A methodology for tenant migration in legacy shared-table multi-tenant applications

Guillaume Rosinosky, Samir Youcef, François Charoy, Etienne Rivière

A methodology for tenant migration in legacy shared-table multi-tenant applications

Guillaume Rosinosky[1], Samir Youcef[2], François Charoy[2], and Etienne Rivière[1]

[1] ICTeam, UCLouvain, Louvain-la-Neuve, Belgium
[2] LORIA - Inria, Nancy, France

Abstract. Multi-tenancy enables cost-effective SaaS through resource consolidation. Multiple customers, or tenants, are served by a single application instance, and isolation is enforced at the application level. Service load for different tenants can vary over time, requiring applications to scale in and out. A large class of SaaS providers operates legacy applications structured around a relational (SQL) database. These applications achieve tenant isolation through dedicated fields in their relational schema and are not designed to support scaling operations. We present a novel solution for scaling in or out such applications through the migration of a tenant's data to new application and database instances. Our solution requires no change to the application and incurs no service downtime for non-migrated tenants. It leverages external tables and foreign data wrappers, as supported by major relational databases. We evaluate the approach using two multi-tenant applications: Iomad, an extension of the Moodle Learning Management System, and Camunda, a business process management platform. Our results show the usability of the method, minimally impacting performance for other tenants during migration and leading to increased service capacity after migration.

Keywords: scalability · multi-tenancy · databases · cloud computing

1 Introduction

Software-as-a-Service (SaaS) is the cloud service model with the highest market size [8]. It allows client organizations, or *tenants*, to benefit from turn-key applications. Multi-tenancy is the sharing application instances across multiple tenants [9]. It enables SaaS providers to benefit from economies of scale, consolidating service load for different tenants, and reducing total cost of ownership. Multi-tenancy can be implemented by sharing computing resources (virtual machines (VM) or containers), by sharing application servers, storing data for multiple tenants in the same database or using a combination of these approaches.

For instance, Rightnow, a customer relationship SaaS provider, reported millions of dollars of savings thanks to multi-tenancy [18].

Despite the current trend towards building new applications using microservices [5] and using cloud-native scalable databases [21,24], a large class of SaaS operators still operate applications based on tried-and-tested monolithic applications using a relational (SQL) database to store customer data. In such legacy

applications, multi-tenancy is typically implemented at the database level, either using distinct dedicated databases, dedicated tables or shared tables [28]. We focus in this paper on shared-table multi-tenancy: Database tables are shared between tenants, and a tenant identifier enables queries to distinguish the tenants' rows (i.e., a shared-table schema [2]). SaaS provider Salesforce has been, for instance, using such a shared-table architecture for over ten years [29]. Other examples of applications using shared-table multi-tenancy include learning management such as Iomad (an evolution of Moodle), content management (e.g., Cortex-CMS), business process modelling (e.g., Activiti, Bonita, Camunda), or cloud billing (e.g., CloudKitty).

The application load for multi-tenant applications hosted in the cloud is dynamic. In order to enforce the essential pay-as-you-go model of cloud computing, applications must support elastic scaling operations, whereby new resources are added and removed as the operation workload changes. Typically, scaling out requires starting new application instances and migrating data for certain tenants to this instance. In legacy shared-table multi-tenant applications, this requirement is impaired by two factors. First, there is limited support for scalability operations in traditional relational database engines [14][3]. Second, legacy applications are generally designed on the basis that a single, centralized, and unified view of the entire database is available, which includes in particular information (tables) shared across tenants. Migrating data for a tenant to the new database instance associated with the new application server invalidates this assumption, requiring non-trivial adaptations in the legacy code.

Contributions. We present a non-intrusive and efficient methodology for stop-and-copy tenant migration in legacy multi-tenant applications using the shared-table approach. Our method enables such legacy applications to scale in and out based on tenants' requirements (e.g., the volume of requests or data). It is adapted to the constraints of SaaS providers: (1) it does not require changes to the code of legacy applications, and maintain guarantees on unified views of the database and of common data; (2) it enables migrations of a tenant with no interruption of service for *other* tenants and only minimal and temporary impact for the migrated one; (3) it effectively enables the provider to scale up the number of requests that can be supported after migration and can, reversely, be used for tenant consolidation and resource savings.

Our method leverages external tables and foreign data wrappers (FDW), ISO standard features supported by major relational database engines. These features enable a *unified view* of the database, and *unmodified* queries while allowing data to be stored onto different instances of the database engine hosted by different VMs or containers. We present a SQL-based systematic process for the migration of tenants' data between these instances. We implemented our

---

[3] Database sharding, e.g., using PL/Proxy [22] allows splitting the content of tables over multiple database nodes, but it is not elastic: changing the sharding plan requires to restart the database. Database specific extensions such as the Citus extension for Postgresql or Vitess for Mysql permit automatic scaling. However, they imply switching to specific engines, with their limitations and limited support.

approach for two representative multi-tenant applications: (1) Iomad, a learning management system evolved from the popular Moodle system; (2) Camunda, a business process management system. These two case studies confirm the ability of our method to enable migrations with no change to the application. Our performance evaluation of the applications in a Kubernetes cluster using PostgreSQL shows the effectiveness of our approach. Tenants migrations happen with negligible impact on query performance for other tenants and enable increased service throughput after migration.

The remainder of the paper is structured as follows. Section 2 presents our migration method. Section 3 presents our two case studies, evaluated in Section 4. We discuss related work in Section 5 and conclude in Section 6.

