SaaS Storage Strategies

Building a Multitenant Storage Model on AWS

November 2016

Notices

This document is provided for informational purposes only. It represents AWS's current product offerings and practices as of the date of issue of this document, which are subject to change without notice. Customers are responsible for making their own independent assessment of the information in this document and any use of AWS's products or services, each of which is provided "as is" without warranty of any kind, whether express or implied. This document does not create any warranties, representations, contractual commitments, conditions or assurances from AWS, its affiliates, suppliers or licensors. The responsibilities and liabilities of AWS to its customers are controlled by AWS agreements, and this document is not part of, nor does it modify, any agreement between AWS and its customers.

Contents

Abstract

Multitenant storage represents one of the more challenging aspects of building and delivering software-as-a-service (SaaS) solutions. There are a variety of strategies that can be used to partition tenant data, each with a unique set of nuances that shape your approach to multitenancy. Adding to this complexity is the need to map each of these strategies to the different storage models offered by AWS, such as Amazon DynamoDB, Amazon Relational Database Service (Amazon RDS), and Amazon Redshift. Although there are high-level themes you can apply universally to these technologies, each storage model has its own approach to scoping, managing, and securing data in a multitenant environment. This paper offers SaaS developers insights into a range of data partitioning options, allowing them to determine which combination of strategies and storage technologies best align with the needs of their SaaS environment.

Introduction

AWS offers software-as-a-service (SaaS) developers a rich collection of storage solutions, each with its own approach to scoping, provisioning, managing, and securing data. The way that each service represents, indexes, and stores data adds a unique set of considerations to your multitenant strategy. As a SaaS developer, the diversity of these storage options represents an opportunity to align the storage needs of your SaaS solution with the storage technologies that best match your business and customer needs.

As you weigh AWS storage options, you must also consider how the multitenant model of your SaaS solution fits with each storage technology. Just as there are multiple flavors of storage, there are also multiple flavors of multitenant partition strategies. The goal is to find the best intersection of your storage and tenant partitioning needs.

This paper explores all the moving parts of this puzzle. It examines and classifies the models that are typically used to achieve multitenancy, and helps you weigh the pros and cons that shape your selection of a partitioning model. It also outlines how each model is realized on Amazon RDS, Amazon DynamoDB, and Amazon Redshift. As you dig into each storage technology, you'll learn how to use the AWS constructs to scope and manage your multitenant storage.

Although this paper gives you general guidance for selecting a multitenant partitioning strategy, it's important to recognize that the business, technical, and operational dimensions of your environment will often introduce factors that will also shape the approach you select. In many cases, SaaS organizations adopt a hybrid of the variations described in this paper.

SaaS Partitioning Models

To get started, you need a well-defined conceptual model to help you understand the various implementation strategies.

Figure 1 shows the three basic models—silo, bridge, and pool—that are commonly used when partitioning tenant data in a SaaS environment.

Each partitioning model takes a very different approach to managing, accessing, and separating tenant data. The following sections give a quick breakdown of the models, allowing you to explore the values and tenets of each model outside of the context of any specific storage technology.

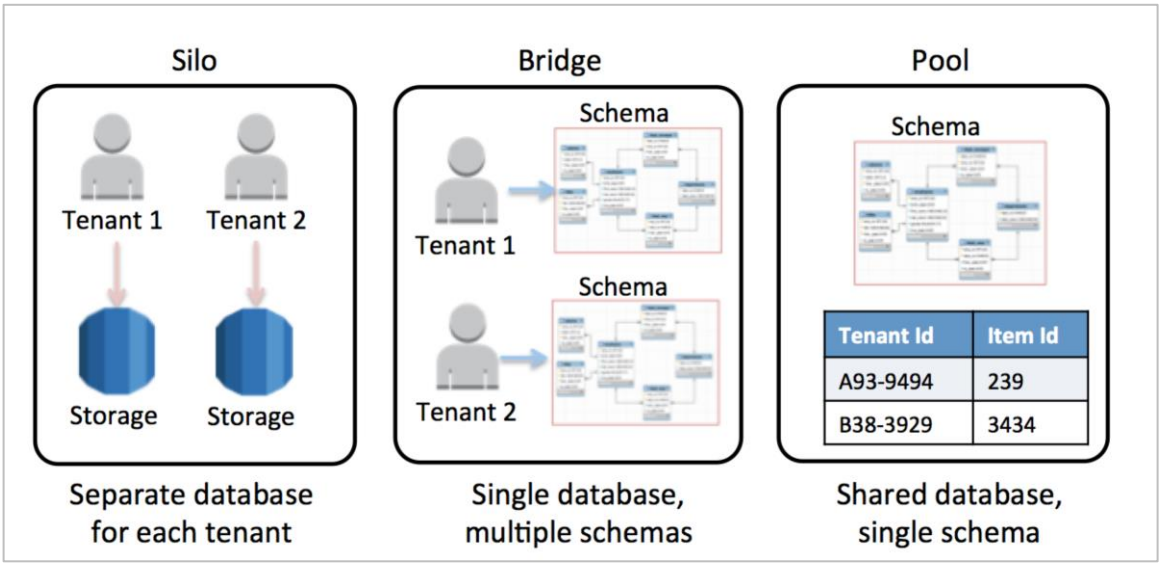

Figure 1: SaaS partitioning models

Silo Model

In the silo model, storage of tenant data is fully isolated from any other tenant data. All constructs that are used to represent the tenant's data are considered logically "unique" to that client, meaning that each tenant will generally have a distinct representation, monitoring, management, and security footprint.

Bridge Model

The bridge model often represents an appealing compromise for SaaS developers. Bridge moves all of the tenant's data into a single database, while still allowing some degree of variation and separation for each tenant. Typically, you achieve this by creating separate tables for each tenant and allow, each of which is allowed table to have its own representation of data (schema).

Pool Model

The pool model represents the all-in, multitenant model where tenants share all of the system's storage constructs. Tenant data is placed into a common database and all tenants share a common representation (schema). This requires the introduction of a partitioning key that is used to scope and control access to tenant data. This model tends to simplify a SaaS solution's provisioning, management, and update experience. It also fits well with the continuous delivery and agility goals that are essential to SaaS providers.

Setting the Backdrop

The silo, bridge, and pool models provide the backdrop for our discussion. As you dig into each AWS storage technology, you'll discover how the conceptual elements of these models are realized on a specific AWS storage technology. Some map very directly to these models; others require a bit more creativity to achieve each type of tenant isolation.

It's worth noting that these models are all equally valid. Although we'll discuss the merits of each, the regulatory, business, and legacy dimensions of a given environment often play a big role in shaping the approach you ultimately select. The goal here is to simply bring visibility to the mechanics and tradeoffs associated with each approach.

Finding the Right Fit

Selecting a multitenant partitioning storage model strategy is influenced by many different factors. If you are migrating from an existing solution, you might favor adopting a silo model because it creates the simplest and cleanest way to transition to multitenancy without rewriting your SaaS application. If you have regulatory or industry dynamics that demand a more isolated model, the efficiency and agility of the pool model might unlock your path to an environment that embraces rapid and continual releases. The key here is to acknowledge that the strategy you select will be driven by a combination of the business and technical considerations in your environment.

In the following sections, we highlight the strengths and weaknesses of each model and provide you with a well-defined set of data points to use as part of your broader assessment. You'll learn how each model influences your ability to

align with the agility goals that are often at the core of adopting a SaaS model. When selecting an architectural strategy for your SaaS environment, consider how that strategy impacts your ability to rapidly build, deliver, and deploy versions in a zero downtime environment.

