Implementing Multi-Tenancy in Cloud Spanner

by Christoph Bussler and Sireesha Pulipati

Christoph Bussler Follow
Dec 1 · 23 min read

Introduction

Multi-tenancy is a software architecture pattern in which a single or few instances of an application serve multiple tenants or customers, often hundreds or thousands. This approach is fundamental to cloud computing platforms where the underlying infrastructure is shared among multiple organizations. Basically, multi-tenancy can be thought of as a form of partitioning based on shared computing resources like databases. An analogy is tenants in an apartment building: shared infrastructure, but dedicated tenant space. Multi-tenancy is also the hallmark of most, if not all, software as a service (SaaS) applications.

An example could be an HR SaaS provider implementing its multi-tenant application on Google Cloud. The multi-tenant application is accessed by several customers of the HR SaaS provider. These customers are called tenants in the context of the multi-tenant application. In this article, the terms “tenant”, “customer”, or “organization” are used interchangeably to indicate the entity that is accessing the multi-tenant application.

In a multi-tenant application, each tenant’s data is isolated in one of several architecture approaches in the underlying database. Cloud Spanner is Google Cloud’s fully managed, enterprise-grade, distributed, and strongly consistent database which combines the benefits of the relational database model with non-relational horizontal scalability. Cloud Spanner has relational semantics with schemas, enforced data types, strong consistency and multi-statement ACID transactions, and a SQL query language implementing ANSI 2011 SQL.

Cloud Spanner provides zero-downtime for planned maintenance or region failures with up to 5 9s availability SLA. Cloud Spanner's ability to provide high availability and scalability makes it an attractive choice for modern multi-tenant applications. In this article, we are going to discuss the different architecture approaches to implement multi-tenancy with Cloud Spanner.

Categories of criteria

The different architecture approaches of how to map a tenant's data to Cloud Spanner are as follows:

- **Instance**. A tenant resides exclusively in one Cloud Spanner instance, with exactly one database for that tenant.
- **Database**. A tenant resides in a database, and several databases are in a single Cloud Spanner instance.
- **Schema**. A tenant resides in exclusive tables within a database, and several tenants can be located in the same database.
- **Table**. Tenant data are rows in tables shared with other tenants.

These are called data management patterns and are discussed in detail below. Their discussion is based on the following criteria:

- **Isolation**. The degree of isolation of data across multiple tenants is a major consideration. It is driven by the choices made for the criteria under other categories. For example, certain regulatory and compliance requirements may dictate greater degree of isolation.
- **Agility**. The ease of onboarding and offboarding activities for a tenant wrt. instance, database or table creation.
- **Operations**. The availability or complexity of implementing typical tenant-specific database operations and administration activities like regular maintenance, logging, backups, or disaster recovery operations.
- **Scale**. The ability to scale seamlessly to allow for the future growth of the size and the number of tenants. In the description of each partner the number of tenants is

discussed that can be supported by the pattern.

- **Performance**. Ability to allocate exclusive resources to each tenant addressing the noisy neighbor phenomenon and enabling predictable read/write performance for each tenant.
- **Cost**. The costs associated with whichever data management pattern selected and the ability to split the cost proportionally among tenants (if needed).
- **Regulations and compliance**. Features to address requirements of highly regulated industries and countries that may require complete isolation of resources, maintenance operations. For example, data residency requirements for France require that personally identifiable information (PII) is physically stored exclusively within France.

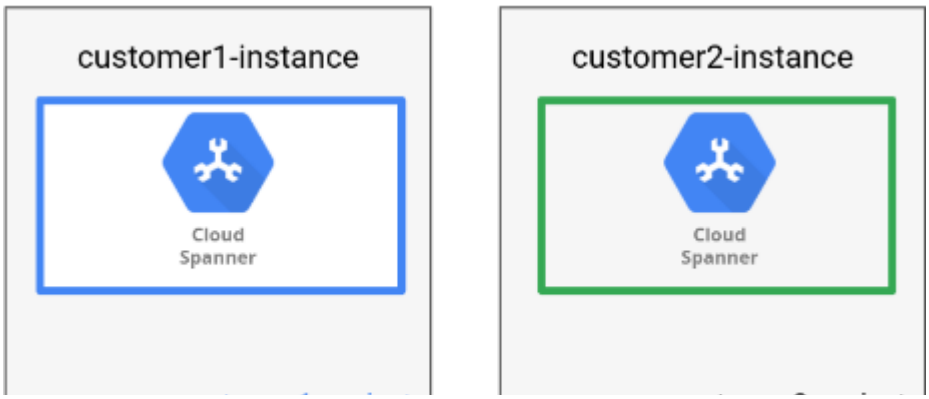
Each data management pattern is described in detail in the next section as it relates to these criteria.The same criteria might be used for the selection process of which pattern to use for a given set of tenants.

