OctoDB - Scalable Multi-Tenant DBaaS with Automated Migrations

1st Naveen Alampally
Department of Computer Science and Engineering
New York University, Brooklyn, USA
an4207@nyu.edu

2nd Anthony Nikhil Reddy Lingala
Department of Computer Science and Engineering
New York University, Brooklyn, USA
al8291@nyu.edu

3rd Sai Akhil Tekuri
Department of Computer Science and Engineering
New York University, Brooklyn, USA
st5050@nyu.edu

4th Adrian Dsouza
Department of Computer Science and Engineering
New York University, Brooklyn, USA
ad7628@nyu.edu

5th Yiheng Chen
Department of Computer Science and Engineering
New York University, Brooklyn, USA
yc7766@nyu.edu

6th Utkarsh Mittal
Department of Computer Science and Engineering
New York University, Brooklyn, USA
um2100@nyu.edu

7th Siyu Liao
Department of Computer Science and Engineering
New York University, Brooklyn, USA
sl11885@nyu.edu

8th Sumedh Sandeep Parvatikar
Department of Computer Science and Engineering
New York University, Brooklyn, USA
sp7479@nyu.edu

***Abstract*—Multi-tenant database systems are critical for modern cloud-based applications, enabling multiple customers (tenants) to share a single database instance while maintaining isolation, security, and scalability. This paper presents a framework for implementing a multi-tenant database system that ensures data segregation, efficient resource utilization, and robust scalability. The system leverages a hybrid approach combining shared and isolated resources to balance performance and cost. Key features include tenant-specific schema customization, dynamic privilege management, automated provisioning via serverless computing, and secure access. Experimental results demonstrate the system's ability to handle dynamic tenant onboarding, support schema updates, and manage concurrent operations with minimal performance overhead. This approach exemplifies the potential of serverless and schema management tools in transforming DBaaS solutions for modern applications.**

***Index Terms*—Cloud-based applications, Scalability, Schema customization, Serverless computing, Automated provisioning, Tenant onboarding, Schema updates, Concurrent operations**

I. Problem Statement

The increasing demand for cloud-based applications requires scalable and cost-efficient multi-tenant database systems capable of supporting multiple tenants while ensuring data isolation, security, and efficient resource utilization. Key challenges include maintaining strict data separation, achieving scalability under dynamic tenant onboarding and high-concurrency workloads, balancing resource optimization between shared and isolated models, and automating provisioning, schema management, and privilege control. Existing solutions often compromise scalability or maintainability. This paper addresses these challenges by proposing a hybrid framework leveraging serverless computing and schema management tools for a secure, scalable, and maintainable multi-tenant database system.

Main challenges:

1) **Dynamic Provisioning:** Bringing new tenants on board usually means a lot of manual work for database provisioning and configuration. This process is not only time-consuming but also prone to errors, particularly as the number of tenants increases.
2) **Schema Migrations:** Keeping database schemas up-to-date across multiple tenant databases is a major challenge in multi-tenant architectures. Doing this manually or with inadequate automation can lead to downtime, inconsistencies, and general chaos.

II. Motivation

With the rise of cloud-based applications, the demand for efficient multi-tenant database systems has become more pressing than ever. Companies need solutions that can securely separate tenant data while maximizing resource efficiency and keeping costs low. Traditional systems often struggle to balance scalability, performance, and ease of maintenance, making them less suitable for fast-paced, dynamic environments. However, advancements in serverless computing and schema management tools provide an opportunity to redesign

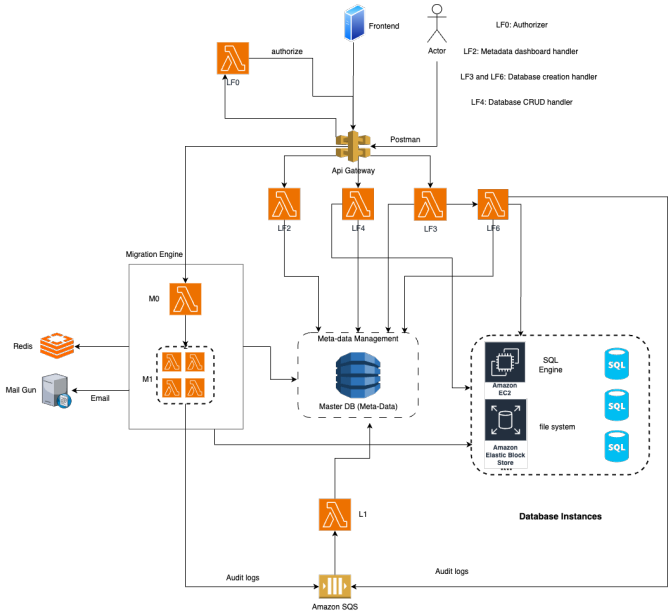

Fig. 1. Architecture for OctoDB

multi-tenant database architectures. These technologies enable features like dynamic tenant onboarding, effortless schema updates, and streamlined automation. This paper is driven by the need to bridge the gaps in existing solutions and offer a scalable, secure, and maintainable framework tailored for modern cloud-based applications.