2 A method for tenant migration

We describe in this section our methodology and migration method for multi-tenant software under the shared-table model. We list our design goals (§2.1), present the architecture (§2.2), detail the classification of database tables (§2.3), and describe the tenant migration process (§2.4).

2.1 Design goals

In a multi-tenant application, tenants' resource requirements vary depending on clients activity. These variations are hard to predict and can bring under-utilization or over-utilization of resources, resulting in additional costs for the provider or severed QoS for the customer. A possibility to address this problem is to migrate the corresponding data and software to another server. However, tenant migration in shared-table multi-tenant applications is a complicated process, due to the meddling of tenant data in different rows of common database tables. Besides, some data can be shared between tenants: For instance, global parameters such as the list of tenants cannot be separated between different tenants. This makes the task of consistently backup or export data challenging to do without changes to the application code.

We propose a method for migrating tenants in legacy applications using a relational database and the shared-table model. Its goals are as follows:

- **Non-intrusion**: It should not require changes to the source code of the application, as these are typically expensive to implement and maintain over time. Instead, it may only apply simple transformations of the database schema.
- **Parsimony**: Only the data relevant for a migrated tenant should be transferred between origin and target database instances, to limit the impact of migrations on resource consumption for the provider.
- **Transparency**: Migration operations should have minimal impact on the application clients, and particularly those of the non-migrated tenants. Application or database restart is not desired to implement migration, and the slowdown imposed by a migration operation must be limited.
- **Reliability**: Changes to the database schema, as required by the architecture to enable migrations, should not impact data and queries consistency.
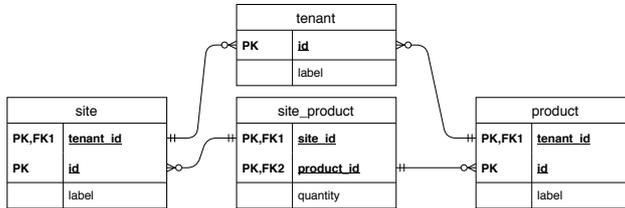
Fig. 1. Multi-tenant stock management application: Entity-Relation schema.

We also make the following assumption of tenant isolation, that is, tenants' queries should be specific for their tenant, i.e. they should not target other tenants [9].

In the remainder of this section, we illustrate our discussion using an example of a multi-tenant stock management application, whose schema is given by Figure 1. This example application allows different tenants (listed in table `tenant`) to trace the availability (association table `site_product`) of products (table `product`) on different sites (table `site`). Tenants can only access their sites and products (e.g., as the primary key for `product` is `<tenant_id,id>` there is no sharing of product identifiers between tenants).

2.2   Overview

The foundation of our method is to split the tables of the application schema between a single *reference database* and multiple *tenant databases*. Each tenant database hosts data for multiple tenants, but data for a given tenant is not split across tenant databases. All tenant databases link with the unique reference database by using *foreign tables*. The reference database contains data common to the whole installation. The application servers connect to the corresponding tenant databases and do not need to be aware of the existence of the reference database accessed through foreign tables. Figure 2 presents an example of deployment with two tenants databases.

Foreign tables are supported by major relational databases [30], and are enabled by Foreign Data Wrappers (FDW), an ISO standard feature for relational databases since SQL/MED (ISO/IEC 9075-9:2003 [11])[4]. Foreign Data Wrappers allow accessing foreign data through *virtual tables*. Foreign data can be tables in a remote database instance, or other data sources accessed through a specific wrapper. Virtual tables behave like regular tables for queries and provide ACID guarantees for foreign data stored in the remote database. It is important to note, however, that foreign tables do not allow the automatic enforcement of

---

[4] Foreign Data Wrappers availability is vendor-dependent. Postgres, MySQL, and MariaDB support the ISO standard. Db2, SQL Server and Oracle do not support it, but offer similar features (with respectively *distributed databases*, *linked servers*, and *database link*). Support of distributed transactions is also vendor-dependent: they are not supported in MySQL, are only partially supported in PostgreSQL and the Open Source engine FederatedX of MariaDB, and fully supported in commercial engines such as Oracle, SQL Server, DB2, and the Spider engine of MariaDB.
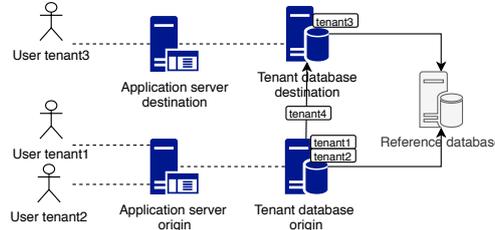
Fig. 2. Application of the method to two tenant databases and migration of tenant 4.

integrity constraints (e.g., foreign keys) between local tables and foreign tables: Such constraints can only be declared between local tables[5]. The transparency of queries using foreign tables directly enables our property of *non-intrusion*: queries in the application do not have to be modified, and we only need to operate changes to the database schema, as we present next.

2.3 Mapping tables to tenant or reference DB

Our method requires classifying which of the tables should be hosted in tenant databases and which should be kept in the reference database, and accessed, therefore, as foreign tables from the former. This classification is application-dependent and cannot be easily automated. We discuss, therefore, guidelines to allow SaaS operators to make informed decisions, as we detail with our two use cases in Section 3.

