Towards a Technique for Extracting Relational Actors from Monolithic Applications

Rodrigo Laigner[1], **Sérgio Lifschitz**[1], **Marcos Kalinowski**[1], **Marcus Poggi**[1], **Marcos Antonio Vaz Salles**[2]

[1]Departamento de Informática – PUC-Rio, Brazil
{rlaigner,sergio,kalinowski,poggi}@inf.puc-rio.br

[2]Department of Computer Science (DIKU) – University of Copenhagen, Denmark
vmarcos@di.ku.dk

Abstract. *Relational actors, or reactors for short, integrate the actor model with the relational data model, providing an abstraction for enabling actor-relational database systems. However, as a novel model of computation for databases, there is no extensive work on reasoning about reactor modeling. To close this gap, this paper aims to propose as well as evaluate a technique to extract reactors from a monolithic system. For evaluation, we selected a REST-based open-source OLTP system in which a decomposition to microservices was conducted and applied our technique on its predecessor monolithic version. Our technique led to the same set of decisions, regarding table and behavior selection, taken by experts when decomposing the same system into microservices. The proposed technique can be seen as a first step towards supporting practitioners in decomposing OLTP systems into reactors.*

1. Introduction

Reactive systems and microservices constitute a new trend in the development of data-driven software applications [Boner et al. 2019, Hasselbring and Steinacker 2017]. To support these new application development needs, proposals have been put forward to integrate databases and actor systems [Bernstein et al. 2017, Shah and Salles 2017]. A key idea in these proposals is to support asynchrony and distribution in application design while still providing for desirable database functionality such as declarative querying and transactions.

In particular, actor-relational database systems aim at integrating actor constructs for encapsulation and concurrency at the level of the programming interface of a relational database management systems (RDBMS) [Shah and Salles 2017]. Relational actors (or reactors, for short) consist of a concrete programming model for actor-relational DBMS [Shah and Salles 2018]. A reactor encapsulates relations in its state, and allows for state manipulations to be executed only by asynchronous function calls returning futures. While reactors appear to be a promising programming model for actor-relational databases, there is limited guidance in how to design applications based on reactors. In [Shah and Salles 2018], the authors assert that "an interesting avenue for future research is to explore an analytical machinery for modeling and comparing the quality of reactor database designs."

Thus, there is a need for a systematic approach to derive a reactor database design. Towards this end, this work presents a technique for extracting reactors from a monolithic

application with Online Transaction Processing (OLTP) characteristics. The technique considers variables such as the access frequency of application entry points and coupling among tables. Importantly, we show that our technique can be used to suggest how to break a REST-based monolithic application into reactors.

The document is organized as follows. Section 2 provides background, presenting reactor database systems, the architectural style REST, and layered architecture. Section 3 discusses related work. Section 4 presents the technique proposed. Section 5 provides an evaluation of the technique based on an open-source OLTP application. The limitations are presented in Section 6. Finally, Section 7 presents concluding remarks.

2. Background

This section introduces background needed to contextualize our proposed technique.

Reactors. Stored procedures in RDBMS [Rowe and Stonebraker 1987] generally do not present constructs to explicitly parallelize operations on the database. This way, practitioners must rely on high level programming languages in the middle tier to handle parallelism in application logic. However, the complexity in source code entailed by the latter is substantial, raising the need for better abstractions [Sutter and Larus 2005]. In order to overcome this problem, [Shah and Salles 2018] proposed a relational actor (reactor) database system, which is derived from the actor model [Agha 1986].

In the actor model, [Shah and Salles 2018] argue that "each communication is described as a message arriving at a computational agent called an actor". In response to a received message, an actor can also send messages to other actors. Reactors abstract such messaging for concurrency and parallelism through asynchronous function calls that trigger operations on encapsulated state. At the same time, reactors provide high-level database abstractions for application programming. Importantly, "the state of each actor is abstracted by a set of relations and application functions employ declarative queries against relations" [Shah and Salles 2018]. Additionally, application functions can be executed transactionally with reactors.

An excerpt of interaction among reactors adapted from [Shah and Salles 2018] is shown in Figure 1. The figure depicts the use case of an order, triggered by the function call *add_items* in the *Cart* reactor, which transactionally retrieves customer information from a disjoint *Customer* reactor. After a separate transactional call to *checkout*, the reactor *Cart* then records a *store visit* in the same *Customer* reactor.