Assessing Tradeoffs

If you were to put the three partitioning models—silo, bridge, and pool—on a spectrum, you'd see the natural tensions associated with adopting any one of these strategies. The qualities that are listed as strengths for one model are often represented as weaknesses in another model. For example, the tenets and value system of the silo model are often in opposition to those of the pool model.

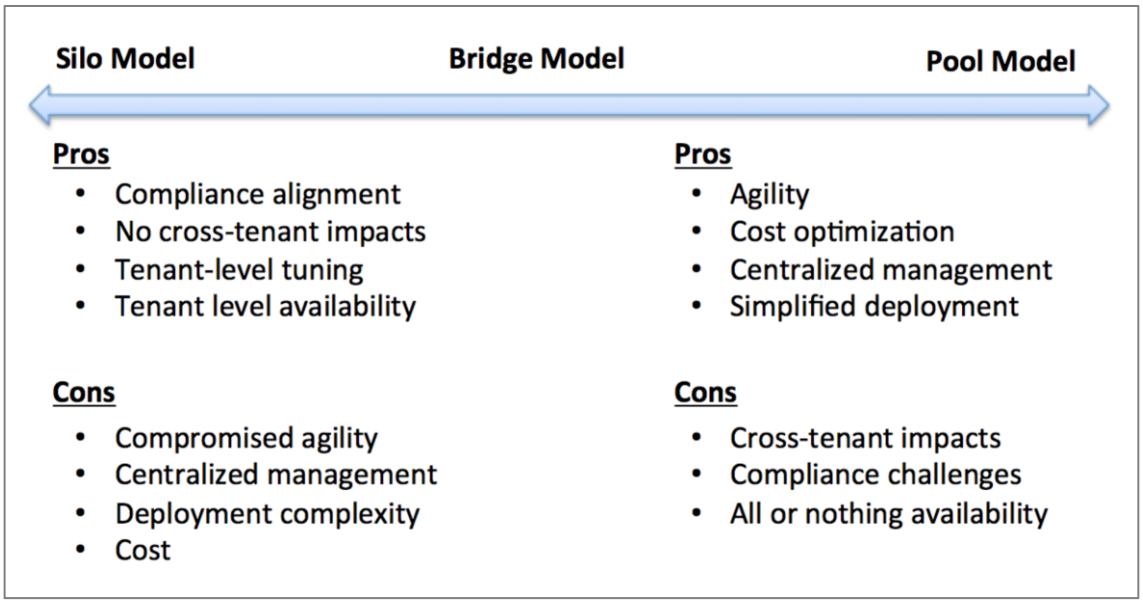

Figure 2: Partitioning model tradeoffs

Figure 2 highlights these competing tenets. Across the top of the diagram you'll see the three partitioning models represented. On the left are the pros and cons associated with the silo model. On the right, we provide similar lists for the pool model. The bridge model is a bit of a hybrid of these considerations and, as such, represents a mix of the pros and cons shown at the extremes.

Silo Model Tradeoffs

Representing tenant data in completely separate databases can be appealing. In addition to simplifying migration of existing single-tenant solutions, this approach also addresses concerns some tenants might have about operating a fully shared infrastructure.

Pros

Silo is appealing for SaaS solutions that have strict regulatory and security constraints. In these environments, your SaaS customers have very specific expectations about how their data must be isolated from other tenants. The silo model lets you offer your tenants an option to create a more concrete boundary between tenant data, and provides your customers with a sense that their data is stored in a more dedicated model.

Cross-tenant impacts can be limited. The idea here is that, via the isolation of the silo model, you can ensure that the activity of one tenant does not impact another tenant. This model allows for tenant-specific tuning, where the database performance SLAs of your system can be tailored to the needs of a given tenant. The knobs and dials that are used to tune the database also generally have a more natural mapping to the silo model, which makes it simpler to configure a tenant-centric experience.

Availability is managed at the tenant level, minimizing tenant exposure to outages. With each tenant in their own database, you don't have to be concerned that a database outage might cascade across all of your tenants. If one tenant has data issues, they are unlikely to adversely impact any of the other tenants of the system.

Cons

Provisioning and management is more complex. Any time you introduce a per-tenant piece of infrastructure, you're also introducing another moving part that must be provisioned and managed on a tenant-by-tenant basis. You can imagine, for example, how a siloed database solution might impact the tenant onboarding experience for your system. Your signup process will require automation that creates and configures a database during the onboarding process. It's certainly achievable, but it adds a layer of complexity and a potential point of failure in your SaaS environment.

Your ability to view and react to tenant activity is undermined. With SaaS, you might want a management and monitoring experience that provides a cross-tenant view of system health. You want to proactively anticipate database performance issues and react with policies in a more holistic way. However, the silo model makes you work harder to find and introduce tooling to create an aggregated, system-wide view of health that spans all tenants.

The distributed nature of a silo model impacts your ability to effectively analyze and assess performance trends across tenants. With each tenant storing data in its own silo, you can only manage and tune service loads in a tenant-centric model. This essentially leads to the introduction of a set of one-off settings and policies that you have to manage and tune independently. This can be both inefficient and could impose overhead that undermines your ability to respond quickly to customer needs.

Silo limits cost optimization. Perhaps the most significant downside, the one-off nature of the silo model tends to limit your ability to tune your consumption of storage resources.

Pool Model Tradeoffs

The pool model represents the ultimate all-in commitment to the SaaS lifestyle. With the pool model, your focus is squarely on having a unified approach to your tenants that lets you streamline tenant storage provisioning, migration, management, and monitoring.

Pros

Agility. Once all of your tenant data is centralized in one storage construct, you are in a much better position to create tooling and a lifecycle that supports a streamlined, universal approach to rapidly deploying storage solutions for all of your tenants. This agility also extends to your onboarding process. With the pool model, you don't need to provision separate storage infrastructure for each tenant that signs up for your SaaS service. You can simply provision your new tenant and use that tenant's ID as the index to access the tenant's data from the shared storage model used by all of your tenants.

Storage monitoring and management is simpler. In the pool model, it's much more natural to put tooling and aggregated analytics into place to summarize tenant storage activity. The everyday tools you'd use to manage a

single storage model can be leveraged here to build a comprehensive, cross-tenant view of your system's health. With the pool model, you are in a much better position to introduce global policies that can be used to proactively respond to system events. Generally, the unification of data into a single database and shared representation simplifies many aspects of the multitenant storage, deployment, and management experience.

Additional options help optimize the cost footprint of your SaaS solutions. The costs opportunities often show up in the form of performance tuning. You might, for example, have throughput optimization that is applied across all tenants as one policy (instead of managing separate policies on a tenant-by-tenant basis).

Pool improves deployment automation and operational agility. The shared nature of the pool model generally reduces the overall complexity of your database deployment automation, which aligns nicely with the SaaS demand for continual and frequent releases of new product capabilities.

Cons

Agility means a higher bar for managing scale and availability. Imagine the impact of a storage outage in a pooled multitenant environment. Now, instead of having one customer down, *all* of your customers are down. This is why organizations that adopt a pool model also tend to invest much more heavily in the automation and testing of their environments. A pooled solution demands proactive monitoring solutions and robust versioning, data, and schema migration. Releases must go smoothly and tenant issues need to be captured and surfaced efficiently.

Pool challenges management of tenant data distribution. In some instances, the size and distribution of tenant data can also become a challenge with pooled storage. Tenants tend to impose widely varying levels of load on your system and these variations can undermine your storage performance. The pool model requires more thought about the mechanisms that you will employ to account for these variations in tenant load. The size and distribution of data can also influence how you approach data migration. These issues are typically unique to a given storage technology and need to be addressed on a case-by-case basis.