We propose to classify tables using the following categories. (1) **Active tables** contain tenant-specific data about currently running processes, tasks, events, users, roles, etc. (2) **Temporary tables** contain system-related data linked to the tenant application server, for instance, cache tables. (3) **System tables** contain system-related data concerning the global installation (e.g., the list of tenants). (4) **Archive tables** contain tenant-specific data about past processes, tasks, events, ... Active tables (containing tenant-specific data) can be further split into two categories: (1.1) tables containing a tenant identifier as a field, and (1.2) tables that do not contain such a field.

The SaaS operator must decide on the classification of tables between the reference database and tenant databases. The choice can consider the following principles: (1) **Distributed queries cost:** Distributed queries (i.e., involving tables in the tenant database and the reference database) are more expensive in term of response time than queries for a single database engine instance. (2) **Data volume:** The deployment must seek a balance between the volume of data kept in the reference database and the volume of data hosted by tenant databases. Large volumes of data in the reference database may lead to scalability issues due to its uniqueness, whereas an arbitrary number of tenant databases can be used. On the other hand, large volumes of data in tenant databases increase the migration duration.

---

[5] This inconvenience can be addressed by using database triggers emulating the integrity constraints.
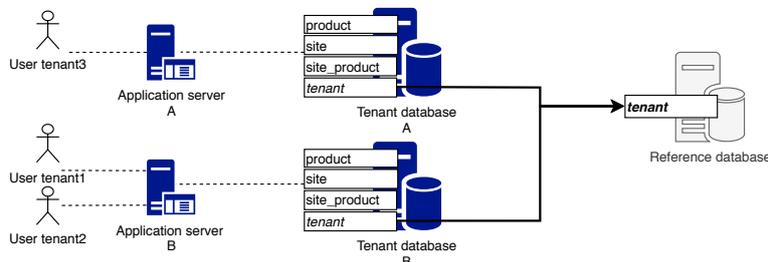
Fig. 3. Multi-tenant stock management application: table placement.

The choice of classification impacts the performance and behaviour of the application. As a general principle, the following is generally a good starting basis: active tables and temporary tables are assigned to the tenant database, and system and archive tables are assigned to the reference database.

We apply this approach to the stock management example of Figure 1. First, table `tenant` is a *system table*: it contains the list of tenants and is global to the system as a whole. Second, `site` and `product` are tenant-specific tables: they should be located on tenant databases. Finally, `site_product` is a tenant-specific table that does not feature a tenant identifier. It can be located either on the reference or tenant database. As frequent queries return the list of products per site or the list of sites holding products, the operator considers that it is preferable to host `site_product` alongside the `site` and `product` tables in the tenant database. Another reason for her choice could be the need to declare integrity constraints between these two tables.

We present our method applied to the tables of Figure 2 in Figure 3. In this case, we keep the tables `site_product`, `site` and `product` on the tenant database. The `tenant` table is located on the reference database, and a permanent foreign table in the tenant database will target it. In the end, an application connecting to the tenant database will be able to access all four tables and issue DQL (Data Query Language), and DML (Data Manipulation Language) queries as if they were accessing a unique database. For tables using sequentially-generated identifiers, collisions could happen in the case of consolidation of tenants in the same database. For reference tables, sequences should be managed at the reference database level. For tenant tables, this issue is manageable with a shift on the sequence for each tenant database in each table to avoid collisions (for instance, keys from 10,000,000 to 19,999,999 from a first tenant, keys from 20,000,000 to 29,999,999 from a second tenant, etc.).

Our methodology supports production features such as database backups, security, or the evolution of schemas. Database backup tools can be configured to use foreign tables. Tables and schemas security access can be tuned in the corresponding tenant databases. Database schemas updates needed by the software evolutions should be done on the tenant and reference databases: Schema upgrade scripts should be split as well and be applied on all databases, and foreign tables should also be updated.

2.4   Migration of a tenant

The use of our proposed evolution of the application schema enables migrations using a series of SQL statements (i.e., in concordance with our objective of non-intrusion, we do not require specific features from the database system itself). Table data stored in the tenant database and specific to the migrated tenant should be inserted in the corresponding table at the destination (which can be, in the case of a horizontal scale-out operation, a freshly-started database instance, or an existing instance in the case of a consolidation). Current transactions on tenants should have been all committed or rollbacked before a migration to avoid data loss. Multi-tenant applications generally feature the possibility to pause or set to "maintenance mode" a specific tenant, and we assume the availability of such a function. When migration occurs, client queries directed at the origin application server should be put on hold during migration and redirected on the destination application server after the migration. Figure 2 illustrates the migration of tenant 4.

The migration process requires the following steps to migrate tenant t from one stack a featuring database D_a and application A_a to stack b formed of database D_b and application A_b:

1. stop the tenant on application A_a and disconnect users;
2. disable foreign key checks for the database connection;
3. for each table in the tenant database, instantiate a temporary foreign table using a foreign data wrapper in database D_a, targeting the corresponding table in database D_b; then, in database D_a, insert data corresponding to tenant t in the foreign tables, thereby inserting it in database D_b;
4. enable foreign key checks for the database connection;
5. if there were no errors, delete temporary foreign tables used for the transfer from database D_a and tenant t data from this database;
6. re-activate the tenant on application A_b. Queries from users of tenant t should now target application A_b;
7. in the advent of an error, remove modifications on A_b and D_b, and reactivate the tenant on application A_a.

The step 3 where we insert tenant-specific data using the temporary foreign table uses regular SQL statements, i.e., `INSERT`, `SELECT`, and `DELETE`. These queries are prepared alongside the schema classification by the SaaS operator deploying the application. In the advent of a fault during the migration, the tenant data remains available at its original location until the migration ends. The system can then be rollbacked to the previous state (i.e. before migration) for the migrated tenant if such a fault is detected.

