Implementing multi-tenancy in Cloud Spanner

This document describes various ways to implement multi-tenancy in Cloud Spanner (/spanner). It also discusses data management patterns and tenant lifecycle management.

Multi-tenancy (https://wikipedia.org/wiki/Multitenancy) is when a single instance, or a few instances, of a software application serves multiple tenants or customers. This software pattern can scale from a single tenant or customer to hundreds or thousands. This approach is fundamental to cloud computing platforms where the underlying infrastructure is shared among multiple organizations.

Think of multi-tenancy 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 part of most, if not all, software-as-a-service (SaaS) applications.

This document is for database architects, data architects, and engineers that implement multi-tenant applications on Spanner as their relational database. Using that context, it outlines various approaches to store multi-tenant data. The terms "tenant", "customer", and "organization" are used interchangeably throughout the article to indicate the entity that's accessing the multi-tenant application.

This article uses a human-resources (HR) SaaS provider implementing its multi-tenant application on Google Cloud as an example. In the example, several customers of the HR SaaS provider must access the multi-tenant application. These customers are called tenants.

Spanner (/spanner) is Google Cloud's fully managed, enterprise-grade, distributed, and consistent database which combines the benefits of the relational database model with non-relational horizontal scalability. Spanner has relational semantics—with schemas, enforced data types, strong consistency, multi-statement ACID transactions, and a SQL query language implementing ANSI 2011 SQL.

Spanner provides zero-downtime for planned maintenance or region failures, with an availability SLA of 99.999% (/spanner/sla). It supports modern, multi-tenant applications by providing high availability and scalability. This article discusses the different architecture approaches to implement multi-tenancy with Spanner.

Criteria for tenant data-mapping criteria

In a multi-tenant application, each tenant's data is isolated in one of several architecture approaches in the underlying Spanner database. The following list outlines the different architecture approaches used to map a tenant's data to Spanner:

- **Instance:** A tenant resides exclusively in one Spanner instance, with exactly one database for that tenant.
- **Database:** A tenant resides in a database in a single Spanner instance containing multiple databases.
- **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 database tables. Those tables are shared with other tenants.

The preceding criteria are called data management patterns and are discussed in detail in the Multi-tenancy data management patterns (#multi-tenancy-data-management-patterns) section. That discussion is based on the following criteria:

- **Isolation:** The degree of data isolation across multiple tenants is a major consideration for multi-tenancy. Isolation is driven by the choices made for the criteria under other categories—for example, certain regulatory and compliance requirements can dictate a greater degree of isolation.
- **Agility:** The ease of onboarding and offboarding activities for a tenant with respect to creating an instance, database, or table.
- **Operations:** The availability or complexity of implementing typical, tenant-specific, database operations and administration activities—for example, regular maintenance, logging, backups, or disaster recovery operations.
- **Scale:** The ability to scale seamlessly to allow for future growth. The description of each pattern contains the number of tenants the pattern can support.
- **Performance:** The ability to allocate exclusive resources to each tenant, address the noisy neighbor (https://wikipedia.org/wiki/Cloud_computing_issues#Performance_interference_and_noisy_neighbors) phenomenon, and enable predictable read and write performance for each tenant.

- **Regulations and compliance:** The ability to address the requirements of highly regulated industries and countries that require the complete isolation of resources and maintenance operations—for example, data residency requirements for France require that personally identifiable information is physically stored exclusively within France.

Each data management pattern as it relates to these criteria is detailed in the next section. Use the same criteria when selecting a data management pattern for a specific set of tenants.

