This document describes best practices for using Spanner as the primary backend database for game state storage. You can use Spanner (/spanner/) in place of common databases to store player authentication data and inventory data. This document is intended for game backend engineers working on long-term state storage, and game infrastructure operators and admins who support those systems and are interested in hosting their backend database on Google Cloud.

Multiplayer and online games have evolved to require increasingly complex database structures for tracking player entitlements, state, and inventory data. Growing player bases and increasing game complexity have led to database solutions that are a challenge to scale and manage, frequently requiring the use of sharding (https://wikipedia.org/wiki/Shard_(database_architecture)) or clustering (https://wikipedia.org/wiki/MySQL_Cluster). Tracking valuable in-game items or critical player progress typically requires transactions and is challenging to work around in many types of distributed databases.

Spanner is the first scalable, enterprise-grade, globally distributed, and strongly consistent database service built for the cloud to combine the benefits of relational database structure with non-relational horizontal scale. Many game companies have found it to be well-suited to replace both game state and authentication databases in production-scale systems. You can scale for additional performance or storage by using the Cloud Console to add nodes. Spanner can transparently handle global replication with strong consistency, eliminating your need to manage regional replicas.

This best practices document discusses the following:

- Important Spanner concepts and differences from databases commonly used in games.
- When Spanner is the right database for your game.
- Patterns to avoid when using Spanner for games.
- Designing your database operations with Spanner as your game's database.
- Modeling your data and creating a schema to get the best performance with Spanner.

Entitlements

Games, expansions, or in-app purchases belonging to a player.

Personally identifiable information (PII)

In games, information that typically includes email address and payment account information, such as a credit card number and billing address. In some markets, this information might include a national ID number.

Game database (game DB)

A database that holds player progress and inventory for a game.

Authentication database (auth DB)

A database that includes player entitlements and the PII that the players use when making a purchase. The auth DB is also known as the account DB or player DB. This database is sometimes combined with the game DB, but they are frequently separated in studios or publishers that have multiple titles.

Transaction

A database transaction (https://wikipedia.org/wiki/Database_transaction)—a set of write operations that have an all-or-nothing effect. Either the transaction succeeds and all updates take effect, or the database is returned to a state that doesn't include any of the updates of the transaction. In games, database transactions are most critical when processing payments, and when assigning the ownership of valuable in-game inventory or currency.

Relational database management system (RDBMS)

A database system based on tables and rows that reference one another. SQL Server, MySQL, and (less commonly) Oracle® are examples of relational databases used in games. These are frequently used because they can provide familiar methodologies and strong guarantees around transactions (https://docs.oracle.com/cd/E17275_01/html/programmer_reference/rep_trans.html).

NoSQL database (NoSQL DB)

Databases that are not structured relationally. These databases are becoming more popular in games because they have a lot of flexibility when the data model changes. NoSQL databases include MongoDB and Cassandra.

Primary key

Usually the column that contains the unique ID for inventory items, player accounts, and purchase transactions.

Instance

A single database. For example, a cluster runs multiple copies of the database software, but appears as a single instance to the game backend.

Node

For the purposes of this document, a single machine running a copy of the database software.

Replica

A second copy of a database. Replicas are frequently used for data recovery and high availability, or to help increase read throughput.

Cluster

Multiple copies of the software running on many machines that together appear as a single instance to the game backend. Clustering is used for scalability and availability.

Shard

An instance of a database. Many game studios run multiple homogeneous database instances, each of which holds a subset of the game data. Each of these instances is commonly referred to as a *shard*. Sharding is typically done for performance or scalability, sacrificing management efficiency while increasing app complexity. Sharding in Spanner is implemented using *splits*.

Split

Spanner divides your data into chunks called *splits* (/spanner/docs/schema-and-data-model#database-splits), where individual splits can move independently from each other and get assigned to different servers. 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". Spanner automatically adds and removes split boundaries, which changes the number of splits in the database. 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.