In our example stock management application, the case of tables `site` and `product` is straightforward: We insert into the temporary foreign tables lines with the corresponding tenant identifier. For the `site_product` table, however, we need to use a `JOIN` with either of the `site` or `product` tables using the tenant identifier. This is necessary to recover the related tenant data, as there is no tenant identifier used in this table. If the temporary foreign tables are named

f_site, f_product, and f_site_product, respectively, the transfer uses the following three queries to migrate tenant of identifier 1 (we omit the corresponding deletion queries that follow on tables site, product, and site_product):

```
INSERT INTO f_site SELECT * FROM site WHERE tenant_id = 1
INSERT INTO f_product SELECT * FROM product WHERE tenant_id = 1
INSERT INTO f_site_product SELECT * FROM site_product sp \
    JOIN site s ON sp.tenantid = s.tenantid WHERE s.tenantid = 1
```

After performing table classification and preparing parameterized table subqueries, we can automatize the operations of schema creation and migration. We developed a configurable tool for this purpose, used in our experiments (§ 4).

Only active data is migrated, thus reducing the duration of migrations compared to a regular backup/restore operation. However, tenants with a high quantity of active data will take longer to be migrated. This duration needs to be taken into account depending on QoS needs for migrations. In this matter, wisely choosing which tables should be placed on the tenant and reference databases is an important step, as we discuss in the next section with two real-world applications.

3   Case studies

In this section, we detail our experience in enabling tenant migrations for two shared-table multi-tenant applications: Iomad, a learning management system and Camunda, a business process management engine.

3.1   Iomad

Iomad (https://www.iomad.org) is an open-source *fork* of Moodle, the popular Learning Management System. Iomad adds features to Moodle, enabling its use in a SaaS context, including multi-tenancy, better reporting, and e-commerce features. As Moodle, Iomad is developed in PhP and uses a traditional database management system to persist its data, such as Mysql, Postgresql, or SQL server. Multi-tenancy in Iomad is implemented using a limited shared-table approach by the addition of tenant-specific tables to the original Moodle schema, without tenant identifiers in tenant-related tables.

The schema of Iomad contains 457 tables. We proceeded as follows for the classification. We first identified the company table, containing the tenants. Tenants are linked to users and courses by association tables. Only a handful of tables use tenant identifiers as a foreign key, as should have been expected under the shared-table model. However, we could trace back a large fraction of the association between the core tenant-specific tables and corresponding data in the other tables by following foreign keys dependencies. We selected first these tables (tenant and the associations to user and course) as **system tables** for storage in the reference database.

We then identify **active tables** and **temporary tables**, using two approaches to perform the classification: **(1)** we selected tables having fields corresponding to users or courses (for instance: user, userid, course, courseid,

etc.) for tenant databases; **(2)** we considered tables involved in the GDPR-compliance [19] feature of Iomad, in order to select tables containing user-specific data, and assigned these tables to the tenant database.

We leveraged the Moodle's entity-relationship schema extracted by Green [7] to classify the tables. In total, 314 tables are assigned to tenant databases. All others (i.e., 143 tables) are assigned to the reference database as **system tables**.

We note the following two limitations, due to the implementation of multi-tenancy in Iomad:

- In Iomad, users and courses can be associated with multiple tenants. With our classification and split, we lose this possibility. Users and courses should be created independently for each tenant. The independence of tenants is a necessary feature for multi-tenant software [9]. This is what we believe to be the normal mode of operation for shared-table multi-tenant applications, and the impact should be limited in practice;
- Identifiers are serial-based, and the tenant identifier is not part of the primary key of each table. When scaling in and consolidating tenants from two to a single database, there is a risk of collision between tenant identifiers. This risk could be mitigated by using appropriate serial ranges, or UUID primary keys, but we did not implement these changes as they would have no impact on performance.

3.2   Camunda

Business Process Modeling (BPM) allows companies to represent, execute, and analyze business processes. A BPM schema is a graph representing a sequence of business tasks. These tasks can be automated (e.g., a call to a web service) or manual (e.g., a form that a human operator should fill). Business processes embed variables that can be modified by these different tasks.

Numerous commercial BPM tools exist, such as IBM Business Process Manager, Oracle Business Process Management. Open-source alternatives include Bonita BPM, Activiti, and Camunda, one of the targets of our experiments. We focus here on the usage of the BPM engine that executes the tasks of business processes. Camunda, being a multi-tenant BPM solution, is a perfect target for our method.

The database schema of Camunda 7.8.0 contains 46 tables. In Camunda, tenants are listed in table `act_id_tenant`, that we classify as a **system table**. Most of the other tables include a tenant identifier as part of their primary key, as a field `tenant_id`. We selected for tenant databases as **active tables** all tables containing such a `tenant_id` field, with two exceptions for tables `act_ge_bytearray` and `act_re_deployment`. These two tables are, indeed, necessary for the application bootstrap and should be classified as **system tables**. As a result, we keep them in the reference database for common access by all tenant databases and all application servers. In total, we select 35 tables for storage in the tenant databases and 11 tables for storage in the reference database. We note that the risk of tenant id collision upon a consolidation we mentioned for Iomad is not present here: Tenant identifiers are part of the primary keys of each tenant table; tenants are well isolated.