III. Existing Solutions

Several solutions exist for managing multi-tenant databases using technologies like MySQL, PostgreSQL, and SQLite. While these options handle basic multi-tenancy, they often lack automation with migration, synchronizing with child schema, or scalability features critical for modern applications.

1) **Amazon RDS:** A scalable, managed service for MySQL and PostgreSQL [1]. However, it lacks native tools for tenant-specific schema migrations and offline synchronization.
2) **Google Cloud SQL:** Similar to Amazon RDS, it supports multi-tenancy and offers high availability [2], but lacks offline synchronization and automated schema management.
3) **Supabase:** An open-source backend-as-a-service platform using PostgreSQL [4]. While it supports multi-tenancy, its schema management and offline synchronization are not robust for large-scale systems.
4) **Firebase:** A NoSQL database focused on real-time synchronization [5], but unsuitable for multi-tenant relational databases or SQL schema migrations.

IV. System Architecture

OctoDB has multiple components we will discuss in this section:

A. User Creation

The workflow for creating a user is as follows:

1) The API gateway receives the request containing the tenant name, email and password. It routes the request to LF3.
2) LF3 processes the request by hashing the password, generating a unique *tenant_id*, which is a unique identifier for a tenant, and a *db_id*, which is the primary key to the database corresponding to the tenant. If the request is missing either name, email or password, the lambda returns an error
3) LF3 invokes LF6 asynchronously, and creates the MySQL database corresponding to the tenant. MySQL commands are executed on the EC2 instance. The metadata about the database is stored in the MetaDB. The asynchronous architecture is used to hide the delay of database creation at the back-end.

B. Database Creation

Version 1: Initial Implementation

- **Database Initialization and Management:**
  - Connected to a MySQL database on an EC2 instance using `mysql.connector`.
  - Established an admin connection to:
    - Check for the target database's existence.
    - Create the database if absent.
    - Grant privileges to `app_user`.
- **Error Handling:** Used MySQL error codes to handle issues such as access denial or missing databases.
- **Privilege Granting:** Dynamically granted database privileges to `app_user`.

Error Encountered:

- **Permission Issue:** Faced errors when dynamically granting privileges, requiring manual intervention to execute `GRANT` commands.
- **Automation Limitation:** Privilege management conflicted with Lambda's stateless and automated nature.

Version 2: Working Implementation

- **Decoupling Database Operations:**
  - Replaced direct MySQL management with Skeema for schema initialization and setup.
  - Developed reusable commands for database creation, privilege granting, and schema deployment.
- **AWS Integration:**
  - Used AWS Systems Manager (SSM) to run shell commands on the EC2 instance hosting the database.
  - Leveraging Skeema for sending commands via the `send_command` API.
- **Error Logging and Reporting:** Implemented filtered logging and sent logs to an SQS queue for debugging.
- **Dynamic Configuration:**
  - Dynamically generated SQL commands and Skeema scripts.

  - Configured default/custom schema setups for various needs.
- **Schema Management:**
  - For admin-level databases, created tables using predefined scripts.
  - For application databases, initialized schemas from templates or default tables.

Error Resolution:

- **Permission Issue:**
  - Managed privileges through predefined Skeema configurations and automated shell commands via SSM.
  - Eliminated manual intervention and ensured consistent privilege management.
- **Automation Enhancement:**
  - Used AWS SSM for secure, automated command execution on the EC2 instance.
  - Centralized schema management with Skeema for seamless updates.

C. Migration

The migration framework is designed to validate, process, and execute SQL commands across tenant databases while maintaining data integrity and ensuring security.

SQL Command Validation The system employs a function to validate `CREATE TABLE` commands for safety:

- Disallowed keywords such as `DROP`, `INSERT`, and `Comments` are checked to prevent malicious activities.
- The command must end with a single semicolon and adhere to the standard `CREATE TABLE` pattern.
- Nested commands like `SELECT` or `EXECUTE` are explicitly disallowed.

Migration Workflow Upon receiving SQL scripts, the `M0` performs the following:

- Extracts tenant-specific metadata (`tenant_id`, `db_id`, etc.) from the authorizer context.
- Validates each SQL command and assigns a unique `job_id` using `UUID`.
- Generates job records containing migration details, including timestamps and admin flags.