Data management patterns

The following sections describe the four main data management patterns in context of multi-tenancy: instance, database, schema and table.

Instance — one tenant per instance

In this data management pattern, each tenant's data is stored in its own Cloud Spanner instance and database. A Cloud Spanner instance can have one or more databases, however, in this pattern only one database is created in an instance for full and complete isolation. Following the example of the HR application above, a separate Cloud Spanner instance is created for each customer organization with one database each.



Data management pattern: one tenant per instance

Having separate instances for each tenant allows the use of separate Google Cloud projects to achieve separate trust boundaries for different tenants. An additional benefit is that each instance configuration, in terms of regional or multi-regional, can be chosen based on each tenant's location, thereby optimizing cost and performance.

This data management pattern provides a complete isolation of the tenant's data from other tenants. The architecture can easily scale for any number of tenants because the SaaS provider can create any number of instances in the desired regions without any practical hard limits. The following table depicts how this data management pattern performs against different criteria called out above.

Criteria	**"Instance — One tenant per instance" data management pattern**
Isolation	Enables greatest level of isolation; no database resources are shared among tenants.
Agility	Onboarding and offboarding requires considerable effort to set up or decommission the Cloud Spanner instance, instance specific security, and connectivity. This can be automated through Infrastructure as Code.
Operations	Backups can be performed independently for each tenant, providing separation and flexibility in backup schedules. This data management pattern results in higher operational overhead owing to the large number of instances to manage and maintain with respect to scaling, monitoring, logging, and performance tuning.
Scale	High scalability, both with respect to the number of customers as well as tenant specific scalability. Cloud Spanner, being a highly scalable database, can accommodate virtually unlimited growth for a tenant by increasing the number of nodes. This pattern can support an unlimited number of tenants as for each tenant a Cloud Spanner instance can be created.
Performance	There is no resource contention among different tenants owing to separate instances. Also allows for tailored performance tuning and troubleshooting for a tenant without impacting others.

Cost	Cloud Spanner is priced based on the number of nodes, amount of storage and amount of network used and not based on the number of instances. That means, using a single instance with a higher number of nodes vs multiple instances with a smaller number of nodes each totaling to the same number of nodes effectively amounts to the same cost. This data management pattern does not necessarily result in higher cost compared to other data management patterns in that aspect. However, isolated instances cannot share resources and thereby cannot attain resource efficiencies that may be possible with other data management patterns for applicable workload patterns.
Regulatory and compliance requirements	In this data management pattern it is possible to store data in specific regions, implement specific security, backup or auditing processes as per the regulatory and compliance requirements applicable for some industries.

In summary, the key takeaways are:

- Advantage: Highest level of isolation
- Disadvantage: Greatest operational overhead

This data management pattern is best suited in the following scenarios:

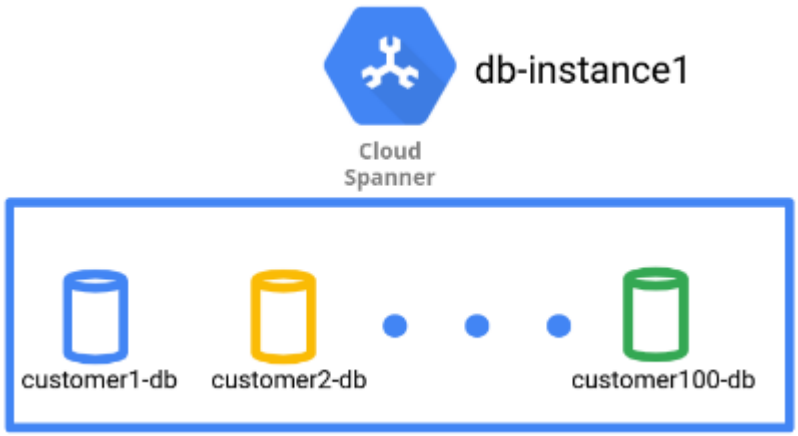
- Different tenants are spread across a wide range of regions and need a localized solution
- Regulatory and compliance requirements for some tenants demand greater levels of security and auditing protocols
- Size of tenants vary significantly so that sharing resources among high volume, high traffic tenants might cause contention and mutual degradation

Database — one tenant per database

In this data management pattern, each tenant resides in a database within a single Cloud Spanner instance. If one instance is insufficient for the number of tenants, multiple instances can be created. This implies that a single Cloud Spanner instance is shared among multiple tenants as multiple databases can reside in a single instance in the context of this pattern.

