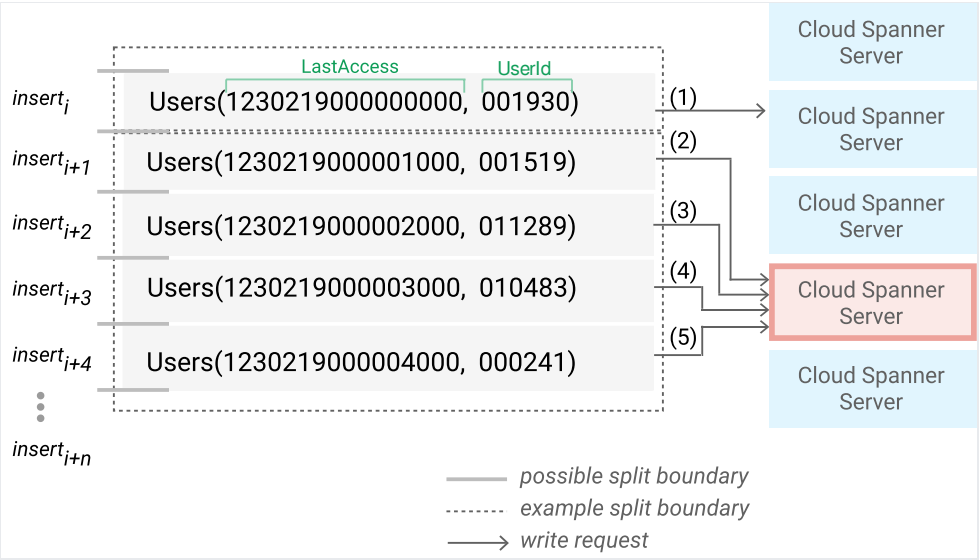
This page describes best practices for designing Cloud Spanner schemas to avoid hotspots and for loading data into Cloud Spanner.

As mentioned in Schema and data model (/spanner/docs/schema-and-data-model#choosing_a_primary_key), you should be careful when choosing a primary key to not accidentally create hotspots in your database. One cause of hotspots is having a column whose value monotonically increases as the first key part, because this results in all inserts occurring at the end of your key space. This pattern is undesirable because Cloud Spanner divides data among servers by key ranges, which means all your inserts will be directed at a single server that will end up doing all the work.

For example, suppose you want to maintain a last access timestamp column on rows of `Users` data. The following table definition that uses a timestamp-based primary key as the first key part is an anti-pattern:

The problem here is that rows will be written to this table in order of last access timestamp, and because last access timestamps are always increasing, they're always written to the end of the table. The hotspot is created because a single Cloud Spanner server will receive all of the writes, which overloads that one server.

The diagram below illustrates this pitfall:



The `Users` table above includes five example rows of data, which represent five different users taking some sort of user action about a millisecond apart from each other. The diagram also annotates the order in which the rows are inserted (the labeled arrows indicate the order of writes for each row). Because inserts are ordered by timestamp, and the timestamp value is always increasing, the inserts are always added to the end of the table and are directed at the same split. (As discussed in Schema and data model (/spanner/docs/schema-and-data-model#database-splits), a split is a set of rows from one or more related tables that are stored in order of row key.)

This is problematic because Cloud Spanner assigns work to different servers in units of splits, so the server assigned to this particular split ends up handling all the insert requests. As the frequency of user access events increases, the frequency of insert requests to the corresponding server also increases. The server then becomes prone to becoming a hotspot, as indicated by the red border and background above. Note that in this simplified illustration, each server handles at most one split but in actuality each Cloud Spanner server can be assigned more than one split.

When more rows are appended to the table, the split grows, and when it reaches its maximum size (approximately 4 GB (#limit_row_size)), Cloud Spanner creates another split, as described in Load-based splitting (/spanner/docs/schema-and-data-model#load-based_splitting). Subsequent new rows are appended to this new split, and the server that is assigned to it becomes the new potential hotspot.

