A Cloud Spanner database can contain one or more tables. Tables look like relational database tables in that they are structured with rows, columns, and values, and they contain primary keys. Data in Cloud Spanner is strongly typed: you must define a schema for each database and that schema must specify the data types of each column of each table. Allowable data types include scalar and array types, which are explained in more detail in Data types (/spanner/docs/data-types). You can also define one or more secondary indexes (/spanner/docs/secondary-indexes) on a table.

You can define multiple tables in a database, and **you can optionally define parent-child relationships between tables if you want Cloud Spanner to physically co-locate their rows for efficient retrieval**. For example, if you have a `Customers` table and an `Invoices` table, and your application frequently fetches all the invoices for a given customer, you can define `Invoices` as a child table of `Customers`. In doing so, you're declaring a data locality relationship between two logically independent tables: you're telling Cloud Spanner to physically store one or more rows of `Invoices` with one `Customers` row.

How do you tell Cloud Spanner which `Invoices` rows to store with which `Customers` rows? You do so using the primary key of these tables. Every table must have a primary key, and that primary key can be composed of zero or more columns of that table. If you declare a table to be a child of another table, the primary key column(s) of the parent table must be the prefix of the primary key of the child table. This means if a parent table's primary key is composed of N columns, the primary key of each of its child tables must also be composed of those same N columns, in the same order and starting with the same column.

Cloud Spanner stores rows in sorted order by primary key values, with child rows inserted between parent rows that share the same primary key prefix. This insertion of child rows between parent rows along the primary key dimension is called **interleaving**, and child tables are also called **interleaved tables**. (See an illustration of interleaved rows in the Creating interleaved tables (#creating-interleaved-tables) below.)

In summary, Cloud Spanner can physically co-locate rows of related tables. The schema examples below (#schema-examples) show what this physical layout looks like.

The primary key uniquely identifies each row in a table. If you want to update or delete existing rows in a table, then the table must have a primary key composed of one or more columns. (A table with no primary key columns can have only one row.) Often your application already has a field that's a natural fit for use as the primary key. For example, in the `Customers` table example above, there might be an application-supplied `CustomerId` that serves well as the primary key. In other cases, you may need to generate a primary key when inserting the row, like a unique `INT64` value that you generate.

In all cases, you should be careful not to create hotspots with the choice of your primary key. For example, if you insert records with a monotonically increasing integer as the key, you'll always insert at the end of your key space. This is undesirable because Cloud Spanner divides data among servers by key ranges, which means your inserts will be directed at a single server, creating a hotspot. There are techniques that can spread the load across multiple servers and avoid hotspots:

- Hash the key (/spanner/docs/schema-design#fix_hash_the_key) and store it in a column. Use the hash column (or the hash column and the unique key columns together) as the primary key.
- Swap the order (/spanner/docs/schema-design#fix_swap_key_order) of the columns in the primary key.
- Use a Universally Unique Identifier (UUID) (/spanner/docs/schema-design#uuid_primary_key). Version 4 UUID (https://tools.ietf.org/html/rfc4122) is recommended, because it uses random values in the high-order bits. Don't use a UUID algorithm (such as version 1 UUID) that stores the timestamp in the high order bits.
- Bit-reverse (/spanner/docs/schema-design#bit_reverse_primary_key) sequential values.

You can define hierarchies of parent-child relationships between tables up to seven layers deep, which means you can co-locate rows of seven logically independent tables. If the size of the data in your tables is small, your database can probably be handled by a single Cloud Spanner server. But what happens when your related tables grow and start reaching the resource limits of an individual server? Cloud Spanner is a distributed database, which means that as your database grows, **Cloud Spanner divides your data into chunks called "splits", where individual splits can move independently from each other and get assigned to different servers, which can be in different physical locations**. A split is defined as a range of rows in a top-level (in other words, non-interleaved) table, where the rows are ordered by primary key. The start and end keys of this range are called "split

boundaries". Cloud Spanner automatically adds and removes split boundaries, which changes the number of splits in the database.

Cloud Spanner splits data based on load: it adds split boundaries automatically when it detects high read or write load spread among many keys in a split. You have some control over how your data is split because Cloud Spanner can only draw split boundaries between rows of tables that are at the root of a hierarchy (that is, tables that are not interleaved in a parent table). Additionally, rows of an interleaved table cannot be split from their corresponding row in their parent table because the rows of the interleaved table are stored in sorted primary key order together with the row from their parent table that shares the same primary key prefix. (See an illustration of interleaved rows in Creating a hierarchy of interleaved tables (#creating_a_hierarchy_of_interleaved_tables).) Thus, **the parent-child table relationships that you define, along with the primary key values that you set for rows of related tables, give you control over how data is split under the hood**.