The shared nature of the pooled environment can meet resistance in some domains. For some SaaS products, customers will demand a silo model to address their regulatory and internal data protection requirements.

Hybrid: The Business Compromise

For many organizations, the choice of a strategy is not as simple as selecting the silo, bridge, or pool model. Your tenants and your business are going to have a significant influence on how you approach selection of a storage strategy.

In some cases, a team might identify a small collection of their tenants that require the silo or bridge model. Once they've made this determination, they assume that they have to implement all of the storage with that model. This artificially limits your ability to embrace those tenants that may be open to a pool model. In fact, it may add cost or complexity for a tier of tenants that aren't demanding the attributes of the silo or bridge model.

One possible compromise is to build a solution that fully supports pooled storage as your foundation. Then, you can carve out a separate database for those tenants that demand a siloed storage solution. Figure 3 provides an example of this approach in action.

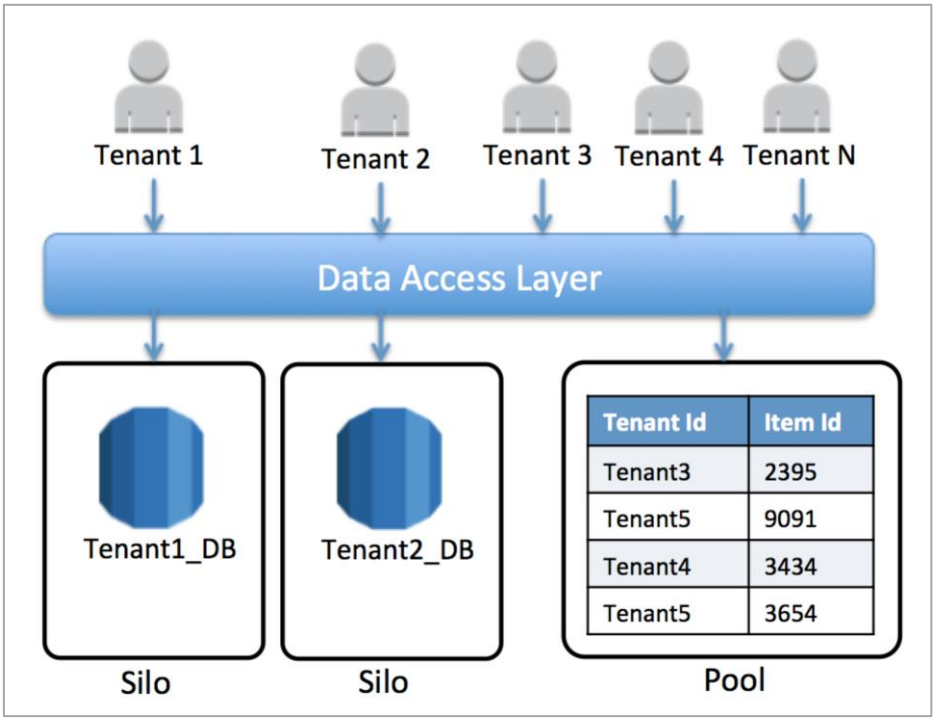

Figure 3: Hybrid silo/pool storage

Here, we have two tenants (Tenant 1 and Tenant 2) that are leveraging a silo model, and the remaining tenants are running in a pooled storage model. This is magically abstracted away by a data access layer that hides developers from the tenant's underlying storage.

Although this can add a level of complexity to your data access layer and management profile, it can also offer your business a way to tier your offering to represent the best of both worlds.

Data Migration

Data migration is one of those areas that is often left out of the evaluation of competing SaaS storage models. However, with SaaS, consider how your architectural choices will influence your ability to continually deploy new features and capabilities. Although performance and general tenant experience are important to emphasize, it's also essential to consider how your storage solution will accommodate ongoing changes in the underlying representation of your data.

Migration and Multitenancy

Each of the multitenant storage models requires its own unique approach to tackling data migration. In the silo and bridge models, you can migrate data on a tenant-by-tenant basis. Your organization may find this appealing because it allows you to carefully migrate each SaaS tenant without exposing all tenants to the possibility of a migration error. However, this approach can introduce more complexity into the overall orchestration of your deployment lifecycle.

Migrating data in the pool model can be both appealing and challenging. In one respect, migration in a pool model provides a single point that, once migrated, has all tenants successfully transitioned to your new data model. On the other hand, any problem introduced during a pool migration could impact all of your tenants.

From the outset, you should be thinking about how data migration fits into your overall multitenant SaaS strategy. If you bake this migration orchestration into your delivery pipeline early, you tend to achieve a greater degree of agility in your release process.

Minimizing Invasive Changes

As a rule of thumb, you should have clear policies and tenets to follow as you consider how the data in your systems will evolve. Wherever possible, teams should favor data changes that have backward compatibility with earlier versions. If you can find ways to minimize changes to your application's data representation, you will limit the high overhead of transforming your data into a new representation.

You can leverage commonly used tools and techniques to orchestrate the migration process. In reality, while minimizing invasive changes is often of great importance to SaaS developers, it's not unique to the SaaS domain. As such, it's beyond the scope of what we'll cover in this paper.

Security Considerations

Data security must be a top priority for SaaS providers. When adopting a multitenant strategy, your organization needs a robust security strategy to ensure that tenant data is effectively protected from unauthorized access. Protecting this data and conveying that your system has employed the appropriate security measures is essential to gaining the trust of your SaaS customers.

The storage strategies you choose are likely to use common security patterns supported on AWS. Encrypting data at rest, for example is a horizontal strategy that can be applied universally across any of the models. This provides a foundational level of security which ensures that—even if there is unauthorized access to data—it would be useless without the keys needed to decrypt the information.

Now, as you look at the security profiles of the silo, bridge, and pool models, you will notice additional variations in how security is realized with each one. You'll discover that AWS Identity and Access Management (Amazon IAM), for example, has nuances in how it can scope and control access to tenant data. In general, the silo and bridge models have a more natural fit with IAM policies because they can be applied to limit access to entire databases or tables. Once you cross over to a pool model, you may not be in a position to leverage IAM to scope access to the data. Instead, more responsibility shifts to the authorization

models of your application's services. These services must use a user's identity to resolve the scope and control they have over data in a shared representation.

Isolation and Security

Supporting tenant isolation is fundamental for some organizations and domains. The notion that data is separated—even in a virtualized environment—can be seen as essential to SaaS providers that have specific regulatory or security requirements.

As you consider each AWS storage solution, think about how isolation is achieved on each of the AWS storage services. As you will see, achieving isolation on RDS looks very different from how it does on DynamoDB. Consider these differences as you select your storage strategy and assess the security considerations of your customers.

Management and Monitoring

The approach you adopt for multitenant storage can have a significant impact on the management and monitoring profile of your SaaS solution. In fact, the complexity and approach you take to aggregate and analyze system health can vary significantly for each storage model and AWS technology.

Aggregating Storage Trends

To build an effective operational view of SaaS storage, you need metrics and dashboards that provide you with an aggregated view of tenant activity. You have to be able to proactively identify storage trends that could be influencing the experience spanning all of your tenants. The mechanisms you need to create this aggregated view look very different in the silo and pool models. With siloed storage, you must put tooling in place to collect the data from each isolated database and surface that information in an aggregated model. In contrast, the pool model, by its nature, already has an aggregated view of tenant activity.

Tenant-centric Views of Activity

Your management and monitoring storage solution should provide a way to create tenant-centric views of your storage activity. If a particular tenant is experiencing a storage issue, you'll want to be able to drill into the storage

metrics and profile data to identify what could be impacting that individual tenant. Here, the silo model aligns more naturally with constructing a tenant-centric view of storage activity. A pooled storage strategy will require some tenant filtering mechanism to extract storage activity for a given tenant.