Hotspot

When a single split in a distributed database like Spanner contains records receiving a large portion of all the queries going to the database. This scenario is undesirable because it degrades performance.

In most cases where you are considering an RDBMS for your game, Spanner is an appropriate choice because it can effectively replace either the game DB, the auth DB, or in many cases, both.

Spanner can operate as a single worldwide transactional authority, which makes it an outstanding fit for game inventory systems. Any in-game currency or item that can be traded, sold, gifted, or otherwise transferred from one player to another presents a challenge in large-scale game backends. Often, the popularity of a game can outpace a traditional database's ability to handle everything in a single-node database. Depending on the type of game, the database can struggle with the number of operations required to handle the player load as well as the amount of stored data. This often leads game developers to shard their database for additional performance, or to store ever-growing tables. This type of solution leads to operational complexity and high maintenance overhead.

To help mitigate this complexity, one common strategy is to run completely separate game regions with no way to move data between them. In this case, items and currency cannot be traded between players in different game regions, because inventories in each region are segregated into separate databases. However, this setup sacrifices the preferred player experience, in favor of developer and operational simplicity.

On the other hand, you can allow cross-region trades in a geographically sharded database, but often at a high complexity cost. This setup requires that transactions span multiple database instances, leading to complex, error-prone application-side logic. Trying to get transaction locks on multiple databases can have significant performance impacts. In addition, not being able to rely on atomic transactions (https://wikipedia.org/wiki/Atomicity_(database_systems)) can lead to player exploits such as in-game currency or item duplication, which harm the game's ecosystem and community.

Spanner can simplify your approach to inventory and currency transactions. Even when using Spanner to hold all of your game data worldwide, it offers read-write transactions with even stronger than traditional atomicity, consistency, isolation, and durability (ACID (/spanner/docs/transactions#rw_transaction_semantics)) properties. With the scalability of Spanner, it means that data doesn't need to be sharded into separate database instances when more performance or storage is needed; instead, you simply add more nodes. Additionally, the high availability and data resiliency for which games often cluster their databases are handled transparently by Spanner, requiring no additional setup or management.

Auth DBs can also be well served by Spanner, especially if you want to standardize on a single RDBMS at your studio or publisher level. Although auth DBs for games often don't require the scale of Spanner, the transactional guarantees and high data availability can make it compelling. Data replication in Spanner is transparent, synchronous, and built-in. Spanner has configurations offering either 99.99% ("four nines") or 99.999% ("five nines") of availability (/spanner/#scale--sql), with "five nines" corresponding to less than five and a half minutes of unavailability in a year. This type of availability makes it a good choice for the critical authentication path required at the beginning of every player session.

This section provides recommendations for how to use Spanner in game design. It's important to model your game data to benefit from the unique features offered by Spanner. Although you can access Spanner by using relational database semantics, some schema design points can help you increase your performance. The Spanner documentation has detailed schema design recommendations (/spanner/docs/schema-design) that you can review, but the following sections are some best practices for game DBs.

The practices in this document are based on experiences from customer usage and case studies (/spanner/docs/media#customer_stories).

The player table typically has one row for each player and their in-game currency, progress, or other data that doesn't map easily to discrete inventory table rows. If your game allows players to have separate saved progress for multiple characters, like many large persistent massively multiplayer games, then this table typically contains a row for each character instead. The pattern is otherwise the same.

We recommend using a globally unique character or player identifier (character ID) as the primary key of the character table. We also recommend using the Universally Unique Identifier (UUID) v4

(/spanner/docs/schema-design#uuid_primary_key), because it spreads the player data across DB nodes and can help you get increased performance out of Spanner.

The inventory table often holds in-game items, such as character equipment, cards, or units. Typically, a single player has many items in their inventory. Each item is represented by a single row in the table.