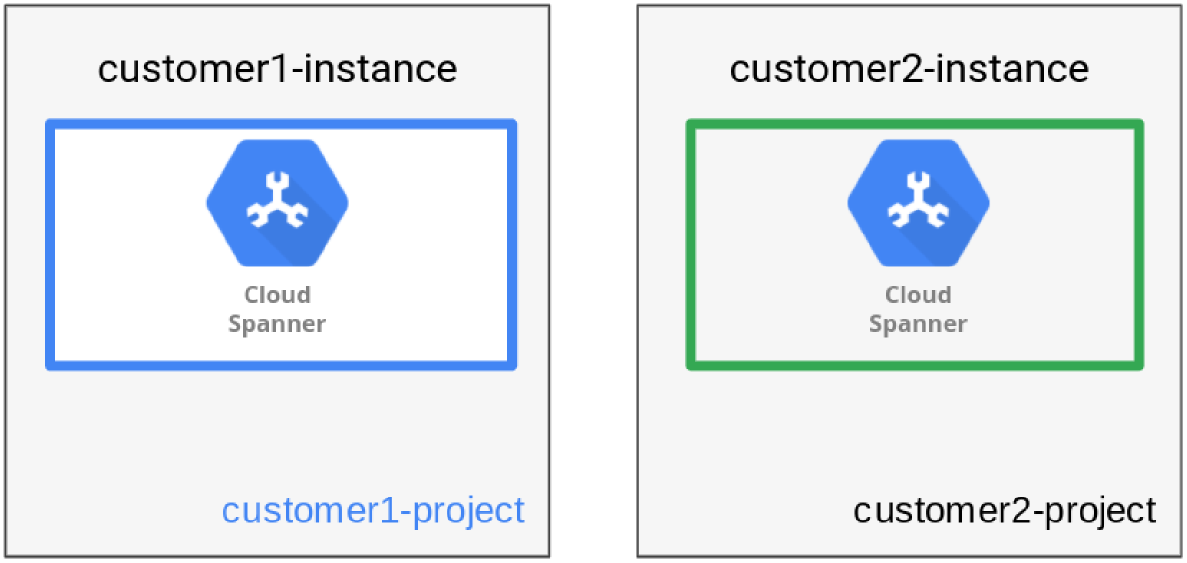
Multi-tenancy data management patterns

The following sections describe the four main data management patterns: instance, database, schema, and table.

Instance

To provide complete isolation, the instance data management pattern stores each tenant's data in its own Spanner instance and database. A Spanner instance can have one or more databases. In this pattern, only one database is created. For the HR application discussed earlier, a separate Spanner instance with one database is created for each customer organization.

As seen in the following diagram, the data management pattern has 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 extra benefit is that each instance configuration can be chosen based on each tenant's location (either regionally or multi-regionally), optimizing location flexibility and performance.

The architecture can easily scale to any number of tenants. SaaS providers can create any number of instances in the desired regions, without any hard limits.

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

Criteria	**Instance — one tenant per instance data management pattern**
Isolation	• Greatest level of isolation • No database resources are shared
Agility	• Onboarding and offboarding require considerable setup or decommissioning for: • The Spanner instance • Instance-specific security • Instance-specific connectivity • Onboarding and offboarding can be automated through Infrastructure as Code (IaC) (/solutions/infrastructure-as-code)
Operations	• Independent backups for each tenant • Separate and flexible backup schedules • Higher operational overhead • Large number of instances to manage and maintain (scaling, monitoring, logging, and performance tuning)
Scale	• Highly scalable database • Unlimited growth by adding nodes • Unlimited number of tenants • Spanner instance available for each tenant

Criteria	Instance — one tenant per instance data management pattern
Performance	• Each tenant in a separate instance • No resource contention • Tailored performance tuning and troubleshooting for each tenant
Regulatory and compliance requirements	• Store data in a specific region • Implement specific security, backup, or auditing processes as required by businesses or governments

In summary, the key takeaways are:

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

The instance data management pattern is best suited for 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.
- Tenant size varies significantly, such that sharing resources among high-volume, high-traffic tenants might cause contention and mutual degradation.