Policies and Alarms

Each AWS storage service has its own mechanisms for evaluating and tuning your application's storage performance. Because storage can often represent a key bottleneck of your system, you should introduce monitoring policies and alarms that will allow you to surface and respond to changes in the health of your application's storage.

The partitioning model you choose will also impact the complexity and manageability of your storage monitoring strategy. The more siloed your solution, the more moving parts to manage and maintain on a tenant-by-tenant basis. In contrast, the shared nature of a pooled storage strategy makes it simpler to have a more centralized, cross-tenant collection of policies and alarms.

The overall goal with these storage policies is to put in place a set of proactive rules that can help you anticipate and react to health events. As you select a multitenant storage model, consider how each approach might influence how you implement your system's storage policies and alarms.

Tiered Storage Models

AWS provides developers with a wide range of storage services, each of which can be applied in combinations to address the varying cost and performance requirements of SaaS tenants. The key here is not to artificially constrain your storage strategy to any one AWS service or storage technology.

As you profile your application's storage needs, take a more granular approach to matching the strengths of a given storage service with the specific requirements of the various components of your application. DynamoDB, for example, might be a great fit for one application service, while RDS might be a better fit for another. If you use a microservice architecture for your solution, where each service has its own view of storage, think about which storage technology best fits each service's profile. It's not uncommon to find a spectrum

of different storage solutions in use across the set of microservices that make up your application.

This strategy also creates an opportunity to use storage as another way of tiering your SaaS solution. Each tier could essentially leverage a separate storage strategy, offering varying levels of performance and SLAs that would distinguish the value proposition of your solution's tiers. By using this approach, you can better align the tenant tiers with the cost and load they are imposing on your infrastructure.

The Developer Experience

As a general architectural principle, developers typically attempt to introduce layers or frameworks that centralize and abstract away horizontal aspects of their applications. The goal here is to centralize and standardize policies and tenant resolution strategies. You might, for example, introduce a data access layer that would inject tenant context into data access requests. This would simplify development and limit a developer's awareness of how tenant identity flows through the system.

Having this layer in place also provides you with more options for policies and strategies that might vary on a tenant-by-tenant basis. It also creates a natural opportunity to centralize configuration and tracking of storage activity.

Linked Account Silo Model

Before digging into specifics of each storage service, let's look at how you can use AWS Linked Accounts to implement the silo model on top of any of the AWS storage solutions. To achieve a silo with this approach, your solution needs to provision a separate Linked Account for every tenant. This can truly achieve a silo because the entire infrastructure for a tenant is completely isolated from other tenants.

The Linked Account approach relies on the Consolidated Billing feature that allows customers to associate child accounts with an overall payer account. The idea here is that—even with separate linked accounts for each tenant—the billing for these tenants is still aggregated and presented as part of a single bill to the payer account.

Figure 4 shows a conceptual view of how Linked Accounts are used to implement the silo model. Here you have two tenants with separate accounts, each of which is associated with a payer account. With this flavor of isolation, you have the freedom to leverage any of the available AWS storage technologies to house your tenant's data.

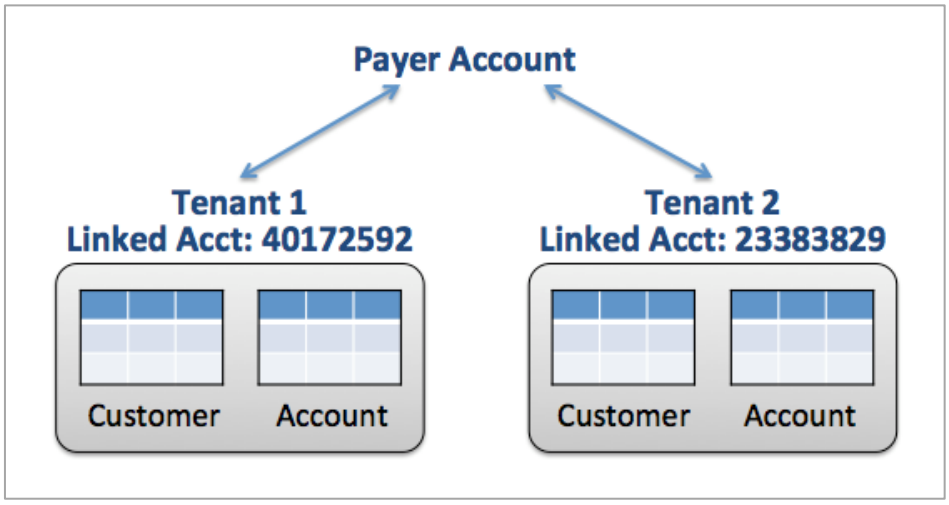

Figure 4: Silo model with linked accounts

At first blush, this can seem like a very appealing strategy for those SaaS providers that require a silo environment. It certainly can simplify some aspects of management and migration of individual tenants. Assembling a view of your tenant costs would also be more straightforward because you can summarize the AWS expenses at the Linked Account level.

Even with these advantages, the Linked Account silo model has important limitations. Provisioning, for example, is certainly more complex. In addition to creating the tenant's infrastructure, you need to automate the creation of each Linked Account and adjust any limits that need it. The larger challenge, however, is scale. AWS has constraints on the number of Linked Accounts you can create, and these limits aren't likely to align with environments that will be creating a large number of new SaaS tenants.

Multitenancy on DynamoDB

The nature of how data is scoped and managed by DynamoDB adds some new twists to how you approach multitenancy. Although some storage services align nicely with the traditional data partitioning strategies, DynamoDB has a slightly

less direct mapping to the silo, bridge, and pool models. With DynamoDB, you have to consider some additional factors when selecting your multitenant strategy.

The sections that follow explore the AWS mechanisms that are commonly used to realize each of the multitenant partitioning schemes on DynamoDB.

Silo Model

Before looking at how you might implement the silo model on DynamoDB, you must first consider how the service scopes and controls access to data. Unlike RDS, DynamoDB has no notion of a database instance. Instead, all tables created in DynamoDB are global to an account within a region. That means every table name in that region must be unique for a given account.

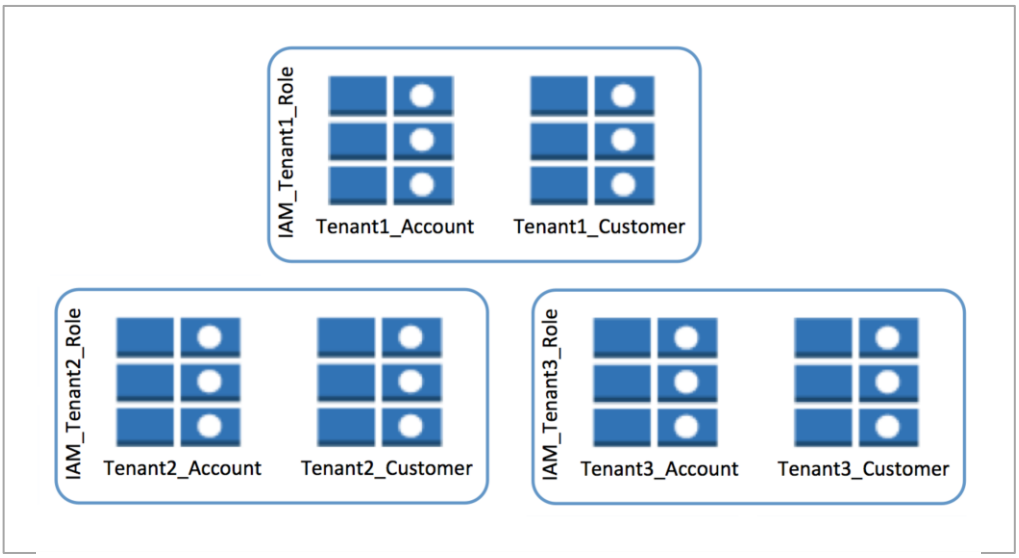

Figure 5: Silo model with DynamoDB tables

If you implement a silo model on DynamoDB, you have to find some way to create a grouping of one or more tables that are associated with a specific tenant. The approach must also create a secure, controlled view of these tables to satisfy the security requirements of silo customers, preventing any possibility of cross-tenant data access.