4   Evaluation

While our use cases show that legacy multi-tenant applications can have their schema adapted to support tenant migration under our proposed architecture, we are interested in the experimental evaluation reported in this section in evaluating its cost and performance. We extend a migration benchmark that we introduced in our previous work [16] to answer the following research questions:

1. What is the overhead of porting a legacy application's data schema to comply with our architecture?
2. What is the duration of a tenant migration, and how does it impact performance for clients of that tenant?
3. What effect do migrations of a tenant have on the performance experienced for other tenants?
4. Is tenant migration successful in enabling scale-out, i.e., a higher service capacity after migrating a tenant?

As our main performance metric for questions 1, 3 & 4, we consider the response time as experienced by the client application (*i.e.*, in response to HTTP queries accessing various features of the application). We further consider migration time from the SaaS operator's perspective when answering question 2.

We use JMeter as a load injection tool. For Iomad, we extended Moodle's load tests, which include HTTP user queries for login, course display, forum display, response and attachment sending, and logout operations. The Iomad database is pre-filled with users and tenants data prior to load injection. For Camunda, we developed a JMeter scenario emulating a user creating and executing a process containing a BPM human task (i.e., a task that must be filled and validated by an operator). The scenario claims the task for the user and emulates its processing. As for Iomad, we pre-fill the database with the process definition and the tenant-specific data prior to load injection.

4.1   Experimental setup

We run all tests in a Kubernetes cluster hosted on the Azure IaaS cloud. We use eight 4-core Azure D3 v2 instances for hosting application containers, and one 2-core B2ms instance for the Kubernetes master. Our deployment includes (1) the host reference database; (2) two application *instances*, origin and destination, each composed of two nodes: a web server and a tenant database; (3) two load injectors; (4) an experiment orchestrator. We developed a script allowing to instantiate the different databases and application instances, orchestrate experiments, and trigger migrations. This script is generic and applies to both applications. We target reproducible research: All our deployments are described using Helm charts and Argo workflows. The source code and experimental data are open-source and available online[6].

We use a distributed architecture with an origin and destination *instances* and a reference database. We make experiments following the scenarios detailed

---

[6] https://github.com/CloudLargeScale-UCLouvain/legacy-sql-migration

Name	Distributed ?	Application	Orig.T	Migr.T	Dur.O	Dur.M	RPS	RQ
`10-nofdw`	no	Iomad	1	0	300	-	10-30	1
`10-nofdw`	no	Camunda	1	0	300	-	50-150	1
`20-nofdw`	no	Iomad	2	0	300	-	10-30	1
`20-nofdw`	no	Camunda	2	0	300	-	50-150	1
`10-fdw`	yes	Iomad	1	0	300	-	10-30	1
`10-fdw`	yes	Camunda	1	0	300	-	50-150	1
`20-fdw`	yes	Iomad	2	0	300	-	10-30	1
`20-fdw`	yes	Camunda	2	0	300	-	50-150	1
`1M-fdw`	yes	Iomad	0	1	-	60-300→0	30	2
`1M-fdw`	yes	Camunda	0	1	-	60-300→0	200	2
`1O1M-fdw`	yes	Iomad	1	1	660	300→300	10-30	3,4
`1O1M-fdw`	yes	Camunda	1	1	660	300→300	50-150	3,4

Table 1. List of experiments. `Orig.T` specifies the number of tenants that start, and stay, in the origin instance. `Migr.T` is the number of tenants that start in the origin instance and later migrate to the destination instance. `Dur.O` is the time during which a workload of `RPS` requests per second per tenant is injected on all tenants staying on origin instance. It lasts during the entire run. `Dur.M` is the time during which a workload of `RPS` requests per second per tenant is injected on the migrated tenant (symbol $\rightarrow$ denotes the injection of this load before, and after, the migration). `RQ` is the corresponding research question.

in Table 1, with variations on the numbers of fixed tenants on the origin *instance* (origin tenants), and tenants originally on origin *instance*, and then migrated on destination *instance* (migrated tenants). Similar runs are launched for Iomad and Camunda. Based on our observation of the two applications in our test infrastructure, we select the HTTP input query throughput. The max throughput for Iomad is the one where the application servers' CPU usage reaches 100%. For Camunda, it is the saturation point for our two injector nodes. We report, for each scenario and RPS value in the considered ranges, aggregate distributions over five individual runs.

For scenarios involving a tenant live migration (`1M-fdw` and `1O1M-fdw`) we consider one or two tenants originally with the origin instance. One tenant is migrated from the origin to the destination during the experiment. In the `1M-fdw`, we only apply queries to the migrated tenant to evaluate the impact of migration on the migrated tenant performance, for multiple durations of injection (this permitting to observe different tenant sizes), while in configuration `1O1M-fdw` we apply workload to both tenants, and migrate after 300 seconds of injection. It allows evaluating the impact of the migration operation on the performance of the non-migrated tenant.