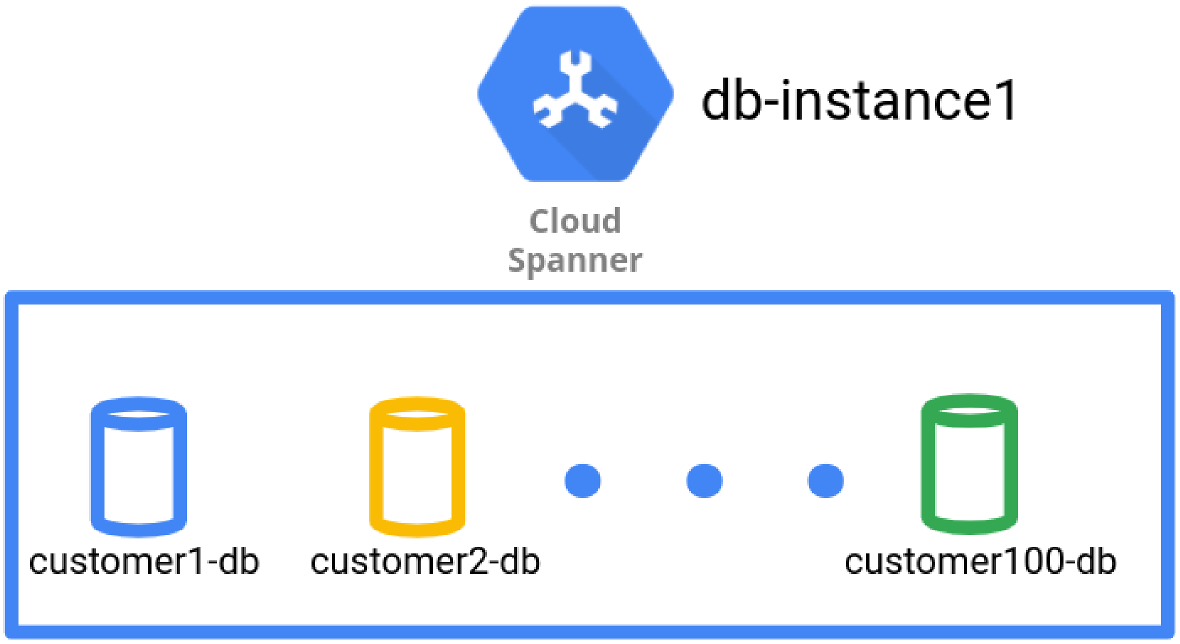
Database

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

Spanner has a hard limit (/spanner/quotas#database_limits) of 100 databases per instance. This limit means that if the SaaS provider needs to scale beyond 100 customers, they need to create and use multiple Spanner instances.

For the HR application, the SaaS provider creates and manages each tenant with a separate database in a Spanner instance.

As seen in the following diagram, the data management pattern has one tenant per database.



The database data management pattern achieves logical isolation on a database level for different tenants' data. However, because it's a single Spanner instance, all the tenant databases share the same regional configuration and underlying compute and storage setup.

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

Criteria	**Database — one tenant per database data management pattern**
Isolation	• Complete logical isolation on the database level • Shared underlying infrastructure resources
Agility	• Requires effort to create or delete the database and any specific security controls • Automation for onboarding and offboarding comes through IaC
Operations	• Independent backups for each tenant • Flexible scheduling • Less operational overhead compared to the instance pattern ◦ One instance to monitor for up to 100 databases

Criteria	Database — one tenant per database data management pattern
Scale	• Highly scalable database • Unlimited instances • Allows thousands of nodes • Limit of 100 databases per instance • For every 100 tenants, create a new Spanner instance
Performance	• Resource contention among multiple databases • Databases spread across Spanner instance nodes • Databases share infrastructure • Noisy neighbors affect performance
Regulatory and compliance requirements	• Location region must match the instance region to meet any specific data residency regulatory requirements

In summary, the key takeaways are:

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

The database data management pattern is best suited for the following scenarios:

- Multiple customers are in the same data residency—for example, France, or the UK—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

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

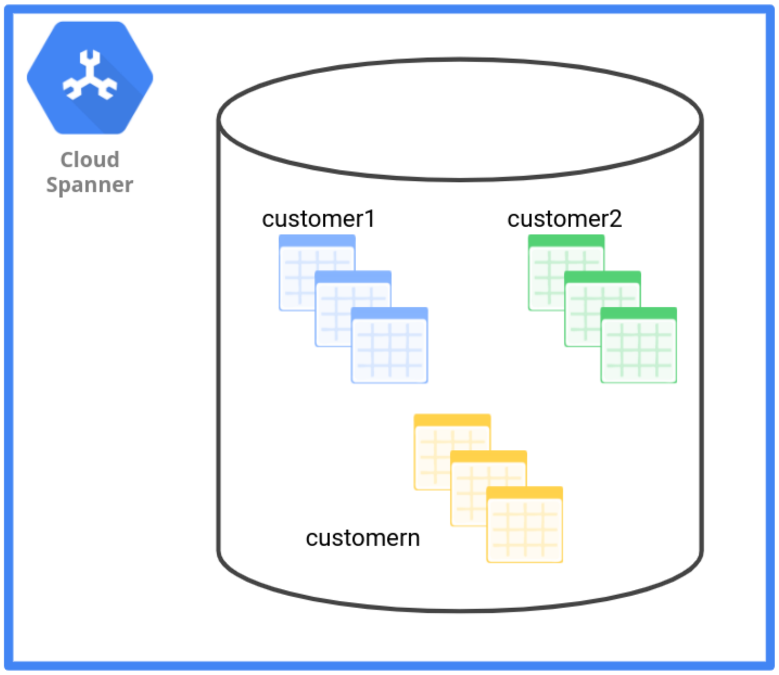
This data management pattern of using a separate set of tables for each tenant provides a much lower level of isolation compared to the preceding options (the instance and database management patterns). The pattern also makes onboarding simple—it involves creating new tables and associated referential integrity and indexes.