Figure 5 shows one example of how you might achieve this tenant-scoped grouping of tables. Notice that two tables are created for each tenant (`Account` and `Customer`). These tables also have a tenant identifier that is prepended to

the table names. This addresses DynamoDB's table naming requirements and creates the necessary binding between the tables and their associated tenants.

Access to these tables is also achieved through the introduction of IAM policies. Your provisioning process needs to automate the creation of a policy for each tenant and apply that policy to the tables owned by a given tenant.

This approach achieves the fundamental isolation goals of the silo model, defining clear boundaries between each tenant's data. It also allows for tuning and optimization on a tenant-by-tenant basis. You can tune two specific areas:

- Amazon CloudWatch metrics can be captured at the table level, simplifying the aggregation of tenant metrics for storage activity.
- Table write and read capacity, measured as input and output per second (IOPS), are applied at the table level, allowing you to create distinct scaling policies for each tenant.

The disadvantages of this model tend to be more on the operational and management side. Clearly, with this approach, your operational views of a tenant require some awareness of the tenant table naming scheme to filter and present information in a tenant-centric context. The approach also adds a layer of indirection for any code that needs to interact with these tables. Each interaction with a DynamoDB table requires you to insert the tenant context to map each request to the appropriate tenant table.

SaaS providers that adopt a microservice-based architecture also have another layer of considerations. With microservices, teams typically distribute storage responsibilities to individual services. Each service is given the freedom to determine how it stores and manages data. This can complicate your isolation story on DynamoDB, requiring you to expand your population of tables to accommodate the needs of each service. It also adds another dimension of scoping, where each table for each service identifies its binding to a service.

To offset some of these challenges and better align with DynamoDB best practices, consider having a single table for *all* of your tenant data. This approach offers several efficiencies and simplifies the provisioning, management, and migration profile of your solution.

In most cases, using separate DynamoDB tables and IAM policies to isolate your tenant data addresses the needs of your silo model. Your only other option is to consider the Linked Account silo model, described earlier. However, as outlined previously, the Linked Account isolation model comes with additional limitations and considerations.

Bridge Model

For DynamoDB, the line between the bridge model and silo model is very blurry. Essentially, if your goal using the bridge model is to have a single account with one-off schema variation for each client, you can see how that can be achieved with the silo model described earlier.

For bridge, the only question would be whether you might relax some of the isolation requirements described with the silo model. You can achieve this by eliminating the introduction of any table-level IAM policies. Assuming your tenants aren't requiring full isolation, you could argue that removing the IAM policies could simplify your provisioning scheme. However, even in bridge, there are merits to the isolation. So, although dropping the IAM isolation might be appealing, it's still a good SaaS practice to leverage constructs and policies that can constrain cross-tenant access.

Pool Model

Implementing the pool model on DynamoDB requires you to step back and consider how the service manages data. As data is stored in DynamoDB, the service must continually assess and partition the data to achieve scale. And, if the profile of your data is evenly distributed, you could simply rely on this underlying partitioning scheme to optimize the performance and cost profile of your SaaS tenants.

The challenge here is that data in a multitenant SaaS environment doesn't typically have a uniform distribution. SaaS tenants come in all shapes and sizes and, as such, their data is anything but uniform. It's very common for SaaS vendors to end up with a handful of tenants that consume the largest portion of their data footprint.

Knowing this, you can see how it creates problems for implementing the pool model on top of DynamoDB. If you simply map tenant identifiers to a

DynamoDB partition key, you'll quickly discover that you also create partition "hot spots". Imagine having one very large tenant who would undermine how DynamoDB effectively partitions your data. These hot spots can impact the cost and performance of your solution. With the suboptimal distribution of your keys, you need to increase IOPS to offset the impact of your hot partitions. This need for higher IOPS translates directly into higher costs for your solution.

To solve this problem, you have to introduce some mechanism to better control the distribution of your tenant data. You'll need an approach that doesn't rely on a single tenant identifier to partition your data. These factors all lead down a single path—you must create a secondary sharding model to associate each tenant with multiple partition keys.

Let's look at one example of how you might bring such a solution to life. First, you need a separate table, which we'll call the "tenant lookup table", to capture and manage the mapping of tenants to their corresponding DynamoDB partition keys. Figure 6 represents an example of how you might structure your tenant lookup table.

Partition Key	Attributes	

TenantID
Tenant1
	CustomerTable
	{
	ShardCount: 4,
	ShardSize: [4, 9, 4, 5],
	ShardIds: ["93", "932", "21", "736"]
	}
		AccountTable
		{
		ShardCount: 4,
		ShardSize: [3, 4, 4, 5],
		ShardIds: ["43", "19", "971", "85"]
		}

TenantID
Tenant2
	CustomerTable
	{
	ShardCount: 2,
	ShardSize: [3, 2],
	ShardIds: ["221", "538"]
	}
		AccountTable
		{
		ShardCount: 3,
		ShardSize: [5, 3, 5],
		ShardIds: ["61", "216", "492"]
		}

Figure 6: Introducing a tenant lookup table

This table includes mappings for two tenants. The items associated with these tenants have attributes that contain sharding information for each table that is associated with a tenant. Here, our tenants both have sharding information for their `Customer` and `Account` tables. Also notice that for each tenant-table combination there are three pieces of information that represent the current sharding profile for a table. These are:

- **ShardCount.** An indication of how many shards are currently associated with the table.
- **ShardSize.** The current size of each of the shards
- **ShardIds.** A list of partition keys mapped to a tenant (for a table).

With this mechanism in place, you can control how data is distributed for each table. The indirection of the lookup table gives you a way to dynamically adjust a tenant's sharding scheme based on the amount of data it is storing. Tenants with a particularly large data footprint will be given more shards. Because the model configures sharding on a table-by-table basis, you have much more granular control over mapping a tenant's data needs to a specific sharding configuration. This allows you to better align your partitioning with the natural variations that often show up in your tenant's data profile.

Although introducing a tenant lookup table provides you with a way to address tenant data distribution, it does not come without a cost. This model now introduces a level of indirection that you have to address in your solution's data access layer. Instead of using a tenant identifier to directly access your data, first consult the shard mappings for that tenant and use the union of those identifiers to access your tenant data. The sample `Customer` table in Figure 7 shows how data would be represented in this model.

Partition Key	Attributes	
ShardID 93	**CustomerID** 4923000093	**Name** Bob Jones
ShardID 221	**CustomerID** 9830193911	**Name** Jane Thomas
ShardID 21	**CustomerID** 3492098u72	**Name** Sally Smith
ShardID 932	**CustomerID** 1158304894	**Name** Randy Hanson
ShardID 93	**CustomerID** 8194922299	**Name** Wendy Wilkerson
ShardID 538	**CustomerID** 4800021941	**Name** Henry Hanks
ShardID 932	**CustomerID** 7918499931	**Name** Mary Young
ShardID 736	**CustomerID** 5939202749	**Name** Lisa Franks

Figure 7: Customer table with shard IDs

In this example, the `ShardID` is a direct mapping from the table shown in Figure 6. That tenant lookup table included two separate lists of shard identifiers for the `Customer` table, one for `Tenant1` and one for `Tenant2`. These shard identifiers correlate directly to the values you see in this sample customer table. Notice that the actual tenant identifier never appears in this `Customer` table.