Similar to other relational databases, an inventory table in Spanner has a primary key that is a globally unique identifier for the item, as illustrated in the following table.

itemID	type	playerID
`7c14887e-8d45`	`1`	`6f1ede3b-25e2`
`8ca83609-bb93`	`40`	`6f1ede3b-25e2`
`33fedada-3400`	`1`	`5fa0aa7d-16da`
`e4714487-075e`	`23`	`5fa0aa7d-16da`
`d4fbfb92-a8bd`	`14`	`5fa0aa7d-16da`
`31b7067b-42ec`	`3`	`26a38c2c-123a`

In the example inventory table, `itemID` and `playerID` are truncated for readability. An actual inventory table would also contain many other columns that aren't included in the example.

A typical approach in an RDBMS for tracking item ownership is to use a column as a foreign key that holds the current owner's player ID. This column is the primary key of a separate database table. In Spanner, you can use interleaving (/spanner/docs/schema-and-data-model#creating-interleaved-tables), which stores the inventory rows near the associated player table row for better performance. When using interleaved tables, keep the following in mind:

- You need to keep the total data in the player row and all their descendant inventory rows under ~2 GiB. This restriction isn't typically an issue with an appropriate data model design.
- You cannot generate an object without an owner. You can avoid ownerless objects in the game design provided the limitation is known ahead of time.

Many game developers implement indexes on many of the inventory fields to optimize certain queries. In Spanner, creating or updating a row with data in that index generates additional write load proportional to the number of indexed columns. You can improve Spanner performance by eliminating indexes that aren't used frequently, or by implementing these indexes in other ways that don't impact database performance (/spanner/docs/whitepapers/optimizing-schema-design#index_options).

In the following example, there is a table for long-term player high-score records:

This table contains the player ID (UUIDv4), a number representing a game mode, stage, or season, and the player's score.

Storing player scores in the database as shown in this example is *not* a suitable approach for maintaining a frequent
ed live leaderboard. For that use case, we recommend an in-memory implementation using Redis, and periodically sa
cores to long-term storage as necessary.

In order to speed up queries that filter for the game mode, consider the following index:

If everyone plays the same game mode called `1`, this index creates a hotspot where `GameMode=1`. If you want to get a ranking for this game mode, the index only scans the rows containing `GameMode=1`, returning the ranking quickly.

If you change the order of the previous index, you can solve this hotspot problem:

This index won't create a significant hotspot from players competing in the same game mode, provided their scores are distributed across the possible range. However, getting scores won't be as fast as with the previous index because the query scans all scores from all modes in order to determine if `GameMode=1`.

As a result, the reordered index solves the previous hotspot on game mode but still has room for improvement, as illustrated in the following design.

We recommend moving the game mode out of the table schema, and use one table per mode, if possible. By using this method, when you retrieve the scores for a mode, you only query a table with scores for that mode in it. This table can be indexed by score for fast retrieval of score ranges without significant danger of hotspots (provided the scores are well distributed). As of the writing of this document, the maximum number of tables per database (/spanner/quotas) in Spanner is 2048, which is more than enough for most games.

Unlike other workloads, where we recommend designing for multitenancy in Spanner (/spanner/docs/schema-and-data-model#recommended_multitenancy) by using different primary key values, for gaming data, we recommend the more conventional approach of separate databases per tenant (/spanner/docs/schema-and-data-model#classic_multitenancy). Schema changes are common with the release of new game features in live service games, and isolation of tenants at a database level can simplify schema updates. This strategy can also optimize the time it takes to back up or restore a tenant's data, because these operations are performed on an entire database at once.

Unlike some conventional relational databases, Spanner remains operational during schema updates. All queries against the old schema are returned (although they might return less quickly than usual), and queries against the new schema are returned as they become available. You can design your update process to keep your game running during schema updates when running on Spanner, provided you keep the preceding constraints in mind.

However, if you request another schema change while one is currently being processed, the new update is queued and won't take place until all previous schema updates have completed. You can avoid this situation by planning larger schema updates, instead of issuing many incremental schema updates in a short period. For more information about schema updates, including how to perform a schema update that requires data validation (/spanner/docs/schema-updates#updates-that-require-validation), see Cloud Schema updates (/spanner/docs/schema-updates).