4.2   Results

Figure 4 presents an evaluation of the overhead of using the distributed architecture, enabling live migrations through the use of foreign tables (denoted as "split"), versus the same application using a regular one database deployment (denoted as "single"). With a single database, Iomad response times increase drastically beginning with 40 RPS, with a 90th percentile reaching over 3 sec-
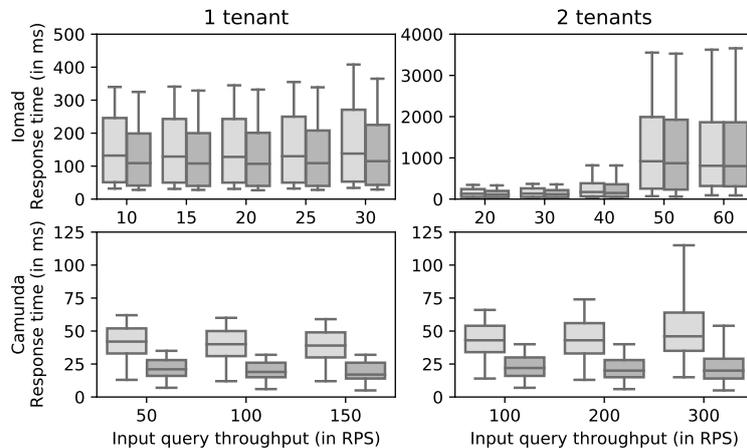
Fig. 4. Comparison of the split architecture performance with the original architecture (experiments `10-*` (1 tenant) and `20-*` (2 tenants)). Light grey is the *split* installation, dark grey the *single* installation. The X axis represents the cumulated throughput from all tenants, the Y axis the distribution of the latencies, with the 10th and 90th percentiles represented by the whiskers.

onds for 50 and 60 RPS, where the system becomes saturated. Camunda median response times slightly increase while staying below 50 ms. The performance of Iomad with a split database is moderately degraded compared to a single deployment, with an additional 20 to 30 ms median response time over the 120 to 150 ms of the single-database configuration. For Camunda, response times are doubled, yet remain low and consistent. These increases are as expected: They are the results of some of the relational queries needing to access data through foreign tables and two database instances. They represent a baseline cost for being able to enable live migrations, that will be compensated by the scalability they allow in large deployments.

Our evaluation of the scalability improvement enabled by live migrations is in figure Figure 5. We present the latency of operations for the tenant staying at the origin (origin tenant) and for the migrated tenant. As expected, performance after the migration is comparable to the performance would both tenants be deployed on separate instances directly. It results in increased service throughput for Iomad: after migration, the application supports 60 RPS with acceptable latencies, in contrast with the maximum of 40 RPS with a single-database configuration. Gains for Camunda are more modest than Iomad's, yet we can observe diminished response times and better throughput scaling. We explain the fact that the latencies for the origin tenant are slightly higher than for the migrated tenant by the additional accumulated data during the migration, only applying to the origin tenant.

We evaluate the live migration duration in Figure 6 using configuration `1M`. The migration time for Iomad is around 11 seconds and slightly increases for longer injection times prior to migration (accumulating more data through more
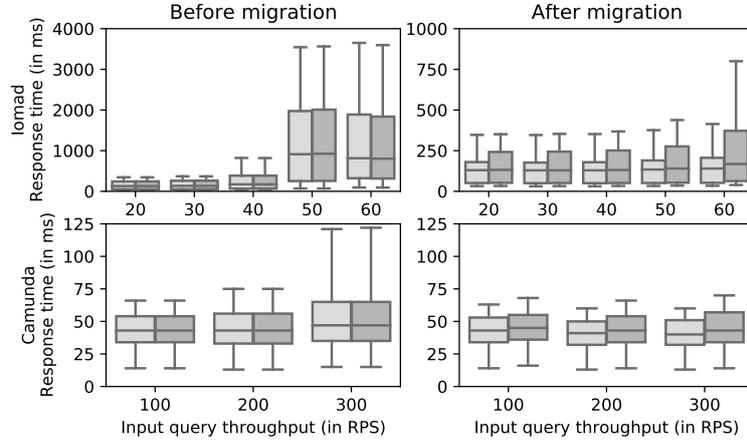
Fig. 5. Performance gain with migration (experiments `101M-*`). The migrated tenant is shown in light grey, the origin tenant in dark grey.
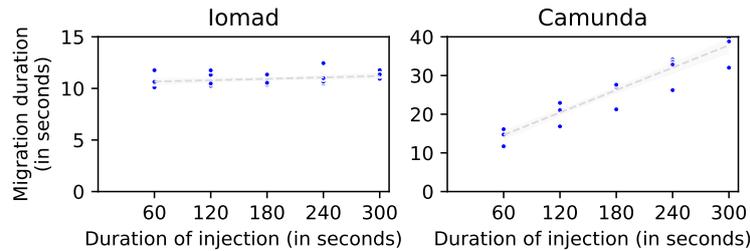

Fig. 6. Duration of injection vs. duration of migration (experiments `1M-*`) represented by the dots. The corresponding linear regression is presented by the dashed line.

requests). With Camunda, migration duration increases linearly with the duration of the injection before this migration. This is a direct effect of the additional volume of data to be migrated, resulting from the additional queries (a total of 60'000 queries in 300 seconds in the case of Camunda, 9'000 for Iomad).

Migrating a tenant involves copying data from the origin database to the destination database. It can provoke negative effects on the response time of other hosted tenants. We evaluate in Figure 7 the impact of a tenant migration on the performance of the other tenants staying at the origin. Our evaluation considers the latencies for both applications, 20 seconds before the migration, and during the migration operation. The latency is only marginally impacted for Iomad, especially in high throughputs, and not perceptible for Camunda. The probable cause of the small effects on Camunda compared to Iomad is that the instalment does not approach saturation.