As an example of how Cloud Spanner performs load-based splitting to mitigate read hotspots, suppose your database contains a table with 10 rows that are read more frequently than all of the other rows in the table. As long as that table is at the root of the database hierarchy (in other words, it's not an interleaved table), Cloud Spanner can add split boundaries between each of those 10 rows so that they're each handled by a different server, rather than allowing all the reads of those rows to consume the resources of a single server.

As a general rule, if you follow best practices for schema design (/spanner/docs/schema-design), Cloud Spanner can mitigate hotspots on reads that target rows of a non-interleaved table such that the read throughput should improve every few minutes until you saturate the resources in your instance or run into cases where no new split boundaries can be added (because you have a split that covers just a single row and its interleaved children).

The schema examples below show how to create Cloud Spanner tables with and without parent-child relationships and illustrate the corresponding physical layouts of data.

Suppose you're creating a music application and you need a simple table that stores rows of singer data:

Singers

SingerId	FirstName	LastName	SingerInfo
1	"Marc"	"Richards"	<Bytes>
2	"Catalina"	"Smith"	<Bytes>
3	"Alice"	"Trentor"	<Bytes>
4	"Lea"	"Martin"	<Bytes>
5	"David"	"Lomond"	<Bytes>

(/spanner/docs/images/singers_logical.svg)

Logical view of rows in a simple Singers table. The primary key column appears to the left of the bolded line.

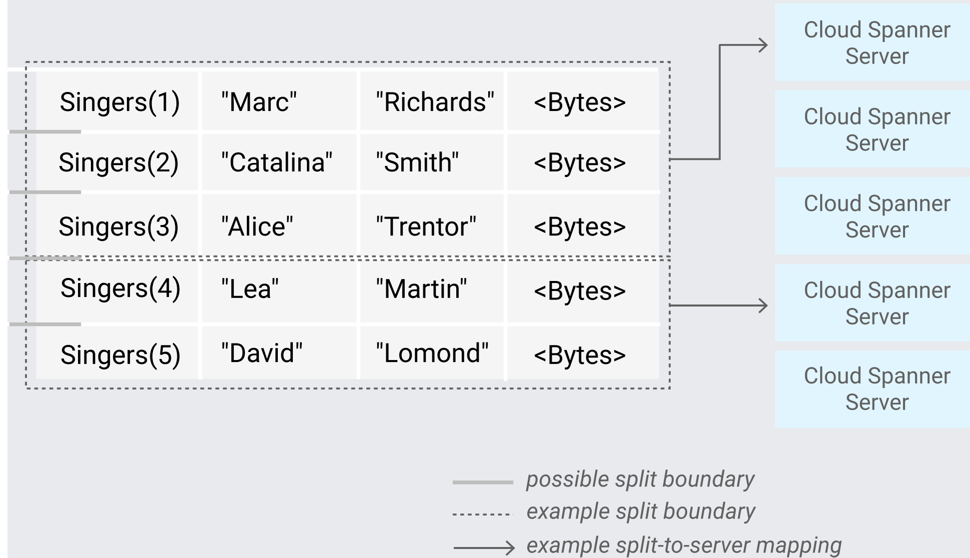
Note that the table contains one primary key column, `SingerId`, which appears to the left of the bolded line, and that tables are organized by rows, columns, and values.

You can define the table with a Cloud Spanner schema like this:

Note the following about the example schema:

- `Singers` is a table at the root of the database hierarchy (because it's not defined as a child of another table).
- Primary key columns are usually annotated with `NOT NULL` (though you can omit this annotation if you want to allow `NULL` values in key columns; see more in Key Columns (#notes_about_key_columns)).
- Columns that are not included in the primary key are called non-key columns, and they can have an optional `NOT NULL` annotation.
- Columns that use the `STRING` or `BYTES` type must be defined with a length, which represents the maximum number of Unicode characters that can be stored in the field. (More details in Scalar Data Types (/spanner/docs/data-definition-language#scalars).)

What does the physical layout of the rows in the `Singers` table look like? The diagram below shows rows of the `Singers` table stored by contiguous (that is, sorted order of) primary key (that is, "Singers(1)", then "Singers(2)", and so on, where "Singers(1)" represents the row in the Singers table keyed by 1).



(/spanner/docs/images/singers_physical.svg)

Physical layout of rows in the Singers table, with an example split boundary that results in splits handled by different servers.