Cloud Spanner has a hard limit of 100 databases per instance. This means that if the SaaS Provider needs to scale beyond 100 customers, multiple Cloud Spanner instances need to be created and used.

In the case of the HR application, the SaaS provider creates and manages each tenant with a separate database in a Cloud Spanner instance.



Data management pattern: one tenant per database

This data management pattern achieves logical isolation on a database level for different tenants' data. However, since it is a single Cloud Spanner instance, all the tenant databases share the same regional configuration and underlying compute and storage setup. The following table describes how this data management pattern behaves against our criteria.

Criteria	**"Instance — One tenant per database" data management pattern**
Isolation	Complete logical isolation on a database level; underlying instance infrastructure resources are shared.
Agility	Onboarding and offboarding requires some effort to create or delete the database and any specific security controls. Can rely on automation through Infrastructure as Code.
Operations	Backups can be performed independently for each tenant providing flexibility in schedules. This data management pattern involves less operational overhead compared to the instance data management pattern, you have only one instance to monitor for performance and

	scale (for up to 100 databases).
Scale	Cloud Spanner instances can be scaled to thousands of nodes and can accommodate any level of growth for the tenants. There is a limit of 100 databases per Cloud Spanner instance currently which limits the number of tenants on-boarded onto a single instance. For every 100 tenants a new Cloud Spanner instance needs to be created and there is no limit to the number of instances.
Performance	Databases are spread across Cloud Spanner instance nodes and thus share the infrastructure resulting in resource contention among multiple databases (noisy neighbor effect) and impacts the performance.
Cost	Cloud Spanner is priced based on the resources consumed rather than on the number of databases or instances. So, this data management pattern doesn't necessarily result in higher cost compared to others. However, the shared resources in terms of compute and storage can result in efficiencies that can reduce the overall cost. The converse, where severe resource contention requires additional node capacity than otherwise, is possible too.
Regulatory and compliance requirements	In this data management pattern, it is not possible to meet any specific data residency regulatory requirements, if the desired location is different from the instance regional configuration.

In summary, the key takeaways are:

- Advantage: Higher level of isolation
- Disadvantage: Limited number of tenants per instance and location inflexibility

This data management pattern is best suited in the following scenarios:

- Multiple customers are in the same geographical area, for example, US, and/or are under the same regulatory authority
- Tenants require system-based data separation and backup/restore, but are fine with infrastructure resource sharing.

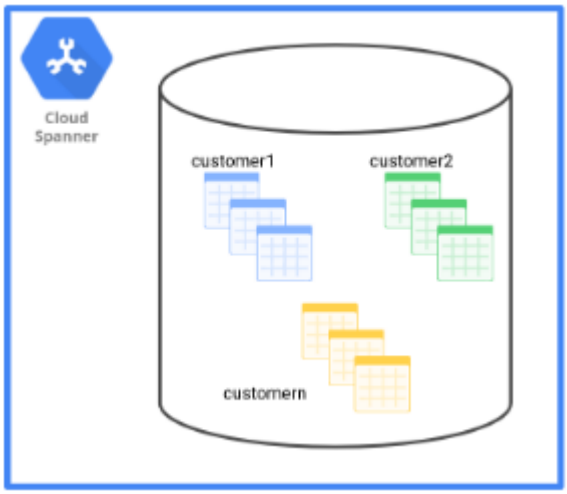
Schema — one set of tables for each tenant

In the schema data management pattern a single database, which implements a single schema, is used for multiple tenants with a separate set of tables used for each tenant's data. These sets of tables can be differentiated from each other by including tenant ID in the table names as either suffix or prefix.

This data management pattern of using a separate set of tables for each tenant provides a much lower level of isolation compared to the above options (instance level, database level). However, this approach makes onboarding quite simple, as it involves only creating new tables and associated referential integrity and indexes. However, one big caveat is that access permissions for Cloud Spanner through Cloud IAM can be provided only at the instance or database level and cannot be provided at the table level. There is also a limit on the number of tables per database — 5000, which limits the scalability of the application for a large number of customers. Furthermore, using separate tables for each customer can result in a large backlog of schema update operations that take a long time to complete.