If the user is an administrator, child database schemas are identified and queued for migration, ensuring hierarchical consistency.

Redis Integration Jobs are cached in Redis for tracking retry attempts:

- Redis is hosted on an EC2 instance, providing a centralized cache for migration tasks.
- Each `job_id` is stored with a retry counts for failure recovery.

Asynchronous Execution The system invokes the `M1` Lambda function asynchronously for table-level migrations:

- Jobs are submitted as events using AWS Lambda's `invoke` API.
- Errors during invocation are logged, and the user is notified for support.

Upon successful submission, the framework returns a status message indicating that the migration job is submitted, and the users are directed to the portal for updates.

The migration process follows an asynchronous architecture that ensures scalability and fault tolerance. The sequence of events is structured as follows.

1) **LF0 (Lambda Function):** The entry point to all the endpoints in the system except for `generate_token()` and `create_user()` endpoint. This function validates the authorization token provided by the user and forwards necessary details like tenant_id, db_id and more to the respective designate target endpoint. If the auth token is invalid, it returns back to the user with an error message.
2) **M0 (Lambda Function):** The entry point for all migration requests. It initializes job metadata, parses SQL scripts, and generates migration jobs with unique `job_id`s. It is also responsible for job orchestration and dispatching details to M1 for each job asynchronously. The M0 is also responsible to perform sanity checks to not allow malicious code into tenant's databases.
3) **M1 (Lambda Function):** Executes individual migration jobs. Each `job_id` corresponds to a specific SQL operation and follows the stages: script backup, schema update, validation. The migration is performed using the `skeema.io` framework. The steps performed by M1 is logged into the audit tables to provide an overview to the user. Moreover, after the execution of the migration, the M1 is responsible to notify the user through email by leveraging the Mail gun service. A failure in the migration automatically rollbacks the table to its original state thus acting as a transaction.

Admin Migration

1) **Migrating Parent Schema (Cascading to Child Schemas):** Initiates a cascading migration where changes to the parent schema are adaptively synchronized to all associated child schemas. This ensures hierarchical consistency.
2) **Creating New Tables:** Facilitates the creation of new tables independent of existing schema migration workflows. These operations are tracked separately and involve unique validation steps.

Retry Mechanism Failed migration jobs are handled using a Redis-backed retry mechanism:

- Each job failure is logged with details about the error cause.
- Failed jobs are re-initiated up to two times, with exponential backoff.
- Persistent failures are flagged for manual intervention, and relevant notifications are sent to administrators and the tenants.

User Notifications Users are informed about events through the following channels:

- **Portal:** Real-time status updates are displayed on the user portal.
- **Logs:** Detailed logs are stored for auditing and debugging purposes.
- **Email:** Notifications summarizing the migration outcome (success or failure) are sent via the Mail gun service.

D. API Endpoints and CRUD Operations

The Tenant/Database API serves as a centralized platform for accessing and managing tenant and database metadata, including operations on tables, logs, and statistics. It uses a single POST endpoint (`/dashboard`), with the desired action specified dynamically in the request body. The API employs a Lambda authorizer to validate access tokens and injects tenant-specific details like `tenant_id` and `db_id` into the request context, ensuring secure and isolated operations for each tenant.

The supported actions include retrieving tenant information, listing associated databases, fetching database metadata, listing tables within a database, and querying logs or performance statistics. For example:

- The `get_user_info` action retrieves metadata about the tenant, excluding sensitive details such as hashed passwords.
- The `get_database_tables` action returns a list of table names within a database, providing a quick overview of available data structures.
- Actions like `get_log_table` and `get_query_logs` support paginated log retrieval, leveraging DynamoDB's `last_evaluated_key` for efficient navigation through large datasets.

The request format includes a JSON body with a required `action` field and optional parameters like `table_name` for table-specific actions or `limit` for paginated results. For instance, a request to fetch logs might include:

```
{
  "action": "get_log_table",
  "limit": 10,
  "last_evaluated_key": { "id": "log123" }
}
```

This structure enables the API to process various operations efficiently, guided by the specified action.

Responses are tailored to the requested action, ensuring relevance and clarity. A successful response for `get_user_info` include the tenant ID, email, and non-sensitive metadata, while a response for `get_query_stats` provides statistical insights into query performance. Error handling is robust, with clear messages for issues like invalid actions 400 Bad Request or missing resources.