The above diagram also illustrates possible split boundaries, which can occur between any rows of `Singers`, because `Singers` is at the root of the database hierarchy. It also illustrates an example split boundary between the rows keyed by `Singers(3)` and `Singers(4)`, with the data from the resulting splits assigned to different servers. This means that as this table grows, it's possible for rows of `Singers` data to be stored in different locations.

Assume you now want to add some basic data about each singer's albums to the music application:

Albums

SingerId	AlbumId	AlbumTitle
1	1	"Total Junk"
1	2	"Go, Go, Go"
2	1	"Green"
2	2	"Forever Hold Your Peace"
2	3	"Terrified"

(/spanner/docs/images/albums_logical.svg)

Logical view of rows in an Albums table. Primary key columns appear to the left of the bolded line

Note that the primary key of `Albums` is composed of two columns: `SingerId` and `AlbumId`, to associate each album with its singer. The following example schema defines both the `Albums` and `Singers` tables at the root of the database hierarchy, which makes them sibling tables:

The physical layout of the rows of `Singers` and `Albums` looks like the diagram, with rows of the `Albums` table stored by contiguous primary key, then rows of `Singers` stored by contiguous primary key:

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

possible split boundary

(/spanner/docs/images/singers_albums_noninterleaved_physical.svg)

Physical layout of rows of Singers and Albums tables, both at the root of the database hierarchy.

One important note about the schema above is that Cloud Spanner assumes no data locality relationships between the `Singers` and `Albums` tables, because they are top-level tables. As the database grows, Cloud Spanner can add split boundaries between any of the rows shown above. This means the rows of the `Albums` table could end up in a different split from the rows of the `Singers` table, and the two splits could move independently from each other.

Depending on your application's needs, it might be fine to allow `Albums` data to be located on different splits from `Singers` data. However, if your application frequently needs to retrieve information about all the albums for a given singer, then you should create `Albums` as a child table of `Singers`, which co-locates rows from the two tables along the primary key dimension. The next example explains this in more detail.

As you're designing your music application, suppose you realize that the app needs to frequently access rows from both `Singers` and `Albums` tables for a given primary key (e.g. each time you access the row `Singers(1)`, you also need to access the rows `Albums(1, 1)` and `Albums(1, 2)`). In other words, `Singers` and `Albums` need to have a strong data locality relationship.

You can declare this data locality relationship by creating `Albums` as a child, or "interleaved", table of `Singers`. As mentioned in Primary keys (#primary_keys), an **interleaved table** is a table that you declare to be a child of another table because you want the rows of the child table to be physically stored together with the associated parent row. As mentioned above, the prefix of the primary key of a child table must be the primary key of the parent table.

The bolded line in the schema below shows how to create `Albums` as an interleaved table of `Singers`.

Notes about this schema:

- `SingerId`, which is the prefix of the primary key of the child table `Albums`, is also the primary key of its parent table `Singers`. This is not required if `Singers` and `Albums` are at the same level of the hierarchy, but is required in this schema because `Albums` is declared to be a child table of `Singers`.
- The `ON DELETE CASCADE` (/spanner/docs/data-definition-language#create_table) annotation signifies that when a row from the parent table is deleted, its child rows in this table will automatically be deleted as well (that is, all rows that start with the same primary key). If a child table does not have this annotation, or the annotation is `ON DELETE NO ACTION`, then you must delete the child rows before you can delete the parent row.
- Interleaved rows are ordered first by rows of the parent table, then by contiguous rows of the child table that share the parent's primary key, i.e. "Singers(1)", then "Albums(1, 1)", then "Albums(1, 2)", and so on.
- The data locality relationship of each singer and their album data would be preserved if this database splits, because splits can only be inserted between rows of the `Singers` table.
- The parent row must exist before you can insert child rows.

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

possible split boundary

(/spanner/docs/images/singers_albums_interleaved_physical.svg)

Physical layout of rows of Singers and its child table Albums.

The parent-child relationship between `Singers` and `Albums` can be extended to more descendant tables. For example, you could create an interleaved table called `Songs` as a child of `Albums` to store the track list of each album:

Songs

SingerId	AlbumId	TrackId	SongName
1	2	1	"42"
1	2	2	"Nothing Is The Same"
2	1	1	"Let's Get Back Together"
2	1	2	"Starting Again"
2	1	3	"I Knew You Were Magic"
2	3	1	"Fight Story"

(/spanner/docs/images/songs_logical.svg)

Logical view of rows in an Songs table. Primary key columns appear to the left of the bolded line

`Songs` must have a primary key that's composed of all the primary keys of the tables above it in the hierarchy, that is, `SingerId` and `AlbumId`.