Managing Shard Distribution

The mechanics of this model aren't particularly complex. The problem gets more interesting when you think about how to implement a strategy that effectively distributes your data. How do you detect when a tenant requires additional shards? Which metrics and criteria can you collect to automate this process? How do the characteristics of your data and domain influence your data profile? There is no single approach that universally resolves these questions for every solution. Some SaaS organizations manually tune this, based on their customer insights. Others have more natural criteria that guide their approach.

The approach outlined here is one way you might choose to handle the distribution of your data. Ultimately, you'll likely find a hybrid of the principles we describe that best aligns with the needs of your environment. The key takeaway is that if you adopt the pool model, be aware of how DynamoDB partitions data. Moving in data blindly without considering how the data will be distributed will likely undermine the performance and cost profile of your SaaS solution.

Dynamically Optimizing IOPS

The IOPS needs of a SaaS environment can be challenging to manage. The load tenants place on your system can vary significantly. Setting the IOPS to some worst case, maximum level undermines the desire to optimize costs based on actual load.

Instead, consider implementing a dynamic model where the IOPS of your tables are adjusted in real time based on the load profile of your application. Dynamic

DynamoDB is one configurable open source solution you can use to address this problem.[1]

Supporting Multiple Environments

As you think about the strategies outlined for DynamoDB, consider how each of these models will be realized in the presence of multiple environments (QA, development, production, etc.). The need for multiple environments impacts how you further partition your experience to separate out each of your storage strategies on AWS. With the bridge and pool models, for example, you can end up adding a qualifier to your table names to provide environment context. This adds a bit of misdirection that you must factor into your provisioning and runtime resolution of table names.

Migration Efficiencies

The schema-less nature of DynamoDB offers real advantages for SaaS providers, allowing you to apply updates to your application and migrate tenant data without introducing new tables or replication. DynamoDB simplifies the process of migrating tenants between your SaaS versions and allows you to simultaneously host agile tenants on the latest version of your SaaS solution, while allowing other tenants to continue using an earlier version.

Weighing the Tradeoffs

Each of the models has tradeoffs to consider as you determine which model best aligns with your business needs. The silo pattern may seem appealing, but the provisioning and management add a dimension of complexity that undermines the agility of your solution. Supporting separate environments and creating unique groups of tables will undoubtedly impact the complexity of your automated deployment. The bridge represents a slight variation of the silo model on DynamoDB. As such, it mirrors most of what we find with the silo model.

The pool model on DynamoDB offers some significant advantages. The consolidated footprint of the data simplifies the provisioning, migration, and management and monitoring experiences. It also allows you to take a more multitenant approach to optimizing consumption and tenant experience by tuning the read and write IOPS on a cross-tenant basis. This allows you to react

more broadly to performance issues and introduces opportunities to minimize cost. These factors tend to make the pool model very appealing to SaaS organizations.

Multitenancy on RDS

With so many early SaaS systems delivered on relational databases, the developer community has established some common patterns for address multitenancy in these environments. In fact, RDS has a more natural mapping to the silo, bridge, and pool models.

The construct and representation of data in RDS is very much an extension of nonmanaged relational environments. The basic mechanisms that are available in MySQL, for example, are also available to you in RDS. This makes the realization of multitenancy on all of the RDS flavors relatively straightforward.

The following sections outline the various strategies that are commonly employed to realize the partitioning models on RDS.

Silo Model

You can achieve the silo pattern on AWS in multiple ways. However, the most common and simplest approach for achieving isolation is to create separate database instances for each tenant. Through instances, you can achieve a level of separation that typically satisfies the compliance needs of customers without the overhead of provisioning entirely separate accounts.

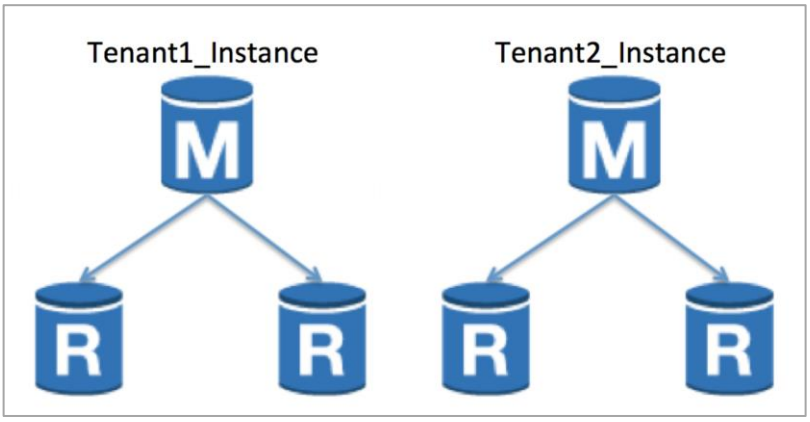

Figure 8: RDS instances as silos

Figure 8 shows a basic silo model as it could be realized on top of RDS. Here, two separate instances are provisioned for each tenant.

The diagram depicts a master database and two read replicas for each tenant instance. This is an optional concept to highlight how you can use this approach to set up and configure an optimized, highly available strategy for each tenant.

Bridge Model

Achieving the bridge model on RDS fits the same themes we see across all the storage models. The basic approach is to leverage a single instance for all tenants while creating separate representations for each tenant within that database. This introduces the need to have provisioning and runtime table resolution to map each table to a given tenant.

The bridge model offers you the opportunity to have tenants with different schemas and some flexibility when migrating tenant data. You could, for example, have different tenants running different versions of the product at a given moment in time and gradually migrate schema changes on a tenant-by-tenant basis.

Figure 9 provides an example of one way you can implement the bridge model on RDS. In this diagram, you have a single RDS database instance that contains separate customer tables for `Tenant1` and `Tenant2`.

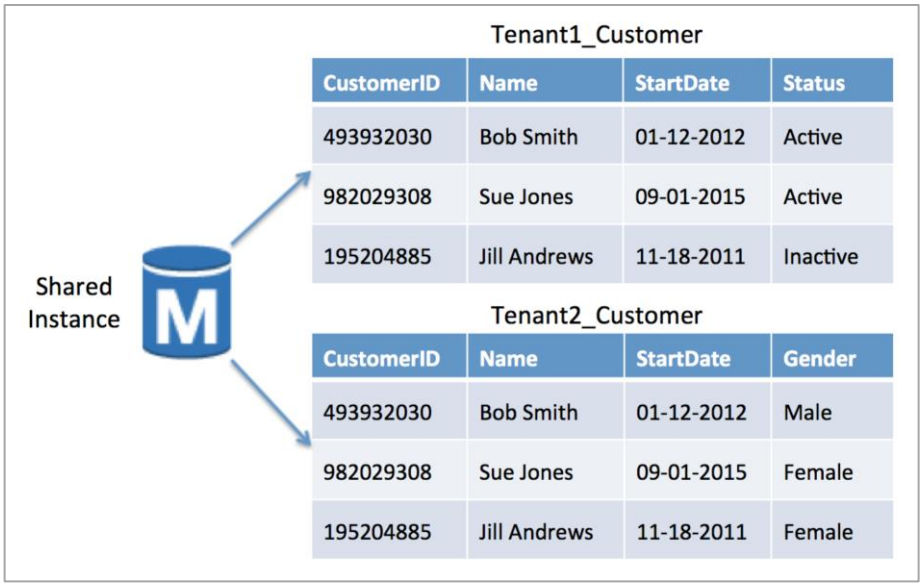

Figure 9: Example of a bridge model on RDS

This example highlights the ability to have schema variation at the tenant level. `Tenant1`'s schema has a `Status` column, while that column is removed and replaced by the `Gender` column used by `Tenant2`.