Discussion Our split-database architecture successfully enables live migration in multi-tenant shared-table applications. Migration operations enable scaling out and supporting higher service throughput, while only minimally impacting the performance for non-migrated tenants, satisfying our objective of *trans-*
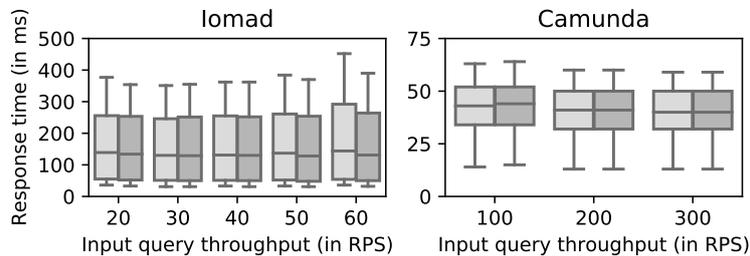
Fig. 7. Effects of migration on colocated tenants (experiments `101M-*`, 20 seconds before migration, and during migration). Performances during migration are in light grey, before migration in dark grey.

parency. Migration time is, unsurprisingly, impacted by the volume of data. As the transfer happens through a series of sequential SQL requests, migration latency (and, therefore, the unavailability period for the migrated tenant) can be high for large tenants. This is a compromise due to our property of *non-intrusion*, that highlights the importance of properly classifying tables for the tenant database, i.e., respecting the *parsimony* property. Using parallel transfers (e.g., using several threads to transfer data from origin tables to the destination tables) could speed up the process, but we leave this optimization to future work.

5   Related work

We review related work on migration in databases, database sharding, and related approaches to multi-tenancy.

Work towards migrations in multi-tenant databases mainly focused on shared-process database [3,6,10], in engines such as H-Store or Elastras. In the industry, Azure proposes a scalable version of Postgresql named *Hyperscale* and based on the Open Source *Citus* Postgresql extension [4]. This approach, like ours, is based on distribution keys and works well with multi-tenant applications. However, join operations on distributed tables are not allowed, thus making adaptations of the source code necessary for most applications. Oracle proposes multi-tenant live migration features [15]. Their solution is based on Real Application Cluster, Oracle's distributed database product, and enables service relocation. However, it provides only *full* database migration (shared disk architecture) [12] and cannot be applied to shared-row multi-tenancy scenarios. Google provides distributed and highly consistent databases F1 and Spanner [21]. Spanner allows redistribution of data across shards. These database systems are only available on Google Cloud, and lack some RDBMS features such as sequences, or DML queries on non-key fields. Vitess is an evolution of MySQL that provides automatic sharding [26]. It does not, however, comply totally to the MySQL feature matrix and applications require code adaptation. Compared to our approach, these works need the usage of specific database engines, and adaptations in the code base. Our migration methodology runs natively on legacy database engines.

Microsoft Azure offers several tools for shared-table multi-tenant applications. The Azure SQL database, based on SQL Server, includes an Elastic Database client library [13]. A tool called *SplitMerge* enables live shard migration based on an identifier or range of values in specific fields. Tables for a sharded tenant can be moved depending on a classification over three table types: shard tables, reference tables and other tables. During a migration, shard tables are copied based on the key of the tenant, reference tables are fully copied (leading to errors if these tables are already initialized), and other tables are ignored. This approach, while interesting, require software adaptation and the use of the specific Elastic database client. It also needs a specific database for tenant management. Our architecture does not require modifications to applications' codebase, as common tables between tenants stay available with the usage of foreign tables. Ghostferry [20] is a live migration tool based on Github's `gh-ost` tool for MySQL. Ghostferry allows copying a list of tables from an origin to a target database and offers filtering capabilities. It is based on the replication feature of MySQL databases and uses the binary log of the database for the migration operation. There are some limitations, such as the need of integer primary keys, the need for row-based replication on the source server, and it is specific to MySQL. Ghostferry does not address the system tables problem in multi-tenant applications as we do in our architecture. Jetpants by Tumblr [25] is an automation toolkit to automatize sharding operations. However, it does not permit to keep the codebase of a legacy application untouched. We note that both Ghostferry and Jetpants could be coupled with our architecture to implement the actual migration operation.

We finally review approaches that are orthogonal or complementary to our work. Slacker [1] is a middleware for automating migration upon SLA violations. Slacker can leverage automated backup or other live migrations methods. Numerous authors proposed models for multi-tenant resource allocation and tenant placement [17,27,23]. The goal of these works is to minimize operational costs and limit the number of quality of service violations. They do not consider, however, the means used for the live migration operations, or the constraints the support for live migration poses on the architecture.

6   Conclusion

We presented a novel migration method for shared-table multi-tenant applications. Our approach is adapted to the needs of SaaS providers wishing to offer scalable installments of legacy applications based on relational databases. Our proposal does not require changes to the application source code but only to its data schema. We proposed guidelines for classifying which tables should be included in the migratable part of this schema, and we presented the application of these guidelines to two real-world multi-tenant applications. Our experimental evaluation in the cloud shows a real gain in capacity for these applications and low overhead. In our future work, we wish to explore the automation of the classification of tenant tables, e.g., using learning techniques and data from actual deployments, and the improvement of the performance of the migration itself through its parallelization.