For the example of HR application, the SaaS provider can create a set of tables for each customer with tenant ID as prefix in the table names like this -

customer1_employee, customer1_payroll, customer1_department, etc.



Data management pattern: one set of tables for each tenant

The following table outlines how this data management pattern affects different criteria.

Criteria	**"Schema — One set of tables for each tenant" data management pattern**
Isolation	Low level of isolation; no table level security.
Agility	Onboarding a customer is a fairly simple task involving creation of new tables and associated keys and indexes. Offloading a customer means deleting the tables, which may have a temporary negative impact on the performance of the other tenants within the database.
Operations	Operations including backups, monitoring, logging cannot be performed separately for tenants by Cloud Spanner itself. Those have to be implemented as separate functionality by the application itself or as utility scripts.
Scale	Cloud Spanner instances can be scaled to thousands of nodes and can accommodate any level of growth for the tenants. However, there is a limit of 5000 tables a single database can have. This pattern therefore supports only floor(5000/<number tables for tenant>) tenants in each database. If that is exhausted, a new database needs to be added for additional tenants.
Performance	High level of resource contention is possible (noisy neighbor). In order to ensure good performance, indexes need to be designed separately for each set of tables as well.
Cost	Cloud Spanner is priced based on the resources consumed rather than on the number of tables, databases or instances. So, this data management pattern does not necessarily result in higher cost compared to others. However, the shared resources in terms of compute and storage can result in efficiencies that can reduce the overall cost. The converse, where severe resource contention requires additional node capacity than otherwise, is possible too.
Regulatory and compliance requirements	In this data management pattern, it is not possible to meet any specific data residency regulatory requirements, if the desired location is different from the instance regional configuration. Also implementing specific security and auditing controls impacts all the tenants that reside in the same database.

In summary, the key takeaways are:

- Advantage: Ease of onboarding
- Disadvantage: Higher operational overhead and lack of security controls at the table level

This data management pattern is best suited in the following scenarios:

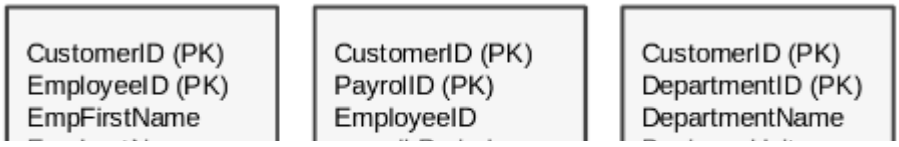
- Internal applications that cater to different departments where strict data security isolation is not a prominent concern compared to ease of maintenance.
- Multi-tenant applications where the data does not require strict separation based on legal or regulatory requirements.

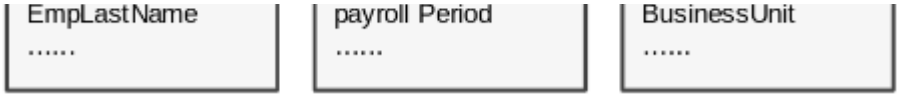
Note: while it is possible to create several sets of tables (each set representing a tenant) in a database, it is the least ideal pattern from a database perspective. The main reasons are that the tables must follow naming conventions and not only the application, but also any database tooling (e.g., IDE, schema migration tools) has to understand that. In addition, if the number of tables is reasonably large per tenant, this does not really provide significant scaling. A better approach would be to move to either a database per tenant and increase the number of instances, or move to the table pattern.

Table — one table for several tenants

The final data management pattern is to serve multiple tenants with a common set of tables. Each table contains data for several tenants. This data management pattern represents an extreme level of multi-tenancy where everything — from infrastructure to schema to data model — is shared completely among multiple tenants. Within a table, rows are partitioned based on primary keys with tenant ID as the first element of the key. From a scalability perspective Cloud Spanner can support this pattern best since it can scale tables without limitation.

Following the HR application example, the payroll table's primary key can be a combination of customer ID and payroll ID.





Data management pattern: one table for several tenants

Similar to the schema pattern, data access cannot be controlled separately for different tenants. Using fewer tables means schema update operations can be completed faster compared to when each tenant has their own tables within the database. This approach simplifies the onboarding, offboarding activities as well as operations to a large extent.