When you develop your game server and platform services to use Spanner, consider how your game accesses the database and how to size the database to avoid unnecessary costs.

When you develop against Spanner, consider how your code interfaces with the database. Spanner offers native client libraries (/spanner/docs/reference/libraries) for many popular languages, which are typically feature-rich and performant. JDBC drivers (/spanner/docs/partners/drivers) are also available, which support data manipulation language (DML) and data definition language (DDL) statements. In cases where Spanner is used in new development, we recommend using the Cloud Client Libraries for Spanner. Although typical game engine integrations don't have much flexibility in language selection, for platform services accessing Spanner, there are cases of gaming customers using Java or Go. For high throughput applications, select a library where you can use the same Spanner client for multiple sequential requests.

During development, a single-node Spanner instance is likely sufficient for most activities, including functional testing.

ng: We don't recommend a single-node instance in a production environment with live players. Before you put live tra
uction environment, see the section about sizing considerations for production
_the_production_environment_to_anticipate_peak_demand).

When you move from development to testing, and then into production, it's important that you reevaluate your Spanner needs to insure your game can handle live player traffic.

Before you move to production, load tests are crucial to verify that your backend can handle the load during production. We recommend running load tests with double the load you expect in production in order to be prepared for spikes in usage and cases where your game is more popular than anticipated.

Running a load test with synthetic data (https://wikipedia.org/wiki/Synthetic_data) isn't sufficient. You should also run load tests using data and access patterns as close as possible to what is expected in production. Synthetic data might not detect potential hotspots in your Spanner schema design. Nothing is better than running a beta test (open or closed) with real players to verify how Spanner behaves with real data.

The following diagram is an example player table schema from a game studio that illustrates the importance of using beta tests to load test.

List of players names and an attribute
for load testing.

The studio prepared this data based on trends from a previous game that they had operated for a couple of years. The company expected the schema to represent the data in this new game well.

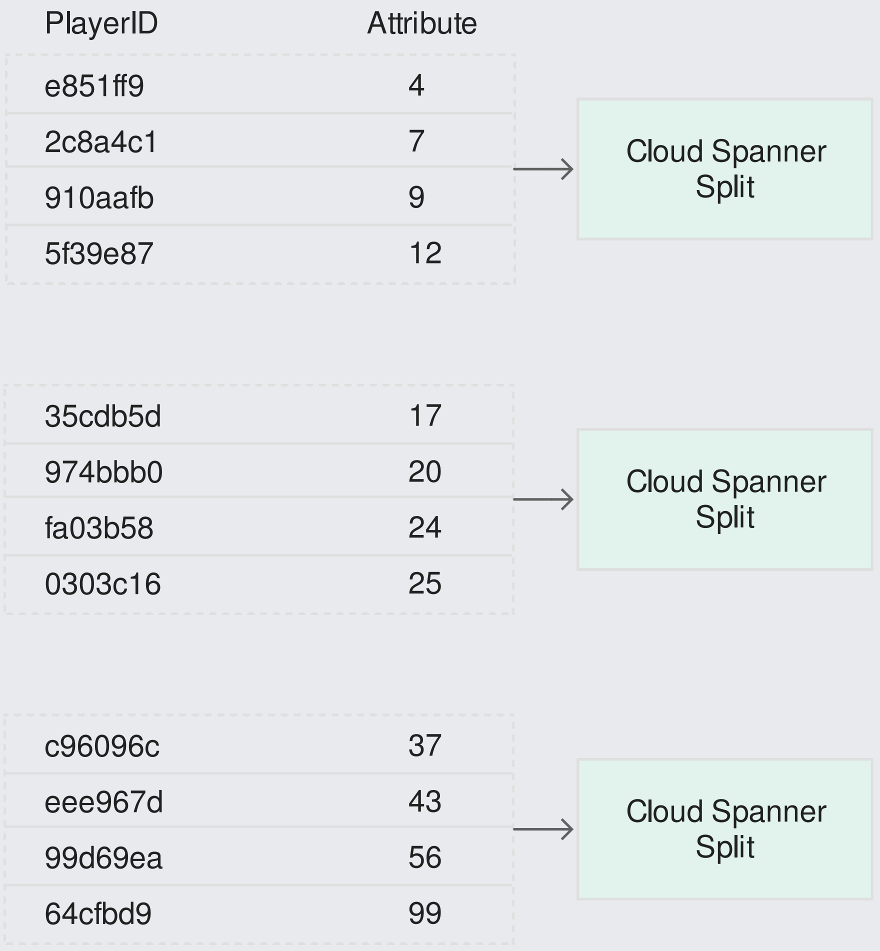
Each player record has some numerical attributes associated with it that tracks the player's progress in the game (such as rank, and play time). For the example attribute used in the preceding table, new players are given a starting value of 50, and this value then changes to a value between 1 and 100 as the player advances.