One significant caveat is that access permissions for Spanner through Identity and Access Management (IAM) are only provided at the instance or database level. Access permissions can't be provided at the table level. There's also a limit of 5,000 tables per database. For many customers, that limit restricts their use of the application.

Furthermore, using separate tables for each customer can result in a large backlog of schema update operations. Such a backlog takes a long time to resolve (/spanner/docs/schema-updates).

For the HR application, the SaaS provider can create a set of tables for each customer with `tenant ID` as the prefix in the table names—for example, `customer1_employee`, `customer1_payroll`, `customer1_department`.

As seen in the following diagram, the schema data management pattern has one set of tables for each tenant.



The following table outlines how the schema 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 trivial ◦ Create new tables ◦ Create associated keys and indexes • Offboarding a customer means deleting tables ◦ May have a temporary negative performance impact on other tenants within the database
Operations	• No separate operations for tenants • Backup, monitoring, and logging must be implemented as separate application functions or as utility scripts
Scale	• Thousands of nodes • Unlimited tenant growth • A single database can only have 5,000 tables ◦ Only floor (5,000/<number tables for tenant>) number of tenants in each database ◦ When the database exceeds 5,000 tables, add a new database for the additional tenants
Performance	• High level of resource contention is possible • Ensure good performance by separately designing indexes for each set of tables
Regulatory and compliance requirements	• Location region must match to meet any specific data residency regulatory requirements • Implementing specific security and auditing controls affects all tenants residing in the same database

In summary, the key takeaways are:

- **Advantage:** Onboarding is trivial

- **Disadvantage:** Higher operational overhead; no security controls at the table level

The schema data management pattern is best suited for the following scenarios:

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

While it's possible to create several sets of tables (each set representing a tenant) in a database, it's the least ideal pattern from a database perspective. The main reason is that the tables must follow naming conventions. The application, and any database tooling (for example, IDE, and schema migration tools), must understand the naming convention. Also, if the number of tables is reasonably large per tenant, the schema data management pattern doesn't provide significant scaling.

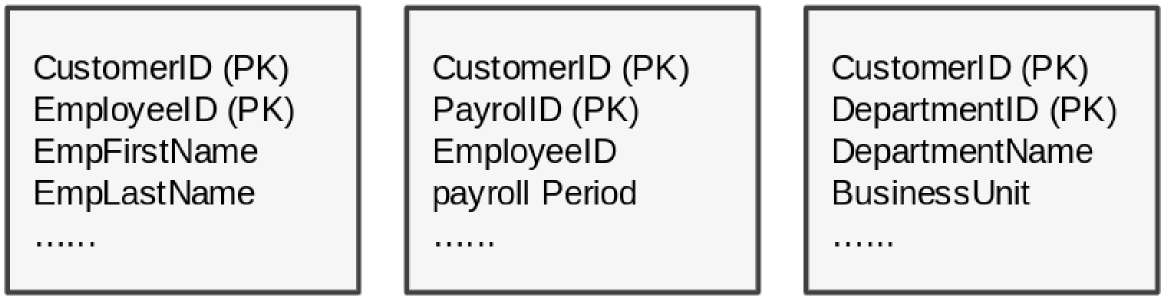
A better approach is to move to either one database per tenant and increase the number of instances, or move to the table data management pattern.

Table

The final data management pattern serves 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 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 scaling perspective, Spanner supports this pattern best because it can scale tables without limitation.

For the HR application, the payroll table's primary key can be a combination of `customerID` and `payrollID`.

As seen in the following diagram, the table data management pattern has one table for several tenants.



Similar to the schema pattern, data access in the table pattern can't be controlled separately for different tenants. Using fewer tables means schema update operations complete faster when each tenant has their own database tables. To a large extent, this approach simplifies onboarding, offboarding, and operations.