Security is a cornerstone of the API. The Lambda authorizer validates the `Authorization` header and injects `tenant_id` and `db_id` into the `requestContext.authorizer` object. This dynamic injection ensures that requests are scoped to the authenticated tenant, enforcing strict isolation while maintaining the flexibility required for multi-tenant environments. By centralizing database-related actions in a secure and extensible API, this system supports scalable, efficient, and tenant-aware management of database resources.

The API supports the execution of single SQL statements, including `SELECT`, `INSERT`, `UPDATE`, and `DELETE`. Schema modification operations (e.g., `CREATE`, `DROP`) and multi-statement queries are prohibited for security and consistency. For example:

- A `SELECT` query will return the requested data.
- An `INSERT` query will update the database and return the number of rows affected.
- Prohibited queries, such as `CREATE TABLE`, will trigger a `403 Forbidden` response.

Request and Response Workflow

Users must include a JSON payload in the request body, with the `query` field specifying the SQL command to execute. A valid request example is:

```
{
  "query": "SELECT * FROM users
  WHERE status = 'active'"
}
```

The API processes the query and returns an appropriate response:

- On success, the response includes:

```
{
  "message": "Query executed
  successfully",
  "rowsAffected": 10,
  "data": [
    { "column1": "value1",
    "column2": "value2" }
  ]
}
```

- For invalid queries or prohibited operations, the response includes error details, such as:

```
{
  "error": "Prohibited operation
  detected: CREATE"
}
```

Error Handling

The API handles errors gracefully:

- `400 Bad Request`: Triggered when the query is missing or invalid.
- `403 Forbidden`: Raised for prohibited operations or malformed SQL.
- `404 Not Found`: Indicates that the specified database (`db_id`) does not exist.

E. Generating tokens

User tokens are required to validate the user and provide them access to the corresponding database. The API gateway calls the corresponding lambda function. The request is passed in the following format:

```
{
    "email":
    "password":
    "read":
}
```

The authorizing token is returned to the user. The third variable `read` is set as `False` by default and indicates that the user has not yet accessed their token. If set to `True`, it returns the previously generated token and does not generate a new token

F. Logging

The logging mechanism is designed to ensure efficient processing, storage, and management of logs generated by the migration functions. Both `LF6` and `M1` connect to an Amazon SQS queue, which in turn triggers a Lambda function responsible for processing log entries and storing them in a DynamoDB table. Below is an outline of the architecture and workflow:

1) **Log Ingestion:** The migration functions send logs as JSON messages to the SQS queue. Each message includes details such as `tenant_id`, log content (`stderr` or `stdout`), and other metadata.
2) **Lambda Trigger (L1):** The Lambda function is triggered by events from the SQS queue. It processes the received log messages using the following steps:
   - Parses the message content to extract essential data, including:
     - `tenant_id`: Identifier for the tenant originating the log.
     - `logs`: The log message, extracted from `stderr` or `stdout`.
     - `create_timestamp`: The timestamp embedded in the log message.
   - Generates a unique `log_id` using UUID to ensure each log entry is uniquely identifiable.
   - Validates the log content and prepares it for storage.
3) **DynamoDB Storage:** Processed log entries are stored in the `log_table` DynamoDB table. Each entry includes:
   - `log_id`: A unique identifier for the log.
   - `tenant_id`: The tenant associated with the log.
   - `create_timestamp`: Timestamp indicating when the log was generated.
   - `logs`: The actual log content.
4) **Error Handling and Retry Mechanism:** If a log entry cannot be stored in DynamoDB due to an error, the Lambda function logs the error for debugging. To ensure reliability:
   - Logs are retried up to three times using the SQS retry mechanism.
   - Persistent failures are flagged for manual intervention.
5) **Message Deletion from SQS:** Once a log entry is successfully stored in DynamoDB, the corresponding SQS message is deleted to prevent reprocessing. This ensures idempotence and efficient queue management.
6) **Monitoring and Debugging:** The Lambda function logs its processing status, including successful operations and errors, to facilitate monitoring and debugging.
   CloudWatch Integration: Describe how CloudWatch metrics (e.g., failed SQS messages) help in identifying issues.
   Slack/Email Alerts: Mention triggering Slack or email notifications via CloudWatch alarms for critical errors.

This mechanism ensures that all logs are reliably ingested, processed, and stored while maintaining high fault tolerance and scalability.

G. Scalability

We use asynchronous architecture and an auto-scaling group of EC2 to scale our system efficiently. We would use a load balancer to automatically distribute the databases among different instances allowing for balance requests and responses. The Redis cache can be further incorporated into our CRUD pipeline to store frequently used queries and return to the user instead of hitting the database saving a lot of resources.