Representational State Transfer. Representational State Transfer (REST) is an architectural style described by Roy Fielding [Fielding 2000] aimed at designing distributed systems over the World Wide Web and has been established as a pattern in industry. According to Fielding [Fielding 2000], "the key abstraction of information in REST is a resource", on which "REST components perform actions on a resource by using a representation to capture the current or intended state of that resource". The action and the intended state of a resource can be understood as a contract established between two communicating parties.

Layered architecture. Layered architecture is a software architecture pattern aimed at distributing responsibilities over different components in an application in order to achieve high cohesion and low coupling among components [Fowler 2015]. According

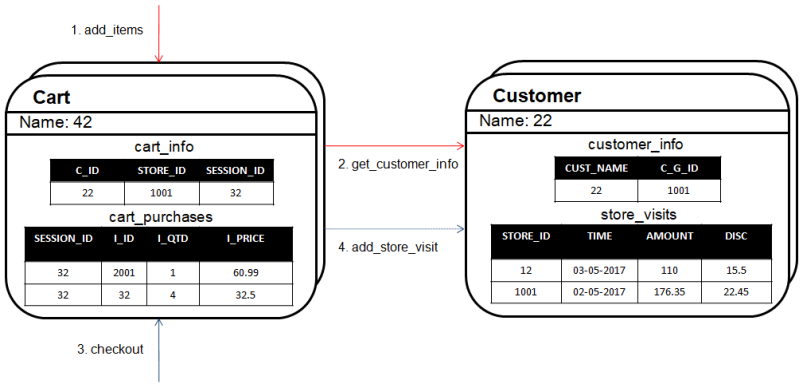

Figure 1. Communication between relational actors

to Richards [Richards 2015], "components within the layered architecture pattern are organized into horizontal layers, each layer performing a specific role within the application (e.g., business logic)". This way, a common approach for layered architecture is dividing the application into the four layers described below.

The **Presentation** layer encodes a set of rules that enable external clients to communicate with the system. This layer thus abstracts the *entry points* of the system, e.g., a REST API resource. The **Business** layer is composed of smaller subsystems. It is characterized as the core of the system, being responsible for handling application logic. The **Repository** layer is composed by a set of modules that primarily handle communication with the database system. This layer commonly deals with data access and data manipulation in SQL or through abstractions, such as provided by the Hibernate framework.[1] Finally, the **Database** layer is the DBMS.

Monolithic Applications. Monolithic systems adopt an architectural style in which modules and subsystems are integrated and cooperate in a centralized manner. By contrast, in a microservices architecture, the system is subdivided into several services, each of which usually represents a single concern within the system [Richards 2015].

3. Related Work

As there is no previous work on decomposing applications into reactors, we observe that the work most related to our proposal is on decomposing modules into thin services, the so-called microservices. As there is a substantial body of work on this topic, we focus on those studies that present similar approaches in relation to our proposed technique.

Mazlami et al. [Mazlami et al. 2017] define an extraction model where the starting point is the source code. Then, class files, change history, and developers that contributed to source code are identified. Employing different coupling strategies (e.g., terms are extracted from class files in order to enable a semantic coupling strategy), a graph representation is created representing the degree of coupling among classes in the monolithic system. A graph clustering algorithm is used to obtain candidates for microservices. The authors [Mazlami et al. 2017] argue that “the unsolved problem of how to share or assign pre-existing databases to different services remains a limitation” of their work.

[1] https://hibernate.org

Gysel et al. [Gysel et al. 2016] propose a service decomposition based on a set of coupling criteria and system specification artifacts, such as domain models and use cases. A supporting tool framework (Service Cutter) was developed and decomposed services were "represented as an undirected, weighted graph to find and score". For instance, the authors [Gysel et al. 2016] define System Specification Artifacts, such as use cases and domain models, as an input of Service Cutter. We consider these artifacts insufficient to assess application workload, particularly in the context of OLTP applications. In contrast to our approach, their technique [Gysel et al. 2016] does not present primitives for considering the workload of the application neither extraction of application logic.

Levcovitz et al. [Levcovitz et al. 2016] present a technique for decomposing microservices from a monolithic application that is divided in three parts: a client side user interface, a server side application, and a database. Levcovitz et al. [Levcovitz et al. 2016] work decomposes the application based on business areas, which they describe as being "responsible for a business process". We consider this an inadequate selection once a business unit is an abstract concept and might not lead to optimal decomposition, as our work present, in regard to workload and behavior distribution.