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

Criteria	**Table — one table for several tenants data management pattern**
Isolation	• Lowest level of isolation • No tenant level security
Agility	• No setup required on the database side when onboarding ◦ The application can directly write data into the existing tables • Offboarding means deleting the customer's rows in the table
Operations	• No separate operations for tenants, including backup, monitoring, and logging • Little to no overhead as the number of tenants increases
Scale	• Scales to thousands of nodes • Can accommodate any level of tenant growth • Supports an unlimited number of tenants

Criteria	Table — one table for several tenants data management pattern
Performance	• Pattern works well in the context of Spanner • Internal distributed storage, processing, and load balancing can easily work with this pattern • If primary key spaces are not designed carefully, a high level of resource contention is possible (noisy neighbor) ◦ Can prevent concurrency and distribution • Following best practices is important • Deleting a tenant's data might have a temporary impact on the load
Regulatory and compliance requirements	• Location must match to meet any specific data residency regulatory requirements • Pattern can't provide system-level partitioning (compared to the instance or database pattern) • Implementing any specific security and auditing controls affects all tenants

In summary, the key takeaways are:

- **Advantage:** Highly scalable; has low operational overhead
- **Disadvantage:** High resource contention; lack of security controls for each tenant

This pattern is best suited for the following scenarios:

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

Data management patterns and tenant lifecycle management

The following table compares the various data management patterns across all criteria at a high level.

	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
Regulations and compliance	Highest	High	Low	Low

* Performance is heavily dependent on the schema design (/spanner/docs/schema-design) and query best practices (/spanner/docs/sql-best-practices). The values here are only an average expectation.

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 isn't required, you can ignore the row it's in.

Combined data management patterns

Often, a single data management pattern is sufficient to address the requirements of a multi-tenant application. When that's the case, the design can assume a single data management pattern.

Some multi-tenant applications require several data management patterns at the same time, however—for example, a multi-tenant application that supports a free tier, a regular tier, and an enterprise tier.

- **Free tier:**
    - Must be cost effective
    - Must have an upper data-volume limit
    - Usually supports limited functionality
    - The table data management pattern is a good free-tier candidate
        - Tenant management is simple

    - No need to create specific or exclusive tenant resources
- **Regular tier:**
  - Good for paying clients who have no specifically strong scaling or isolation requirements
  - The schema data management pattern or the database data management pattern is a good regular-tier candidate
    - Tables and indexes are exclusive for the tenant
    - Backup is easy in the database data management pattern
    - Backup isn't supported for the schema data management pattern
      - Tenant backup must be implemented as a utility outside Spanner
- **Enterprise tier:**
  - Usually a high-end tier with full autonomy in all aspects
  - Tenant has dedicated resources that include dedicated scaling and full isolation
  - The instance data management pattern is well suited for the enterprise tier

A best practice is to keep different data management patterns in different databases. While it's possible to combine different data management patterns in a Spanner database, doing so makes it difficult to implement the application's access logic and lifecycle operations.

The Application design (#application-design) section outlines some multi-tenant application design considerations that apply when using a single data management pattern or several data management patterns.

Manage the tenant lifecycle

Tenants have a life cycle. Therefore, you must implement the corresponding management operations within your multi-tenant application. Beyond the basic operations of creating, updating, and deleting tenants, consider the following additional data-related operations:

- **Export tenant data:**
  - When deleting a tenant, it's a best practice to export their data first and possibly make the dataset available to them.

  - When using the table or schema data management pattern, the multi-tenant application system must implement the export or map it to the database functionality (database export).
- **Back up tenant data:**
  - When using the instance or database data management pattern and backing up data for individual tenants, use the database's export or backup functions.
  - When using the schema or table data management pattern and backing up data for individual tenants, the multi-tenant application must implement this operation. The Spanner database can't determine which data belongs to which tenant.
- **Move tenant data:**
  - Moving a tenant from one data management pattern to another (or moving a tenant within the same data management pattern between instances or databases) 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, perform an export/import.
    - When downtime isn't possible, perform a zero downtime database migration (https://medium.com/google-cloud/zero-downtime-database-migration-and-replication-to-and-from-cloud-spanner-99ad0c654d12)
.
  - Mitigating a noisy-neighbor situation is another reason to move tenants.

Application design

When designing a multi-tenant application, implement tenant-aware business logic. That means each time the application runs business logic, it must always be in the context of a known tenant.

From a database perspective, application design means that each query must be run against the data management pattern in which the tenant resides. The following sections highlight some of the central concepts of multi-tenant application design.

