row on the same split. Any additional child rows are stored separately for the next 8GB in another split.

- It is tempting to keep historical data for a parent row as child rows. For example, every change of a parent is added as a separate child row. This is a design that ensures an ever growing number of child rows and will run into the 8GB soft limit at

some point in time possibly slowing down query performance. History data is in general best stored in a separate table.

- Accessing an interleaved table without accessing the parent table will not reap the benefits of storage collocation. Instead a query accessing child rows only might incur a performance penalty as the query might have to access many different splits holding child rows that otherwise might be stored contiguously.
- If only parent rows are accessed without at the same time joining child tables, and the query's predicate not a primary key then the benefit of interleaving might not materialize either since the query will have to access many splits "jumping" over the child rows.
- If after the schema design phase interleaved tables are used, and in the worst case all tables are interleaved with each other in one form or another, then step back and reevaluate if this is really the best design given that the queries are unknown or maybe not known completely at this point in time.

Data size matters

Data volume matters as well and the following use case shows the situation. Assume a parent table that has many columns, and each column stores a large amount of data. In order to emphasize the use case, let's assume 100 columns, and each column stores 1 MB of data. A row therefore stores about 100 MB.

For the child rows, let's assume that each parent has about 25 children in a normal case, and each child has a few columns that in total are up to 1 MB.

A join between a parent row and all its children that retrieves all columns (select *) will result in a result set size of 101 MB * 25, aka, about 2.5 GB. Aside from it being a lot of data in the result set, most of the data will be redundant. In this case retrieving the parent and its children separately will result in two queries, with about 125 MB total result set size. When using two queries the amount of data returned is a lot less compared to executing a join, and no redundant data.

This is just an example of a specific edge case situation where table interleaving is a perfect design, but querying the data by join might not be the best strategy from a performance standpoint.

Summary

In summary, regard table interleaving as a performance improvement tool to be used on the basis of measurement and actual performance numbers for the most critical queries in your application design. Avoid the temptation to use table interleaving as a schema design concept as it might interfere with query performance when not employed based on performance measurements.

Next steps

As next steps, follow the best practice design process at the next opportunity. In the meanwhile, review the product documentation here and here.

Acknowledgements

I'd like to thank Neha Deodhar, Vlad Lifliand and Yuki Furuyama for the thorough review and many comments to improve the accuracy of this content.

Disclaimer

Christoph Bussler is a Solutions Architect and Anand Jain is Data & Analytics Cloud Engineer at Google, Inc. (Google Cloud). The opinions stated here are our own, not those of Google, Inc.

Google Cloud Spanner Table Interleaving Query Optimization