The studio wants to index this attribute in order to speed up important queries during gameplay.

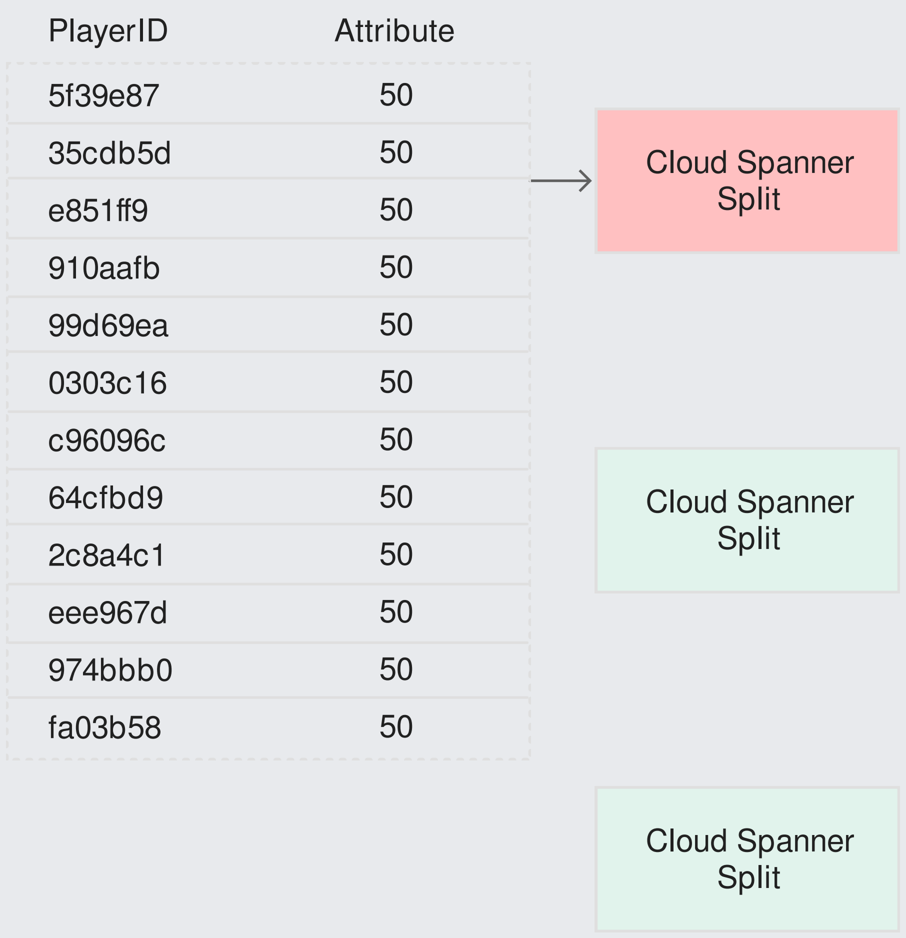
Based on this data, the studio created the following Spanner table, with a primary key using the `PlayerID` and a secondary index on `Attribute`.