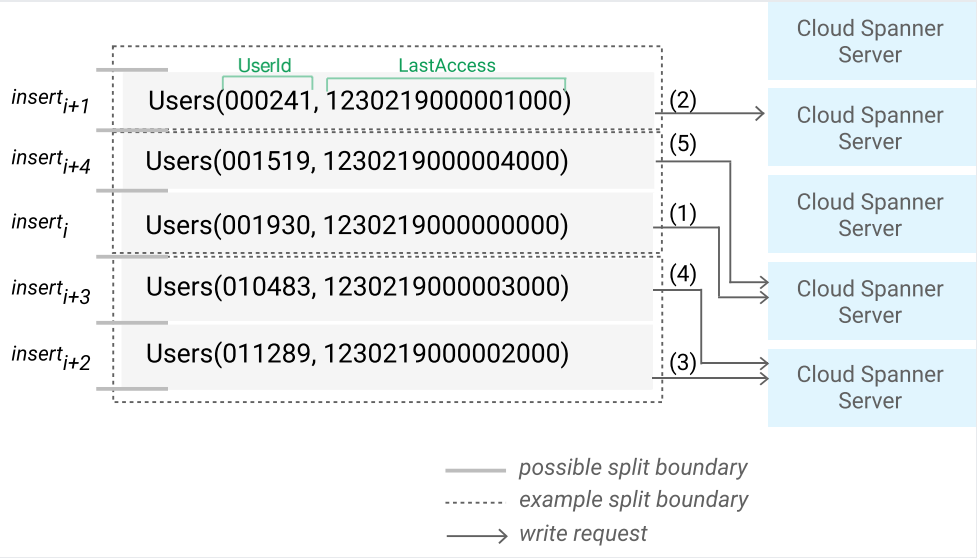
When hotspots occur, you might observe that your inserts are slow and other work on the same server might slow down. Changing the order of the `LastAccess` column to ascending order doesn't solve this problem because then all the writes are inserted at the top of the table instead, which still sends all the inserts to a single server.

na design best practice #1: Do not choose a column whose value monotonically increases or decrea
e first key part.

One way to spread writes over the key space is to swap the order of the keys so that the column that contains the monotonically increasing or decreasing value is not the first key part:

In this modified schema, inserts are now ordered by `UserId`, rather than by chronological last access timestamp. This schema spreads writes among different splits because it's unlikely that a single user will produce thousands of events per second.

The diagram below illustrates the five rows from the `Users` table ordered by `UserId` instead of by access timestamp:



Here the `Users` data is chunked into three splits, with each split containing on the order of a thousand rows of ordered `UserId` values. This is a reasonable estimate of how the user data could be split, assuming each row contains about 1MB of user data and given a maximum split size on the order of GB. Even though the user events occurred about a millisecond apart, each event was done by a different user, so the order of inserts is much less likely to create a hotspot compared with ordering by timestamp.

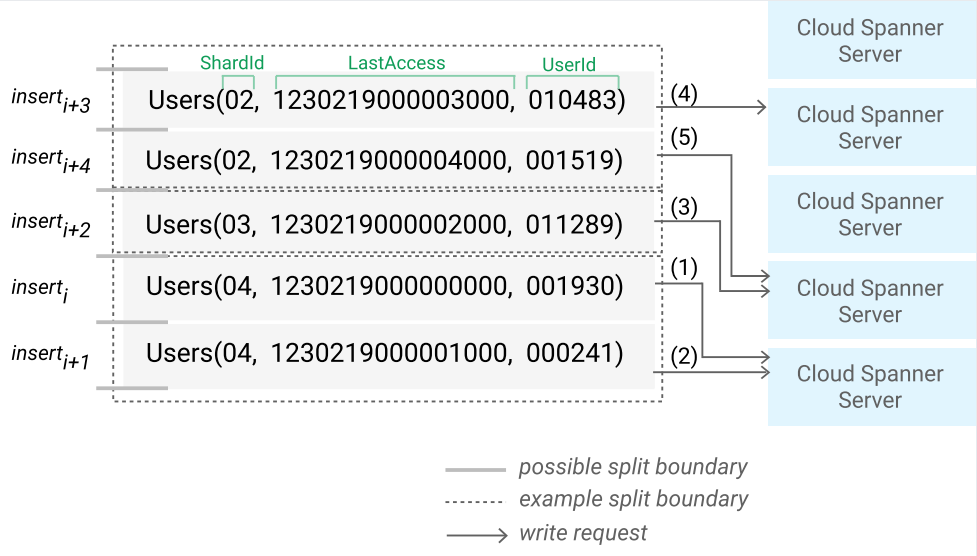
See also the related best practice for ordering timestamp-based keys (#ordering_timestamp-based_keys).

Another common technique for spreading the load across multiple servers is to create a column that contains the hash of the actual unique key, then use the hash column (or the hash column and the unique key columns together) as the primary key. This pattern helps avoid hotspots, because new rows are spread more evenly across the key space.

You can use the hash value to create logical shards, or partitions, in your database. (In a physically sharded database, the rows are spread across several databases. In a logically sharded database, the shards are defined by the data in the table.) For example, to spread writes to the `Users` table across N logical shards, you could prepend a `ShardId` key column to the table:

To compute the `ShardId`, you hash a combination of the primary key columns and calculate modulo N of the hash - `ShardId = hash(LastAccess and UserId) % N`. Your choice of hash function and combination of columns determines how the rows are spread across the key space. Cloud Spanner will then create splits across the rows to optimize performance. Note that the splits might not align with the logical shards.