References

1. Barker, S., Chi, Y., Moon, H.J., Hacigümüş, H., Shenoy, P.: Cut me some slack: Latency-aware live migration for databases. In: 15th international conference on extending database technology. EDBT, ACM (2012)
2. Chong, F., Carraro, G., Wolter, R.: Multi-tenant data architecture. MSDN Library, Microsoft Corporation pp. 14–30 (2006)
3. Das, S., Agrawal, D., El Abbadi, A.: ElasTraS: An elastic, scalable, and self-managing transactional database for the cloud. ACM Transactions on Database Systems **38**(1), 1–45 (2013)
4. Data, C.: citus: Scalable PostgreSQL for multi-tenant and real-time analytics workloads (2020)
5. Di Francesco, P., Lago, P., Malavolta, I.: Migrating towards microservice architectures: an industrial survey. In: International Conference on Software Architecture. ICSA, IEEE (2018)
6. Elmore, A.J., Arora, V., Taft, R., Pavlo, A., Agrawal, D., El Abbadi, A.: Squall: Fine-Grained Live Reconfiguration for Partitioned Main Memory Databases. In: ACM SIGMOD International Conference on Management of Data (2015)
7. Green, M.: Moodle Database schema (2020)
8. IDC: Worldwide Semiannual Public Cloud Services Tracker
9. Kalra, S., Prabhakar, T.: Multi-tenant quality attributes to manage tenants in saas applications. In: 2020 IEEE International Conference on Software Architecture Companion (ICSA-C). pp. 83–88. IEEE (2020)
10. Lin, Y.S., Pi, S.K., Liao, M.K., Tsai, C., Elmore, A., Wu, S.H.: MgCrab: transaction crabbing for live migration in deterministic database systems. Proc. of the VLDB Endowment **12**(5), 597–610 (2019)
11. Melton, J., Michels, J.E., Josifovski, V., Kulkarni, K., Schwarz, P.: SQL/MED: A Status Report. SIGMOD Rec. **31**(3) (2002)
12. Michael, N., Shen, Y.: Downtime-free live migration in a multitenant database. In: Technology Conference on Performance Evaluation and Benchmarking. TPCTC, Springer (2014)
13. Microsoft: Scaling out - Azure SQL Database (2019), `https://docs.microsoft.com/en-us/azure/sql-database/sql-database-elastic-scale-introduction`
14. Moniruzzaman, A.B.M.: NewSQL: Towards Next-Generation Scalable RDBMS for Online Transaction Processing (OLTP) for Big Data Management. International Journal of Database Theory and Application **7**(6), 121–130 (2014)
15. Oracle: Database Live-Migration with Oracle Multitenant and the Oracle Universal Connection Pool (UCP) on Oracle Real Application Clusters (RAC) (2015), `http://www.oracle.com/technetwork/database/multitenant/learn-more/pdblivemigration-2301324.pdf`
16. Rosinosky, G., Labba, C., Ferme, V., Youcef, S., Charoy, F., Pautasso, C.: Evaluating Multi-tenant Live Migrations Effects on Performance. In: On the Move to Meaningful Int. Sys. OTM, Springer (2018)
17. Rosinosky, G., Youcef, S., Charoy, F.: A genetic algorithm for cost-aware business processes execution in the cloud. In: International Conference on Service-Oriented Computing. ICSOC, Springer (2018)
18. Schaffner, J., Jacobs, D., Kraska, T., Plattner, H.: The Multi-Tenant Data Placement Problem. In: 4th International Conference on Advances in Databases, Knowledge, and Data Applications. DBKDA (2012)

19. Shah, A., Banakar, V., Shastri, S., Wasserman, M., Chidambaram, V.: Analyzing the impact of GDPR on storage systems. In: 11th USENIX Workshop on Hot Topics in Storage and File Systems. HotStorage (2019)
20. Shopify: ghostferry: The swiss army knife of live data migrations (2018)
21. Shute, J., Ellner, S., Cieslewicz, J., Rae, I., Stancescu, T., Apte, H., Vingralek, R., Samwel, B., Handy, B., Whipkey, C., Rollins, E., Oancea, M., Littlefield, K., Menestrina, D.: F1: a distributed SQL database that scales. Proceedings of the VLDB Endowment **6**(11) (2013)
22. Suursoho, S., Kreen, M.: PL/Proxy: Function-based sharding for PostgreSQL
23. Taft, R., Lang, W., Duggan, J., Elmore, A.J., Stonebraker, M., DeWitt, D.: Step: Scalable tenant placement for managing database-as-a-service deployments. In: 7th ACM Symposium on Cloud Computing. SoCC (2016)
24. Taft, R., Sharif, I., Matei, A., VanBenschoten, N., Lewis, J., Grieger, T., Niemi, K., Woods, A., Birzin, A., Poss, R., et al.: CockroachDB: The Resilient Geo-Distributed SQL Database. In: ACM SIGMOD International Conference on Management of Data (2020)
25. Tumblr: jetpants: MySQL toolkit for managing billions of rows and hundreds of database machines (2020)
26. Vitess: A database clustering system for horizontal scaling of MySQL (2020), https://vitess.io/
27. Wang, F., Li, J., Zhang, J., Huang, Q.: Research on the multi-tenant placement genetic algorithm based on eucalyptus platform. In: 12th Intl. Conf. on Computational Intelligence and Security. CIS, IEEE (2016)
28. Wang, Z.H., Guo, C.J., Gao, B., Sun, W., Zhang, Z., An, W.H.: A study and performance evaluation of the multi-tenant data tier design patterns for service oriented computing. In: Intl. Conf. on e-Business Eng. IEEE (2008)
29. Weissman, C.D., Bobrowski, S.: The design of the force.com multitenant internet application development platform. In: ACM SIGMOD International Conference on Management of data (2009)
30. Yu, X., Gadepally, V., Zdonik, S., Kraska, T., Stonebraker, M.: FastDAWG: Improving data migration in the BigDAWG polystore system. In: Heterogeneous Data Management, Polystores, and Analytics for Healthcare. Springer (2018)