The following table evaluates this data management pattern against the list of criteria under consideration.

Criteria	**"Table — One table for several tenants" data management pattern**
Isolation	Lowest level of isolation; no tenant level security.
Agility	Onboarding a customer does not require any setup on the database side. The application can write data directly into the existing tables. Offboarding simply means deleting the customer's rows.
Operations	Operations including backups, monitoring, logging cannot be performed separately for tenants by Cloud Spanner and they have to be implemented separately. Little to no overhead as the number of tenants increases.
Scale	Cloud Spanner instances can be scaled to thousands of nodes and can accommodate any level of growth for the tenants. This pattern can support an unlimited number of tenants.
Performance	This pattern works very well in the context of Cloud Spanner as its internal distributed storage and processing as well as load balancing can easily deal with this pattern. A high level of resource contention is possible (noisy neighbor) if the primary key spaces are not designed carefully preventing concurrency and distribution, though. Following best practices is very important. Deleting a tenant''s data might have a temporary impact on the load.
Cost	Cloud Spanner is priced based on the resources consumed rather than on the number of tables,

```
              | databases or instances. The shared resources in terms
              | of compute and storage can result in efficiencies
              | that can reduce the overall cost. The converse, where
              | severe resource contention requires additional node
              | capacity than otherwise, is possible too.
--------------+-------------------------------------------------------
Regulatory    | In this data management pattern, it is not possible
and           | to meet any specific data residency regulatory
compliance    | requirements, if the desired location is different
requirements  | from the instance regional configuration. It cannot
              | provide system level partitioning either (compared to
              | the instance or database pattern). Also any specific
              | security and auditing controls cannot be implemented
              | without affecting all the tenants.
```

In summary, the key takeaways are:

- Advantage: Highly scalable and low operations overhead
- Disadvantage: High resource contention, lack of security controls for each tenant

This pattern is best suited in the following scenarios:

- Internal applications that cater to different departments where strict data security isolation is not a prominent concern compared to ease of maintenance.
- Maximum resource sharing for tenants using free tier application functionality when minimizing cost at the same time

Combination of data management patterns and tenant life cycle management

Overview of data management patterns

The following table compares the various data management patterns across all criteria in order to provide an overview on a very high level abstracting from the details above.

	Instance	Database	Schema	Table
Isolation	Complete	Complete	Low	Lowest
Agility	Low	Moderate	Moderate	Highest
Ease of Operations	High	High	Low	Low

Scale	High	Limited	Potentially very limited	High
Performance*	High	Moderate	Moderate	Potentially high
Cost**	see footnote			
Regulations and compliance	Highest	High	Low	Low

* Performance is heavily dependent on the schema design and query best practices and so the value here is only an average expectation.
** Cost is based on the number of nodes allocated to a Cloud Spanner instance. The direct comparison across the different patterns is difficult since the same number of tenants might result in different numbers of nodes depending on the query access patterns, transaction frequency, tenant access concurrency and schema design owing to the presence and level of resource contention. We highly recommend testing the different patterns in a POC before the final design decision.

The best data management patterns for a specific multi-tenant application are those that satisfy most of its requirements based on the criteria. If a particular criterion is not required for an application then the corresponding row does not play a role in the decision process.

Combination of data management patterns

In many cases, a single data management pattern is sufficient to address the requirements of a multi-tenant application. The design of a multi-tenant application can then assume a single data management pattern and all interactions with Cloud Spanner are based on the implemented data management pattern.

However, some multi-tenant applications require several data management patterns at the same time. A good example is a multi-tenant application that supports a free tier, a regular tier and an enterprise tier to its clients.

- **Free tier**. The free tier must be cost effective, and has for example an upper limit of the data volume supported. It usually supports limited functionality as well. The table data management pattern would be a good candidate for a free tier as the

tenant management is simple and no specific or exclusive resources have to be created for that tenant.

- **Regular tier**. A regular tier might be established for paying clients that have no specifically strong requirements when it comes to scaling or to isolation. For these clients the schema data management pattern or the database data management pattern might be appropriate. In both cases the tables and indexes are exclusive for the tenant. A difference is that backup is easy in case of the database data management pattern, but not supported by Cloud Spanner for the schema data management pattern — in that case a tenant-backup has to be implemented as a utility outside Cloud Spanner.