And the index was queried to find up to ten players with `Attribute=23`, like this:

According to the documentation on optimizing schema design (/spanner/docs/whitepapers/optimizing-schema-design), Spanner stores index data in the same way as tables, with one row per index entry. In load tests, this model does an acceptable job of distributing the secondary index read and write load across multiple Spanner splits, as illustrated in the following diagram:

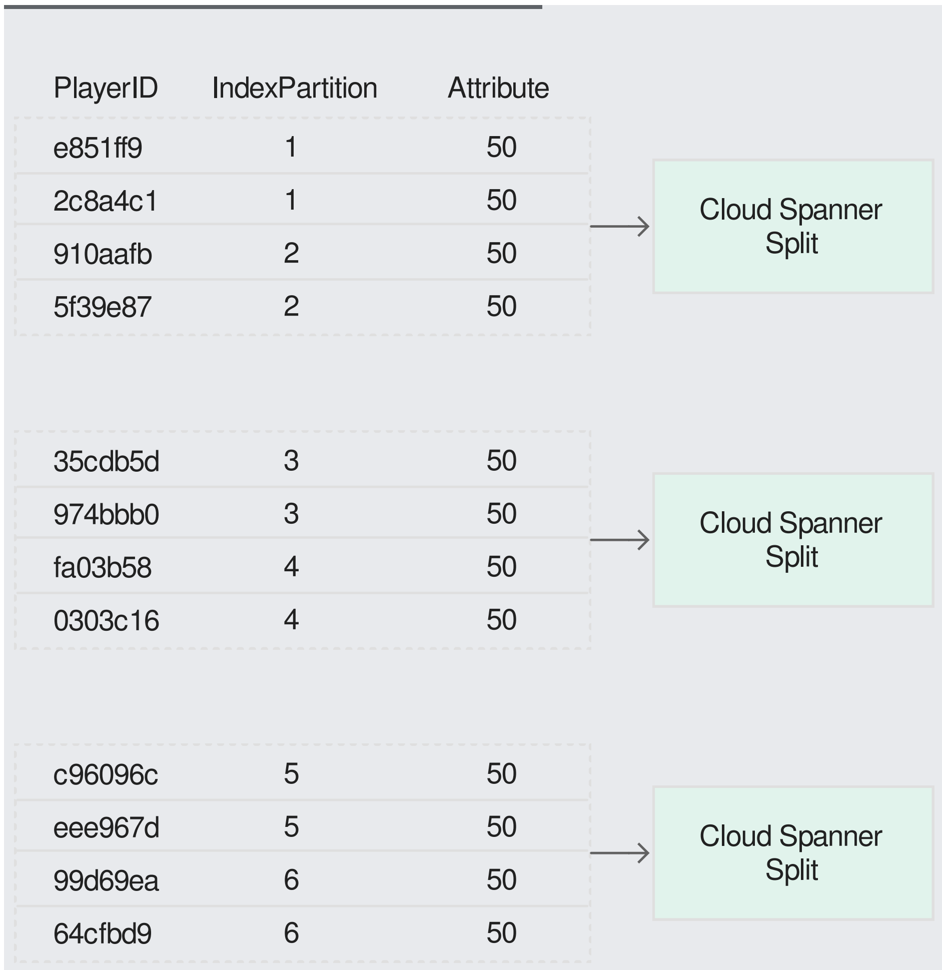


Although the synthetic data used in the load test is similar to the eventual steady state of the game where `Attribute` values are well distributed, the game design dictates that all players start with `Attribute=50`. Because each new player starts with `Attribute=50`, when new players join they are inserted in the same part of the `idx_attribute` secondary index. This means updates are routed to the same Spanner split, causing a hotspot during the game's launch window. This is an ineffecient use of Spanner.