H. Security & Isolation

We provide isolation at a database level between different tenants in our system. Each user's data resides in a completely separate database, reducing the impact of any breaches. Compromising one database affects only that user's data. In addition, we also handle SQL injections and other attacks in the CRUD pipeline and Migrations to avoid malicious code into the databases. We also instantiate the database names to random 128-bit numbers which helps in identifying the databases to the user and also making it anonymous. In the migration process, whenever there is a failure due to improper SQL commands, we automatically rollback to the latest stable version ensuring consistency and safety to the tenant's database

V. Design Choices

A. Asynchronous workflow for migration

The Migration happens asynchronously at the table level instead of a synchronous migration of the entire database. While this may seem counterintuitive to the notion of having one singular unified database, it has several upsides. Some of them are:

1) **Scalability across tables:** Asynchronous migration workflow greatly enhances the scalability of the migration. Synchronising the process across multiple tables can be extremely time and resource intensive as more tables are added to the database.

2) **Optimal Resource utilization:** A decoupled migration process allows for more optimal utilization of resources. It is less likely to require large contiguous chunks of resources.

B. MySQL databases over SQLite

SQLite offers a quick way to spin up databases and perform operations. However, we chose MySQL as the default method to spin up databases to take advantage of MySQL's features and scalability.

Firstly, we observed that the speed difference between MySQL and SQLite is not very significant. We ran tests on our system to determine the latency of database creation for both three frameworks and the results are in the table I:

SQLite	MySQL	RDS
0.005s	0.012s	363.84s

TABLE I
TIME TAKEN BY THE THREE DATABASE FRAMEWORKS TO GENERATE THE DATABASE IN OUR SYSTEM. SQLITE IS MARGINALLY FASTER THAN MYSQL.

As we see in the table above, SQLite is only marginally faster than MySQL. However, MySQL offers enhanced scalability and seamless migration on the backend using libraries like `skeema.io`.

VI. AMAZON VENDOR SYSTEM AS AN ANALOGY FOR OCTODB MULTI-TENANT DBAAS

The OctoDB Multi-Tenant system resembles Amazon's vendor database system. For instance, when a vendor joins Amazon to sell electronics, they are provisioned a default database schema specific to the *electronics* category. This schema includes fields such as `Sl.No`, `Product Name`, `Product ID`, `Price`, and `Description`. Vendors can populate their product data using this schema.

Customizations and Updates

Vendors can customize their schema (e.g., adding a `Discount` column), making it vendor-specific. These custom schemas remain isolated, meaning global updates to the parent schema, like adding a `Manufacturer Warranty` field, do not propagate to customized schemas.

How OctoDB Mirrors This Behavior

OctoDB adopts a similar approach by supporting *flexibility*, *isolation* and *scalability* to the end-user. In addition, the system also follows an asynchronous architecture which makes it highly available for client requests with faster response times.

VII. FUTURE WORK

OctoDB has laid a strong foundation for scalable, flexible, and isolated multi-tenant databases, but there is room for further enhancement:

- **Selective Schema Updates:** Allowing tenants to choose which updates from the parent schema to adopt could strike a balance between customization and consistency.
- **Performance Optimization:** Advanced techniques like adaptive caching, tenant-specific indexing, and smarter query optimizations can improve efficiency in large-scale deployments.
- **Regulatory Compliance:** Adding tools to enforce data residency rules and automate compliance audits would strengthen support for global regulations like GDPR.
- **Multi-Cloud Support:** Enabling seamless operation across multiple cloud providers can improve resilience and reduce dependency on single vendors.
- **Enhanced Security:** Integrating AI-driven anomaly detection and advanced encryption could further secure tenant data.
- **Offline Synchronization:** Many applications require seamless operation in unreliable internet conditions, necessitating offline database synchronization with the central server while resolving data conflicts.

These improvements would help OctoDB stay ahead of evolving user needs while solidifying its role as a robust DBaaS solution.

REFERENCES

[1] Amazon Web Services, "Amazon RDS", [Online]. Available: https://aws.amazon.com/rds/. [Accessed: Dec. 8, 2024].
[2] Google Cloud, "Cloud SQL", [Online]. Available: https://cloud.google.com/sql. [Accessed: Dec. 8, 2024].
[3] Turso, "TursoDB Documentation", [Online]. Available: https://turso.tech/. [Accessed: Dec. 8, 2024].
[4] Supabase, "Supabase Documentation", [Online]. Available: https://supabase.com/docs. [Accessed: Dec. 8, 2024].
[5] Firebase, "Firebase Realtime Database", [Online]. Available: https://firebase.google.com/products/realtime-database/. [Accessed: Dec. 8, 2024].