- **Enterprise tier**. The enterprise tier is usually a high end tier with full autonomy in all aspects. A tenant in that tier has dedicated resources that include dedicated scaling as well as full isolation. The instance data management pattern is well-suited for this requirement.

As a note, a best practice is to keep different data management patterns separated into different databases. For example, the schema and table data management patterns are ideally not implemented within a single database. While it is possible to combine different data management patterns in a database, the application's access logic as well as the life cycle operations are more difficult to implement.

The section "Application Design" below will outline some multi-tenant application design considerations that apply when using a single data management pattern or several data management patterns.

Tenant data life cycle management

Tenants have a life cycle and corresponding management operations have to be implemented within a multi-tenant application. Beyond the basic operations of creating, updating and deleting tenants, the following additional data-related operations should be considered:

- **Export tenant data**. When a tenant is to be deleted there might be reasons to export the tenant data first and possibly make the data set available to the tenant. Depending on the data management pattern the export has to be implemented by

the multi-tenant application system (e.g., table, schema pattern), or can be mapped to database functionality (e.g., database export).

- **Backup tenant data**. Similar to export, it might be necessary to back up data for individual tenants. When the data management pattern is instance or database, database functionality like export or backup can be used. However, in the case of the schema or table data management patterns,this operation has to be implemented by the multi-tenant application as the database functionality would not be able to determine which data belong to which tenant.

- **Move tenant data**. A very important operation is moving a tenant from one data management pattern to another (or within the same data management pattern between instances or databases). A use case is a small tenant in a table data management pattern growing to such an extent that it has to move to a database data management pattern. This requires extracting the data from the table data management pattern and inserting that data into the database data management pattern. When application downtime is possible, an export/import can be performed (see above), however, when downtime is not possible this corresponds to a zero downtime database migration in the context of Cloud Spanner. Another reason for moving tenants is rebalancing to mitigate a noisy neighbor situation.

Application design

Multi-tenant application design is about implementing tenant-aware business logic, meaning that each execution of business logic must always be in the context of a known tenant.

From a database perspective this means that each query must be executed against the data management pattern in which the tenant resides. The following discussion highlights some of the central concepts of multi-tenant application design.

Dynamic tenant connection and query configuration

A mapping configuration is typically used to dynamically map the tenant's data to the requests from the tenant application.

- In the case of instance or database data management patterns, a connection string is sufficient to access a tenant's data.

- In case of the schema data management pattern, the correct table names have to be determined.
- In case of the table data management pattern, queries have to be executed against the database with appropriate predicates to retrieve only the data of the tenant in question.

In principle, a tenant can reside in any of the four data management patterns. The following mapping implementation addresses connection configuration for the general case of a multi-tenant application making use of all data management patterns at the same time. A given tenant resides in one pattern. In some multi-tenant applications only one data management pattern is used for all tenants. This case is covered implicitly by the following mapping.

If a tenant executes business logic (e.g., an employee logging in with their tenant id) then the application logic must determine the tenant's data management pattern, the location of the data for a given tenant id, and optionally the table naming convention (for the schema pattern).

This requires a tenant-to-data-management-pattern mapping as follows:

```
tenant id -> (data management pattern,
              database connection string,
              [table_prefix])
```

The connection string refers to the database where the tenant data resides. This identifies the Cloud Spanner instance as well as the database instance. For the data management pattern instance and database this is sufficient for the application to connect and execute queries.

However, for the schema and table data management patterns additional design is required. For the schema data management pattern there are several tenants within the same database with each having its own set of tables. The tables are distinguished by their name so that it is deterministic which table belongs to a given tenant.

One approach is to prepend the table names with the tenant id. For example, the `EMPLOYEE` table is called `T356_EMPLOYEE` for the tenant with the id `356`. The application has to prepend each table with the prefix `T<tenant id>` before sending the query to the database that the mapping returned.

An alternative approach is shown in the preceding mapping. Instead of using a default naming convention, a table prefix is added to the mapping that needs to be prepended to the table names in queries to find the correct tables for a tenant.

Of course, a mixed approach is possible as well: if the pattern is schema, and the table prefix is empty, the default mapping takes place.

A similar design is required for the table data management pattern. However, in this case there is a single schema, and the tenants' data are stored as rows. In order to properly access the data a predicate has to be appended to each query that selects the appropriate tenant.

One approach is to have a column called `TENANT` in each table and the values of these columns are tenant ids. Each query has to append a predicate `AND TENANT = <tenant id>` to an existing `WHERE` clause or add a `WHERE` clause for this purpose.