In addition to the above, numerous approaches have investigated horizontal and vertical partitioning in object-oriented [Bellatreche et al. 2000] or relational databases [Pavlo et al. 2012]. In contrast to our proposal, these approaches focus exclusively on database access logic and do not consider layered architectures with application code across tiers. Lastly, Wang et al. [Wang et al. 2019] discuss the modeling of Internet-of-Things applications with actor-oriented databases. However, the work focuses on actor databases with an object-oriented data abstraction, being thus only partially applicable to relational actors. Furthermore, the modeling guidelines presented are aimed at the development of new applications, as opposed to decomposition of existing monoliths.

4. Proposal

This work aims to extract reactors from a monolithic system. Therefore, we rely on the concept of *entry points* of a system, which are contracts that external clients should follow in order to communicate with the application. Several approaches, such as RPC and SOAP, have been defined to enable distributed systems communication. Thus, in order to establish a common pattern for systems communication, our technique targets OLTP web applications that communicate through REST APIs.

In addition, in order to formalize the system under analysis, we extend Levcovitz et al. [Levcovitz et al. 2016] system formalization. A monolithic system S is represented by a quadruple (I;B;R;D), where I = { int_1, int_2, ..., int_n } is a set of interfaces (REST resources) from the presentation layer, B = { bf_1, bf_2, ..., bf_n } is a set of business functions, R = { rf_1, rf_2, ..., rf_n } is a set of repository functions, and T = { tb_1, tb_2, ..., tb_n } is a set of database tables. Figure 2 depicts a representation of the layered architecture considered in this work. Our technique for extracting reactors from a monolithic application comprises five phases, which are described as follows.

System dependency graph construction. This phase comprises building a dependency graph G = (V, E) representing the execution flow for each interface described in the presentation layer. An execution flow represents the branches associated with a given interface, where vertices represent available functions and resources.

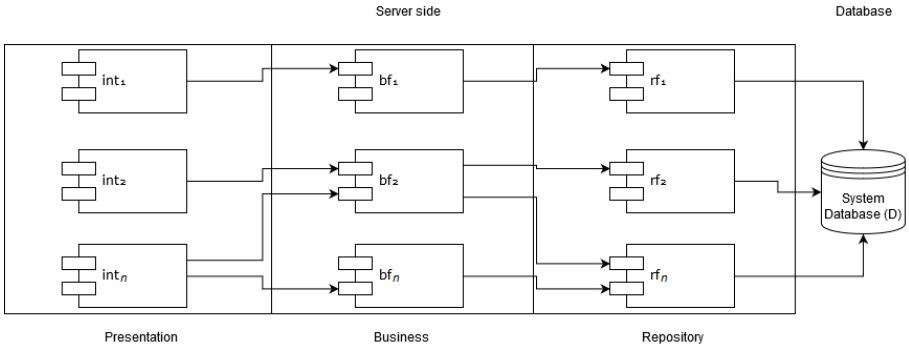

Figure 2. Layered architecture considered in this work

Vertices are represented by the following elements: (i) interfaces ($int_i \in$ I) $\forall i$; business functions ($bf_i \in$ B) $\forall i$; repository functions ($rf_i \in$ R) $\forall i$; database tables ($tb_i \in$ T) $\forall i$. Furthermore, the edges represent: (i) interface calls to business functions; (ii) calls among business functions; (iii) calls from business functions to repository functions; (iv) calls among repository functions; and (v) table accesses from repository functions.

Profile data collection. After building the dependency graph, it is important to know how frequently each REST resource (interface) is called. Profiling can be achieved by introducing dynamic instrumentation in the application or by techniques such as aspect-oriented programming, in a non-intrusive manner [Kiczales et al. 1997]. When such profiling data are not available, we suggest reasoning about the resources and assigning weights related to their expected frequency. Based on this step, vertices representing interfaces are weighted by a frequency cost, the so-called *access frequency*.

Table coupling identification. In the context of software development, coupling is commonly used for assessing source code structural quality [Olbrich et al. 2010]. On the other hand, since relational actors operate on data, it is necessary to define coupling in the context of tables.