The physical view of interleaved rows shows that the data locality relationship is preserved between a singer and their albums and songs data:

Singers(1)	"Marc"	"Richards"	<Bytes>		
Albums(1, 1)				"Total Junk"	
Albums(1, 2)				"Go, Go, Go"	
Songs(1, 2, 1)					"42"
Songs(1, 2, 2)					"Nothing Is The Same"
Singers(2)	"Catalina"	"Smith"	<Bytes>		
Albums(2, 1)				"Green"	
Songs(2, 1, 1)					"Let's Get Back Together"
Songs(2, 1, 2)					"Starting Again"
Songs(2, 1, 3)					"I Knew You Were Magic"
Albums(2, 2)				"Forever Hold Your Peace"	
Albums(2, 3)				"Terrified"	
Songs(2, 3, 1)					"Fight Story"

possible split boundary

(/spanner/docs/images/singers_albums_songs_interleaved_physical.svg)

Physical layout of rows of Singers, Albums, and Songs tables, which form a hierarchy of interleaved tables.

In summary, a parent table along with all of its child and descendant tables forms a hierarchy of tables in the schema. Although each table in the hierarchy is logically independent, physically interleaving them this way can improve performance, effectively pre-joining the tables and allowing you to access related rows together while minimizing disk accesses.

If possible, join data in interleaved tables by primary key. Each interleaved row is guaranteed to be physically stored in the same split as its parent row. Therefore, Cloud Spanner can perform joins by primary key locally, minimizing disk access and network traffic. In the following example, `Singers` and `Albums` are joined on the primary key, `SingerId`:

Interleaving tables in Cloud Spanner is not required, but is recommended for tables with strong data locality relationships. Avoid interleaving tables if there is a chance that the size of a single row and its descendents will become larger than a few GB (/spanner/docs/schema-design#limit_row_size).

The keys of a table can't change; you can't add a key column to an existing table or remove a key column from an existing table.

Primary key columns can be defined to store NULLs. If you would like to store NULLs in a primary key column, omit the `NOT NULL` clause for that column in the schema.

Here's an example of omitting the `NOT NULL` clause on the primary key column `SingerId`. Note that because `SingerId` is the primary key, there can be at most only one row in the `Singers` table that stores `NULL` in that column.

The nullable property of the primary key column must match between the parent and the child table declarations. In this example, `Albums.SingerId INT64 NOT NULL` is not allowed. The key declaration must omit the `NOT NULL` clause because `Singers.SingerId` omits it.

These cannot be of type `ARRAY`:

- A table's key columns.
- An index's key columns.

You might want to provide multitenancy if you are storing data that belongs to different customers. For example, a music service might want to store each individual record label's separately.

The classic way to design for multitenancy is to create a separate database for each customer. In this example, each database has its own `Singers` table:

Database 1: Ackworth Records

SingerId	FirstName	LastName
1	Marc	Richards
2	Catalina	Smith

Database 2: Cama Records

SingerId	FirstName	LastName
3	Marc	Richards
4	Gabriel	Wright

Database 3: Eagan Records

SingerId	FirstName	LastName
1	Benjamin	Martinez
2	Hannah	Harris

The recommended way to design for multitenancy in Cloud Spanner is to use a different primary key value for each customer. You include a `CustomerId` key, or similar key, column in your tables. If you make `CustomerId` the first key column, then the data for each customer has good locality. Cloud Spanner automatically splits your data across your nodes based on size and load patterns. In this example, there is a single `Singers` table for all customers:

Cloud Spanner multitenancy database

CustomerId	SingerId	FirstName	LastName
1	1	Marc	Richards
1	2	Catalina	Smith
2	3	Marc	Richards
2	4	Gabriel	Wright
3	1	Benjamin	Martinez
3	2	Hannah	Harris

If you must have separate databases for each tenant, there are constraints to be aware of:

- There are limits (/spanner/quotas) on the number of databases per instance and tables per database. Depending on the number of customers, it might not be possible to have separate databases or tables.
- Adding new tables and non-interleaved indexes can take a long time (/spanner/docs/schema-updates#schema_update_performance). You might not be able to get the performance you want if your schema design depends on adding new tables and indexes.

If you want to create separate databases, you might have more success if you distribute your tables across databases in such a way that each database has a low number of schema changes per week (/spanner/docs/schema-updates#week-window).

If you create separate tables and indexes for each customer of your application, do not put all of the tables and indexes in the same database. Instead, split them across many databases, to mitigate the performance issues (/spanner/docs/schema-updates#large-updates) with creating a large number of indexes. There are also limits on the number of tables and indexes per database.