The diagram below illustrates how using a hash to create three logical shards can spread write throughput more evenly across servers:



Here the `Users` table is ordered by `ShardId`, which is calculated as a hash function of key columns. The five `Users` rows are chunked into three logical shards, each of which is coincidentally in a different split. The inserts are spread evenly among the splits, which balances write throughput to the three servers that handle the splits.

Your choice of hash function will determine how well your insertions are spread across the key range. You don't need a cryptographic hash, although a cryptographic hash can be a good choice. When picking a hash function, you need to consider several factors:

- Avoiding hotspots. A function that results in more splits tends to reduce hotspots.
- Reading efficiency. Reads are faster if there are fewer splits to scan.
- Node count.

You can use a Universally Unique Identifier (UUID) as defined by RFC 4122 (https://tools.ietf.org/html/rfc4122) as the primary key. Version 4 UUID is recommended, because it uses random values in the bit sequence. Version 1 UUID stores the timestamp in the high order bits and is not recommended.

There are several ways to store the UUID as the primary key:

- In a `STRING(36)` column.
- In a pair of `INT64` columns.
- In a `BYTES(16)` column.

There are a few disadvantages to using a UUID:

- They are large, using 16 bytes or more. Other options for primary keys don't use this much storage.
- They carry no information about the record. For example, a primary key of SingerId and AlbumId has an inherent meaning, while a UUID does not.
- You lose locality between records that are related, which is why using a UUID eliminates hotspots.

When you generate unique primary keys that are numerical, the high order bits of subsequent numbers should be distributed roughly equally over the entire number space. One way to do this is to generate sequential numbers by conventional means, then bit-reverse them to obtain the final values.

Reversing the bits maintains unique values across the primary keys. You need to store only the reversed value, because you can recalculate the original value in your application code.

The size of a row should be less than 4 GB for best performance. The size of a row includes the top-level row and all of its interleaved child and index rows. Cloud Spanner typically creates a new split when an existing split reaches 4 GB, and Cloud Spanner can only split along top-level rows. If a row is larger than 4 GB, then write throughput might be affected.

Cloud Spanner can only create splits along top-level rows. Consider this example with three interleaved tables:

Cloud Spanner creates splits that keep all the albums and songs for each singer together in the same split. If it turns out that the songs for a single singer become a hotspot for reads or writes, Cloud Spanner cannot split the `Songs` table across servers. If, instead, `Songs` is a top-level table, then Cloud Spanner can create splits based on songs:

If you have a table for your history that's keyed by timestamp, consider using descending order for the key column(s) if any of the following apply:

- **If you're using an interleaved table for the history, and you'll be reading the parent row as well**. In this case, with a `DESC` timestamp column, the latest history entries are stored adjacent to the parent row. Otherwise, reading the parent row and its recent history will require a seek in the middle to skip over the older history.
- **If you're reading sequential entries in reverse chronological order, and you don't know exactly how far back you're going**. For example, you might use a SQL query with a `LIMIT` to get the most recent N events, or you might plan to cancel the read after you've read a certain number of rows. In these cases, you want to start with the most recent entries and read sequentially older entries until your condition has been met, which Cloud Spanner does more efficiently for timestamp keys that are stored in descending order.

Add the `DESC` keyword to make the timestamp key descending. For example:

na design best practice #2: Use descending order for timestamp-based keys.

Similar to the previous primary key anti-pattern, it's also a bad idea to create non-interleaved indexes on columns whose values are monotonically increasing or decreasing, even if they aren't primary key columns.

For example, suppose you define the following table, in which `LastAccess` is a non-primary-key column:

It might seem convenient to define an index on the `LastAccess` column for quickly querying the database for user accesses "since time X", like this:

However, this results in the same pitfall as described in the previous best practice, because indexes are implemented as tables under the hood, and the resulting index table would use a column whose value monotonically increases as its first key part.

It is okay to create an interleaved index like this though, because rows of interleaved indexes are interleaved in corresponding parent rows, and it's unlikely for a single parent row to produce thousands of events per second.

ma design best practice #3: Do not create a non-interleaved index on a column whose value tonically increases or decreases.

- Look through examples of schema designs (https://cloudplatform.googleblog.com/2018/06/What-DBAs-need-to-know-about-Cloud-Spanner-part-1-Keys-and-indexes.html)
  .
- Learn about bulk loading data (/spanner/docs/bulk-loading).