Table coupling, as used in this work, thus concerns associations between tables. Two tables are associated by a foreign key (FK) if a FK is present in one or both relations. In case of a many-to-many association, we assume the FK is present on both relations and discard the join table built in consequence of physical design. A formalization of the information on table coupling is provided below.

$$coup_{i,j} = \begin{cases} 1, & \text{if tables } i \text{ and } j \text{ are associated} \\ 0, & \text{otherwise} \end{cases} \qquad \forall (i,j) \in T \times T$$

Reactor table identification. Towards identifying tables that would compose a reactor, the information collected in previous steps is employed to identify an optimal distribution of tables and interfaces among clusters. The formulation is based on the clustering problem found in optimization studies [Du and Pardalos 1998], in which the goal is to identify a set of clusters that respect a given connectivity measure among vertices.

It is necessary to establish a limit on the amount of workload a reactor can sustain while properties, such as coupling, are maintained within reactors. Given the access frequency, the table coupling, and the maximum workload limit, our technique aims to

divide workload among clusters by properly allocating tables and interfaces to reactors. It is noteworthy that this work only considers interfaces that enable GET and POST operations. The parameters and the decision variables are defined as follows.

Parameters

$access_i$ access frequency for interface i;
$coup_{i,j}$ coupling degree between tables i and j;
Q the maximum access frequency load a reactor can sustain.

$$post_{i,j} = \begin{cases} 1, & \text{if POST for table } j \text{ is fulfilled through interface } i \\ 0, & \text{otherwise} \end{cases} \quad \forall i \in I,\ \forall j \in T$$

$$get_{i,j} = \begin{cases} 1, & \text{if GET for table } j \text{ is fulfilled through interface } i \\ 0, & \text{otherwise} \end{cases} \quad \forall i \in I,\ \forall j \in T$$

Decision variables

$tb_{k,i}$ equals 1 if table *i* is allocated to reactor type *k*, and 0 otherwise;
$int_{k,i}$ equals 1 if interface *i* is allocated to reactor type *k*, and 0 otherwise.

Based on the parameters and decision variables, we introduce the model designed to optimize the definition of reactors.

$$\max \quad \sum_{k=1}^{n} \omega_k \tag{1}$$

The objective function (1) maximizes the coupling level among tables allocated within each cluster. In order words, we aim to keep associated tables together as maximum as possible in order to reduce communication costs in case of JOIN operations.

$$\sum_{k=1}^{n} tb_{k,i} = 1 \qquad \forall i \in T \tag{2}$$

$$\sum_{k=1}^{n} int_{k,i} = 1 \qquad \forall i \in I \tag{3}$$

Constraints (2) and (3) limit the maximum number of clusters each table and interface are allowed to be allocated to, respectively. We aim to allocate a given table (or interface) to only one cluster.

$$int_{k,i} - tb_{k,j} = 0 \qquad where \quad post_{i,j} = 1 \quad \forall k = 1, ..., n \tag{4}$$

$$int_{k,i} - tb_{k,j} = 0 \qquad where \quad get_{i,j} = 1 \quad \forall k = 1, ..., n \tag{5}$$

$$\sum_{k=1}^{n} \sum_{i \in I} \sum_{j \in T} int_{k,i} + tb_{k,j} = 2 \qquad where \quad post_{i,j} = 1 \tag{6}$$

$$\sum_{k=1}^{n} \sum_{i \in I} \sum_{j \in T} int_{k,i} + tb_{k,j} = 2 \qquad where \quad get_{i,j} = 1 \tag{7}$$

Constraints (4)-(7) force the table that represents a resource to be allocated in the same cluster that its respective GET and POST operations are allocated to (through respective interfaces). In addition, constraints (4)-(7) force GET and POST interfaces to be allocated in the same cluster.

$$\alpha_k = \sum_{i \in T} tb_{k,i} \qquad \forall k = 1, ..., n \tag{8}$$

$$\omega_k = \sum_{i \in T} \sum_{j \in T/i} tb_{k,i}.tb_{k,j}.coup_{i,j} \qquad \forall k = 1, ..., n \tag{9}$$

$$\alpha_k \leq \omega_k + 1 \qquad \forall k = 1, ..., n \tag{10}$$

$$\sum_{i \in I} int_{k,i}.access_i \leq Q \qquad \forall k = 1, ..., n \tag{11}$$

$$\sum_{i \in I} int_{k,i} \geq \sum_{j \in T} tb_{k,j} \qquad \forall k = 1, ..., n \tag{12}$$

Constraints (8) represent the total number of tables a given cluster holds. Constraints (9) represent the level of coupling a given cluster holds. Constraints (10) force tables in a cluster to have a positive coupling level. In other words, if there is more than a table in the cluster, the tables must be associated through FK. Constraints (11) limit the maximum access frequency a cluster can sustain. Constraints (12) restrict the existence of a cluster with tables and no interfaces.

$$int_{k,i} \in \{0, 1\} \tag{13}$$

$$tb_{k,i} \in \{0, 1\} \tag{14}$$

Constraints (13-14) are integrality constraints.