Another option here would be to introduce the notion of separate databases for each tenant within an instance. The terminology varies for each flavor of RDS. Some RDS storage containers refer to this as a database; others label it as a schema.

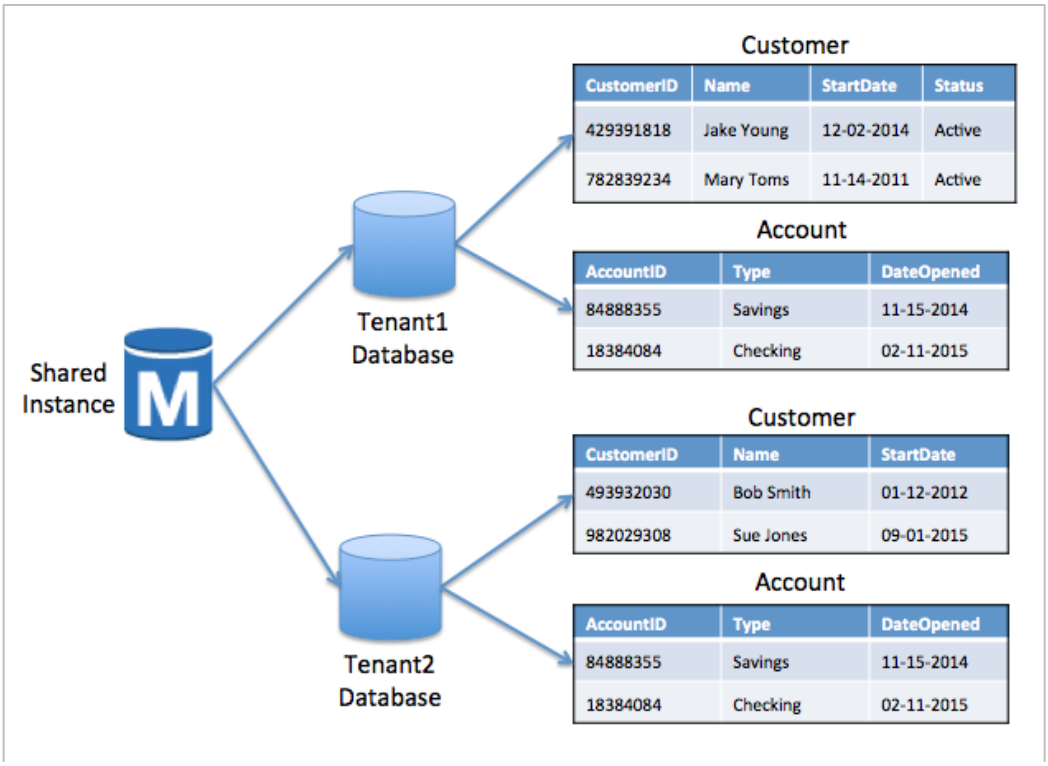

Figure 10: RDS bridge with separate tables/schemas

Figure 10 provides an illustration of this alternate bridge model. Notice that we created databases for each of the tenants, and the tenants then have their own collection of tables. For some SaaS organizations, this scopes the management of their tenant data more naturally, avoiding the need to propagate the naming to individual tables.

This model is appealing, but it may not be the best fit for all flavors of RDS. Some RDS containers limit the number of databases/schemas that you can create for an instance. The SQL Server container, for example, allows only 30 databases per instance, which is likely unacceptable for most SaaS environments.

Although the bridge model allows for variation from tenant to tenant, it's important to know that, typically, you should still adopt policies that try to limit schema changes. Each time you introduce a schema change, you can take on the challenge of successfully migrating your SaaS tenants to the new model without absorbing any downtime. So, although this model simplifies those migrations, it doesn't promote one-off tenant schemas or regular changes to the representation of your tenant's data.

Pool Model

The pool model for RDS relies on traditional relational indexing schemes to partition tenant data. As part of moving all the tenant data into a shared infrastructure model, you store the tenant data in a single RDS instance and the tenants share common tables. These tables are indexed with a unique tenant identifier that is used to access and manage each tenant's data.

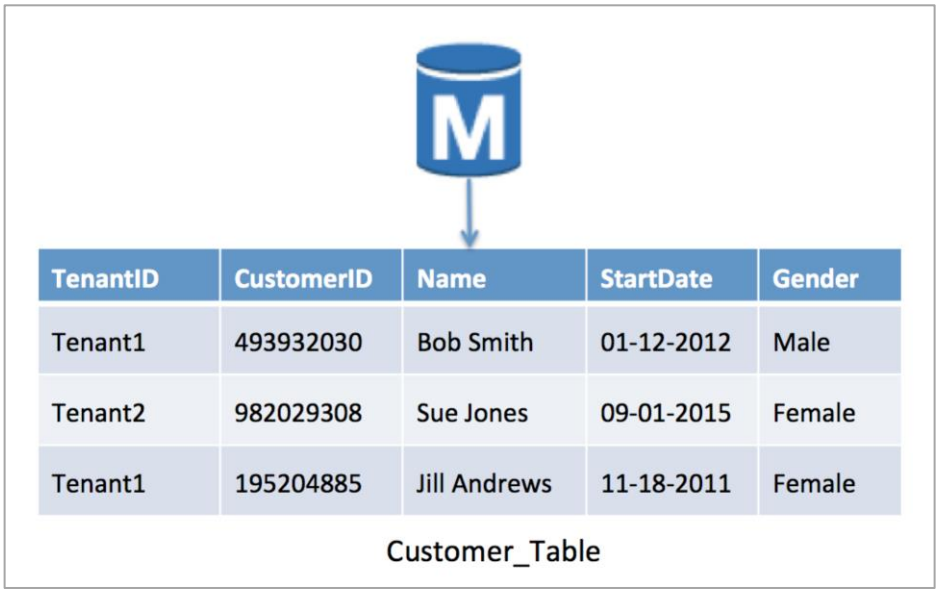

Figure 11 - RDS pool model with shared schema

Figure 11 provides an example of the pool model in action. Here a single RDS instance with *one* `Customer` table holds data for all of the application's tenants. RDS is an RDBMS, so all tenants must use the same schema version. RDS is not like DynamoDB, which has a flexible schema that allows each tenant to have a unique schema within a single table.

Factoring in Single Instance Limits

Many of the models we described concentrate heavily on storing data in a single instance and partitioning data within that instance. Depending on the size and performance needs of your SaaS environment, using a single instance might not fit the profile of your tenant data. RDS has limits on the amount of data that can be stored in a single instance. The following is a breakdown of the limits:

- MySQL, MariaDB, Oracle, PostgreSQL – **6 TB**
- SQL Server – **4 TB**
- Aurora – **64 TB**

In addition, a single instance introduces resource contention issues (CPU, memory, I/O).

In scenarios where a single instance is impractical, the natural extension is to introduce a sharding scheme where your tenant data is distributed across multiple instances. With this approach, you start with a small collection of sharded instances. Then, continually observe the profile of your tenant data and expand the number of instances to ensure that no single instance reaches limits or becomes a bottleneck.