PlayerID	Attribute
5f39e87	50
35cdb5d	50
e851ff9	50
910aafb	50
99d69ea	50
0303c16	50
c96096c	50
64cfbd9	50
2c8a4c1	50
eee967d	50
974bbb0	50
fa03b58	50

In the following diagram, adding an `IndexPartition` column to the schema after the launch resolves the hotspot issue, and players are evenly distributed across the available Spanner splits. The updated command for creating the table and index looks like this:



The `IndexPartition` value needs to have a limited range for efficient querying, but it should also have range that is at least double the number of splits for efficient distribution.

In this case, the studio manually assigned every player an `IndexPartition` between `1` and `6` in the game application.

Alternative methods could be to assign a random number to each player, or assigning a value derived from a hash on the `PlayerID` value. See What DBAs need to know about Cloud Spanner, part 1: Keys and indexes
(https://cloud.google.com/blog/products/gcp/what-dbas-need-to-know-about-cloud-spanner-part-1-keys-and-indexes)
for more application-level sharding strategies.

Updating the previous query to use this improved index looks like the following:

Because no beta test was run, the studio didn't realize they were testing by using data with incorrect assumptions. Although synthetic load tests are a good way to validate how many queries per second (QPS) (https://wikipedia.org/wiki/Queries_per_second) your instance can handle, a beta test with real players is necessary to validate your schema and prepare a successful launch.

Major games often experience the peak of their traffic at launch. Building up a scalable backend applies not only to platform services and dedicated game servers (/solutions/gaming/cloud-game-infrastructure), but to databases as well. Using Google Cloud solutions such as App Engine (/appengine/docs/), you can build frontend API services that can scale up quickly. Even though Spanner offers the flexibility to add and remove nodes online, it isn't an autoscaling database. You need to provision enough nodes to handle the traffic spike at launch.

Based on the data you gathered during load testing or from any public beta testing, you can estimate the number of nodes required to handle requests at launch. It's a good practice to add a few nodes as buffer in case you get more players than expected. You should always size the database based on not exceeding an average CPU usage of 65%.

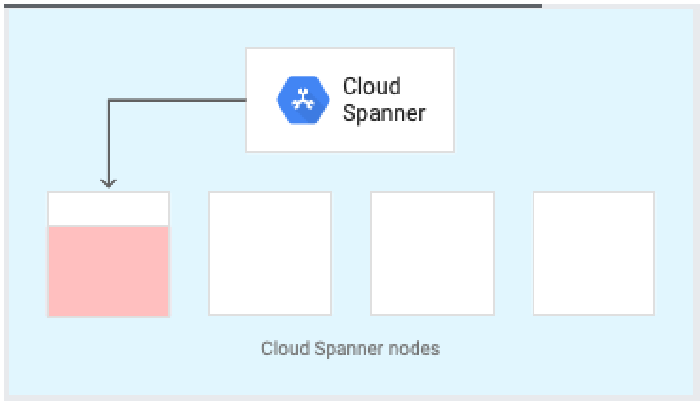
One other important thing to prepare before launch is warming the database up.

Spanner in a cold state can't provide you the nodes you need for launch without warming up first.

Spanner is a distributed database, which means that as your database grows, Spanner divides your data into chunks called splits. Individual splits can move independently from each other and get assigned to different servers, which can be in different physical locations. For more information, see Database splits (/spanner/docs/schema-and-data-model#database-splits).

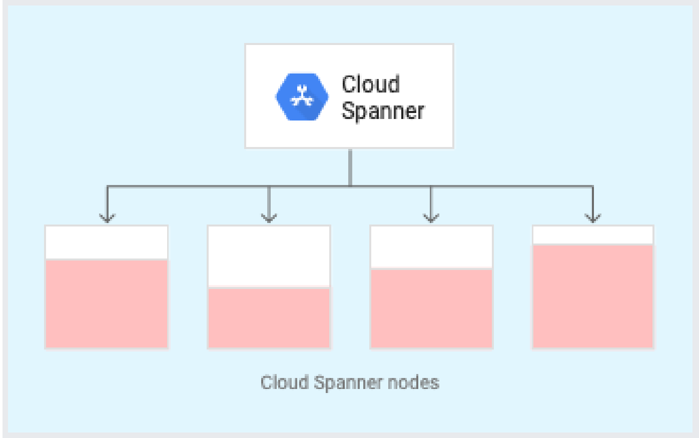
A split is defined as a range of rows. In other words, it contains a subset of your table. Spanner splits data based on load and size. That way, splits can be dynamically moved across Spanner nodes to balance the overall load on the database. The more data you insert into Spanner, the more splits are generated.