Dynamic tenant connection and query configuration

Dynamically mapping tenant data to tenant application requests uses a mapping configuration:

- For database data management patterns or instance data management patterns, a connection string is sufficient to access a tenant's data.
- For schema data management patterns, the correct table names have to be determined.
- For table data management patterns, queries have to be executed against the database. Use the appropriate predicates to retrieve a specific tenant's data.

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

If a tenant executes business logic (for example, 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 application logic requires tenant-to-data-management pattern mapping. In the following code sample, the `connection string` refers to the database where the tenant data resides. The sample identifies the Spanner instance and the database. For the data management pattern instance and database, the following code is sufficient for the application to connect and execute queries:

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

Additional design is required for the schema and table data management patterns.

Schema data management pattern

For the schema data management pattern, there are several tenants within the same database. Each tenant has its own set of tables. The tables are distinguished by their name. Which table belongs to which tenant is deterministic.

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.

Another approach is to prepend a `table_prefix` to the mapping used by the query so it finds the correct tables for a tenant.

A mixed approach is possible as well: if the data management pattern is the schema pattern, and the table prefix is empty, the default mapping takes place (prepend table names with tenant IDs).

Table data management pattern

A similar design is required for the table data management pattern. In this pattern, there's a single schema. Tenant data are stored as rows. To properly access the data, append a predicate to each query to select the appropriate tenant.

One approach to find the appropriate tenant is to have a column called `TENANT` in each table. The column value is `tenant ID`. Each query must append a predicate `AND TENANT = `***`tenant ID`*** to an existing `WHERE` clause or add a `WHERE` clause with the predicate `AND TENANT = `***`tenant ID`***.

To connect to the database and to create the proper queries, the tenant identifier must be available in the application logic. It can be passed in as parameter or stored as thread context.

Some lifecycle operations require you to modify the tenant-to-data-management-pattern mapping configuration—for example, when you move a tenant between data management patterns, you must update the data management pattern and the database connection string. You might also have to update the table prefix.

Query generation and attribution

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

The sharing of resources goes beyond sharing data. Monitoring and logging is also shared—for example, in the table data management pattern and schema data management pattern, all queries for all tenants are recorded in the same audit log.

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

In the database data management pattern or the instance data management pattern, the query text doesn't have any tenant information. To get tenant information for these patterns, you must query the tenant-to-data-management-pattern mapping table.

It would be easier to analyze logs and queries by determining the tenant for a given query without parsing the query text. One way to uniformly identify a tenant for a query across all data management patterns is to add a comment to the query text that has the **`tenant ID`**, and (optionally) a **`label`**.

The following query selects all employee data for the tenant identified by **`TENANT 356`**. To avoid parsing the SQL syntax and extracting the tenant ID from the predicate, the tenant ID is added as a comment. A comment can be extracted without having to parse the SQL syntax.

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

or

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

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

The preceding code sample is only one method. Another method is to insert a JSON (https://www.json.org/) object as a comment instead of a label and value:

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

Tenant access lifecycle operations

Depending on your design philosophy, a multi-tenant application can directly implement the data lifecycle operations described earlier, or it can create a separate tenant-administration tool.

Independent of the implementation strategy, lifecycle operations might have to be run without the application logic running at the same time—for example, while moving a tenant from one data management pattern to another, the application logic can't run because the data isn't in a single database. When data isn't in a single database, it requires two additional operations from an application perspective:

- **Stopping a tenant:** Disables all application logic access while permitting data lifecycle operations.
- **Starting a tenant:** Application logic can access a tenant's data while the lifecycle operations that would interfere with the application logic are disabled.

While not often used, an emergency tenant shutdown might be another important lifecycle operation. Use this shutdown when you suspect a breach, and you need to prohibit all access to a tenant's data—not only application logic, but lifecycle operations as well. A breach can originate from inside or outside the database.

A matching lifecycle operation that removes the emergency status must also be available. Such an operation can require two or more administrators to log in at the same time in order to implement mutual control (https://wikipedia.org/wiki/Mutual_authentication).

Application isolation

The various data management patterns support different degrees of tenant-data isolation. 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 must 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 might support all tenants and when a tenant accesses its data, the same cluster runs the business logic.

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

Instead of directly matching the data management patterns discussed in this article with equivalent application-data management patterns, you can use the database data management pattern so that all tenants share a single application deployment. It's possible to have the database data management pattern and all these tenants share a single application deployment.

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

What's next

- Try out other Google Cloud features for yourself. Have a look at our tutorials (/docs/tutorials).