Weighing the Tradeoffs

The tradeoffs of using RDS are fairly straightforward. The primary theme is often more about trading management and provisioning complexity for agility. Overall, the pain points of provisioning automation are likely lower with the silo model on RDS. However, the cost and management efficiency associated with the pool model is often compelling. This is especially significant as you think about how these models will align with your continuous delivery environment.

Multitenancy on Amazon Redshift

Amazon Redshift introduces additional twists to factor into your multitenant thinking. Amazon Redshift focuses on building high-performance clusters to house large-scale data warehouses. Amazon Redshift also places some limits on the constructs that you can create within each cluster. Consider the following limits:

- 60 databases per cluster
- 256 schemas per database
- 500 concurrent connections per database
- 50 concurrent queries
- Access to a cluster enables access to all databases in the cluster

You can imagine how these limits influence the scale and performance that is delivered to Amazon Redshift. You can also see how these limits can impact your approach to multitenancy with Amazon Redshift. If you are targeting a modest tenant count, these limits might have little influence on your solution. However, if you're targeting a large number of tenants, you'd need to factor these limits into your overall strategy.

The following sections highlight the strategies that are commonly used to realize each multitenant storage model on Amazon Redshift.

Silo Model

Achieving a true, silo model isolation of tenants on Amazon Redshift requires you to provision separate clusters for each tenant. Via clusters, you can create the well-defined boundary between tenants that is commonly required to assure customers that their data is successfully isolated from cross-tenant access. This approach best leverages the natural security mechanisms in Amazon RedShift, so you can control and restrict tenant access to a cluster using a combination of IAM policies and database privileges. IAM controls overall cluster management, and the database privileges are used to control access to data within the cluster.

The silo model gives you the opportunity to create a tuned experience for each tenant. With Amazon Redshift, you can configure the number and type of nodes in your cluster, so that you can create environments that target the load profile

of each individual tenant. You can also use this as a strategy for optimizing costs.

The challenge of this model, as we've seen with other silo models, is that each tenant's cluster must be provisioned as part of the onboarding process. Automating this process and absorbing the extra time and overhead associated with the provisioning process adds a layer of complexity to your deployment footprint. It also has some impact on the speed with which a new tenant can be allocated.

Bridge Model

The bridge model does not have a natural mapping on Amazon Redshift. Technically, you could create separate schemas for each tenant. However, you would likely run into issues with the Amazon Redshift limit of 256 schemas. In environments with any significant number of tenants, this simply doesn't scale. Security is also a challenge for Amazon Redshift in the bridge model. When you are authorized as a user of an Amazon Redshift cluster, you are granted access to all the databases within that cluster. This pushes the responsibility for enforcing finer-grained access controls to your SaaS application.

Given the motives for the bridge model and these technical considerations, it seems impractical for most SaaS providers to consider using this approach on Amazon Redshift. Even if the limits are manageable for your solution, the isolation profile is likely unacceptable to your customers. Ultimately, the best answer is to simply use the silo model for any tenant that requires isolation.

Pool Model

Building the pool model on Amazon Redshift looks very much like the other storage models we've discussed. The basic idea is to store data for all tenants in a single Amazon Redshift cluster with shared databases and tables. In this approach, the data for tenants is partitioned via the introduction of a column that represents a unique tenant identifier.

This approach gives most of the goodness that we saw with the other pool models. Certainly the overall management, monitoring, and agility are improved by housing all of the tenant data in a single Amazon Redshift cluster.

The limit on concurrent connections is the area that adds a degree of difficulty to implementing the pool model on Amazon Redshift. With an upper limit of 500 concurrent connections, many multitenant SaaS environments can quickly exceed this limit. This doesn't eliminate the pool model from contention. Instead, it pushes more responsibility to the SaaS developer to put an effective strategy in place to manage how and when these connections are consumed and released.

There are some common ways to address connection management. Developers often leverage client-based caching to limit their need for actual connections to Amazon Redshift. Connection pooling can also be applied in this model. Developers need to select a strategy that ensures that the data access patterns of their application can be met effectively without exceeding the Amazon Redshift connection limit.

Adopting the pool model also means keeping your eye on the typical issues that come up any time you're operating in a shared infrastructure. The security of your data, for example, requires some application-level policies to limit cross-tenant access. Also, you likely need to continually tune and refine the performance of your environment to prevent any one tenant from degrading the experience of others.

Keeping an Eye on Agility

The matrix of multitenant storage options can be daunting. It can be challenging to identify the solution that represents the best mix of flexibility, isolation, and manageability. Although it's important to consider all the options, it's also essential to continually factor agility into your multitenant storage thinking. The success of SaaS organizations is often heavily influenced by the amount of agility that is baked into their solution.

The storage technology and isolation model you select directly impacts your ability to easily deploy new features and functionality. The shape of your structure and content of your data often change to support new features, and this means your underlying storage model must accommodate these changes without requiring downtime. Each isolation model has pros and cons when it comes to supporting this seamless migration. As you consider your options, give these factors the appropriate weight.

While the silo, bridge, and pool models all have an agility footprint, you can apply common tenets to help you remain as nimble as possible. A key tenet is the rather obvious but occasionally violated need to minimize one-off variations for tenant data. The silo and bridge models, for example, can lead to storage variations that can complicate your ability to push out new features to all of your SaaS customers as part of a single automated event. Teams often use automation and continuous deployment to limit the amount of friction introduced by their multitenant storage strategy.

As you settle into a storage strategy, expect and embrace the reality that your storage requirements continually evolve. The needs of SaaS customers are a moving target, and the storage model you pick today might not be a good fit tomorrow. AWS also continues to introduce new features and services that can represent new opportunities to enhance your approach to storage.

Conclusion

The storage needs of SaaS customers aren't simple. The reality of SaaS is that your business's domain, customers, and legacy considerations affect how you determine which combination of multitenant storage options best meet the needs of your business.

Although there is no single strategy that universally fits every environment, it is clear that some models do align better with the core tenets of the SaaS delivery model. In general, the pool-based approaches to storage—on any AWS storage technology—align well with the need for a unified approach to managing and operating a multitenant environment. Having all your tenants in one shared repository and representation streamlines and unifies your approach's operational and deployment footprint, enabling cross-tenant views of health and performance.

The silo and bridge models certainly have their place and, for some SaaS providers, are absolutely required. The key here is that, if you head down this path, agility can get more complicated. Some AWS storage technologies are better positioned to support isolated tenant storage schemes. Building a silo model on RDS, for example, is less complex than it is on DynamoDB. Generally, whenever you rely on linked accounts as your partitioning model, you will tackle more provisioning, management, and scaling challenges.

Beyond the mechanics of achieving multitenancy, think about how the profile of each AWS storage technology can fit with the varying needs of your multitenant application's functionality. Consider how tenants will access the data and how the shape of that data will need to evolve to meet the needs of your tenants. The more you can decompose your application into autonomous services, the better positioned you are to pick and choose separate storage strategies for each service.

After exploring these services and portioning schemes, you should have a much better sense of the patterns and inflection points that will guide your selection of a multitenant storage strategy. AWS equips SaaS providers with a rich palette of services and constructs that can be combined to address any number of multitenant storage needs.

Contributors

The following individuals and organizations contributed to this document:

- Tod Golding, Partner Solutions Architect, AWS Partner Program
- Clinton Ford, Senior Product Marketing Manager, DynamoDB
- Zach Christopherson, Database Engineer, Amazon Redshift
- Brian Welker, Principal Product Owner, RDS MySQL and MariaDB

Further Reading

For additional help, see the following sources:

- Amazon RDS
- Amazon DynamoDB
- Amazon Redshift

Notes

[1] https://dynamic-dynamodb.readthedocs.io/en/latest/index.html