To implement the connectivity to the database and to create the proper queries the tenant identifier must be available in the application logic (maybe passed in as parameter or stored as thread context).

Some life cycle operations require the modification of the tenant-to-data-management-pattern mapping configuration. For example, when a tenant is moved between data management patterns, the data management pattern and the database connection string have to be updated, and possibly the table prefix.

Query generation and attribution

A fundamental underlying principle of multi-tenant applications is the ability to share resources for tenants so that several tenants can share a single cloud resource. The data management patterns above fall into this category except for the case where a single tenant is allocated to a single Cloud Spanner instance.

The sharing of resources goes beyond the data aspect. Monitoring and logging is shared as well. For example, in case of the table data management pattern, all queries for all tenants are recorded in the same audit log. The same is true for the schema data management pattern.

If a query is logged, then the query text has to be examined to determine the tenant for which this query was executed. In case of the table data management pattern, the predicate has to be parsed, in case of the schema data management pattern one of the table names.

In order to ease analyzing logs and queries, it would be easier if the tenant for a given query could be determined without parsing the query text. In context of the database or instance data management pattern, the query text would not have the tenant information at all — the tenant-to-data-management pattern mapping table would have to be queried for this information.

In order to uniformly be able to identify the tenant for a query across all data management patterns one solution is to add a comment to the query text that has the tenant id, and maybe a label. For example,

```
select * from EMPLOYEE
-- TENANT 356
where TENANT = 'T356';
```

or

```
select * from T356_EMPLOYEE;
-- TENANT 356
```

With this design every query executed for a tenant is attributed to that tenant independent of the data management pattern used. If a tenant is moved from one data management pattern to another, the query text might change, but the attribution would be the exact same comment in the query text.

Above is only one example. Instead of a label and value, a JSON object could be inserted as comment as well:

```
select * from T356_EMPLOYEE;
-- {"TENANT": 356}
```

Tenant access life cycle operations

Depending on the design philosophy, a multi-tenant application can implement the data life cycle operations as described earlier directly, or a separate tenant administration tool can be created.

Independent of the implementation strategy of data life cycle operations, an important aspect is that life cycle operations might have to be executed exclusively without the application logic executing at the same time. For example, while moving a tenant from one data management pattern to another, the application logic can't execute because the data is not in a single database. This requires two additional operations from an application perspective:

- **Stopping a tenant**. This life cycle operation disables all application logic access while permitting data life cycle operations.
- **Starting a tenant**. This life cycle operation implements the opposite constraints: application logic can access a tenant's data while those life cycle operations are disabled that would otherwise interfere with the application logic.

While not very often used, an emergency tenant shutdown might be another important life cycle operation when all access to a tenant's data needs to be prohibited, not only application logic, but life cycle operations as well, in case a breach is suspected. And a breach can originate at the outside or the inside.

A matching life cycle operation that removes the emergency status must be available as well, possibly requiring two or more administrators to login at the same time in order to implement mutual control.

Application isolation

The various data management patterns support different degrees of isolation of tenant data. From the most isolated level (instance) to the least isolated level (table) different degrees of isolation are possible.

In the context of a multi-tenant application a similar deployment decision has to be made: do all tenants access their data (in possibly different data management patterns) using the same application deployment? For example, a single Kubernetes cluster could support all tenants and when a tenant accesses its data, the same cluster is executing the business logic.

Alternatively, as in the case of the data management patterns, different tenants could be directed to different application deployments. Large tenants could have access to an application deployment exclusive to them, whereas smaller tenants or tenants in the free tier share an application deployment.

Even though it is tempting to directly match the (data) data management patterns with equivalent application data management patterns, it does not have to be this way. It is possible to have the database data management pattern and all these tenants share a single application deployment.

Summary

Multi-tenancy is an important application design data management pattern, especially when resource efficiency plays a vital role. Cloud Spanner supports several data management patterns and can easily be used for implementing multi-tenant applications. With Cloud Spanner's extreme scalability and super strict SLAs, it is an ideal database for large multi-tenant application deployments.

Disclaimer

Christoph Bussler is a Solutions Architect and Sireesha Pulipati is a Data Management Customer Engineer at Google, Inc. (Google Cloud). The opinions stated here are our own, not those of Google, Inc.

Cloud Spanner | Multitenancy | Database | Google Cloud Platform