In the following diagram, there are four nodes.



Because you have no data in Spanner, when you start writing data you only write to a single node. Spanner is currently in a cold state.

The following diagram illustrates the split to the other nodes.



As the data comes into the system, Spanner starts to split that data to rebalance the load across the four provisioned nodes. Now Spanner is in a warm state.

You want to launch your game when Spanner is in a warm state with splits already balanced across all of the nodes. In order to warm your database up, follow these steps:

1. Make sure that the table primary keys you generate for your load test are in the same keyspace (have the same statistical properties) as the keys you are using for real production traffic
2. Run a load test two days before your launch. Run the load test for at least one hour at the expected peak load. The load test causes Spanner to create more splits due to load-based splitting.
3. After the load test is complete, you can delete the rows generated by your load test from your tables, but don't delete the tables themselves. This keeps the splits available for your launch window.

Any production database requires comprehensive monitoring and performance metrics. Spanner comes with built-in metrics in Stackdriver (/stackdriver/). Where possible, we recommend incorporating the provided gRPC libraries into your game backend process because these libraries include OpenCensus tracing (https://opencensus.io/tracing/). OpenCensus tracing lets you see query traces in Stackdriver as well as other supported open source tracing tools.

In Stackdriver, you can see details on your Spanner usage, including data storage and CPU usage. For most cases, we recommend that you base your Spanner scaling decisions on this CPU usage metric or observed latency. For more information about suggested CPU usage for optimized performance, see Best practices for instances (/spanner/docs/instances#regional-best-practices).

Spanner offers query execution plans (/spanner/docs/query-execution-plans). You can review these plans in the Cloud Console, and contact support if you need help understanding your query performance.

When you're evaluating performance, keep short cycle testing to a minimum because Spanner transparently splits your data behind the scenes to optimize performance based on your data access patterns. You should evaluate performance by using sustained, realistic query loads.

When you're working with Spanner, newly created tables haven't yet had an opportunity to undergo load-based or size-based splitting to improve performance. When you delete data by dropping a table and then recreating it, Spanner needs data, queries, and time to determine the correct splits for your table. If you are planning to repopulate a table with the same kind of data (for example, when running consecutive performance tests), you can instead run a `DELETE` query on the rows containing data you no longer need. For the same reason, schema updates should use the provided Cloud Spanner API, and should avoid a manual strategy, such as creating a new table and copying the data from another table or a backup file.

Many games must comply with data locality laws such as GDPR (/security/gdpr/) when played worldwide. To help support your GDPR needs, see the Google Cloud and the GDPR whitepaper (/security/gdpr/resource-center/pdf/googlecloud_gdpr_whitepaper_618.pdf) and select the correct Spanner regional configuration (/spanner/docs/instances#regional_configurations).

- Read about how Bandai Namco Entertainment used Spanner in their successful Dragon Ball Legends launch (/blog/products/gcp/behind-the-scenes-with-the-dragon-ball-legends-gcp-backend).
- Watch the Cloud Next '18 session on Optimizing Applications, Schemas, and Query Design on Spanner (https://www.youtube.com/watch?time_continue=231&v=DxrdatA_ULk).
- Read our guide on Migrating from DynamoDB to Spanner (/solutions/migrating-dynamodb-to-cloud-spanner).
- Try out other Google Cloud features for yourself. Have a look at our tutorials (/docs/tutorials).