Reactor function extraction. In order to identify application logic that would be more efficiently executed by the DBMS, Cheung et al. [Cheung et al. 2012] assert that "programmers must identify sections of code that make multiple (or large) database accesses and can be parameterized by relatively small amounts of input". Based on this observation, this step aims at identifying source code lines with: (*i*) high degree of data access and manipulation, and (*ii*) low complexity. Thus, we seek the identification of application logic in the business layer to be migrated to a reactor function (*RAF*). Equation 15 exhibits the formula for detecting methods from source code for migration.

$$RAF(M) = \begin{cases} 1, \text{ CYCLE}(M) < HIGH_1 \wedge \text{NOAV}(M) < MANY \wedge \\ \qquad \text{DDLOC}(M) \geq HIGH_2 \\ 0, \text{otherwise} \end{cases} \tag{15}$$

In the equation above, *M* is the business function being inspected; CYCLE(*M*) is the cyclomatic complexity of *M*; NOAV(*M*) is the Number of Accessed Variables of *M*; DDLOC(*M*) is the degree of Data-Driven Lines of Code (DDLOC) in *M*, which are those lines of code that make access to and manipulation of data.

As mentioned by Cheung et al. [Cheung et al. 2012], "identifying sections of application logic that are good candidates for conversion [...] is tricky". Thus, we suggest that the thresholds $HIGH_1$, $HIGH_2$, and $MANY$ must be adapted for each application,

taking into consideration characteristics such as average lines of code (LOC) of methods present in each module of the business layer.

Reactor function allocation. Once reactor functions are identified, based on the locality of their source methods in the dependency graph, it is possible to assign a function that operates on a given table to the cluster holding it. For example, if table tb_x is assigned to reactor type R_k and method M_y manipulates tb_x, then M_y is allocated to R_k. In case of a method manipulating multiple tables, then the method is assigned to the reactor type with the respective highest *access frequency* for the given interface.

5. Evaluation

Our technique is applied to the monolithic version of Petclinic,[2] an OLTP open-source demonstration project of the Spring Framework.[3] The system adopts a three-tier layered architecture and has been under development since 2016.

System dependency graph construction. Due to space constraints, we provide a partial representation of the dependency graph for Petclinic. Figure 3 depicts the execution path for the */visits/new* resource regarding a POST operation. A node is represented by a rectangle, in which the header is the class name followed by method name. The complete dependency graph for Petclinic can be accessed online.[4]

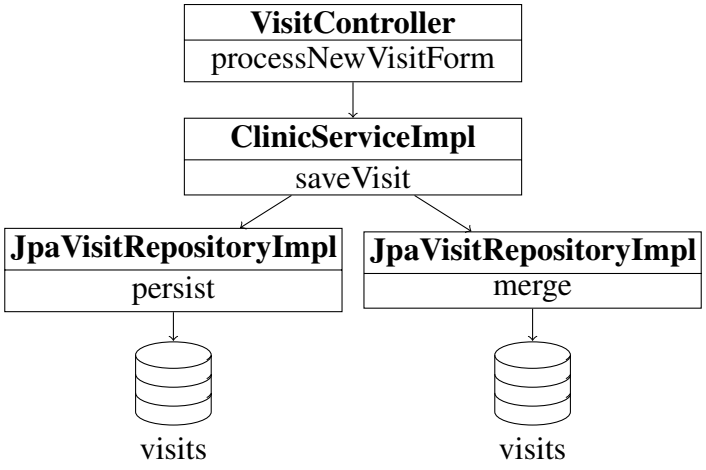

Figure 3. Dependency graph of */visits/new* interface POST operation in Petclinic

Profile data collection phase. Since Petclinic is a demonstration project, this study relies on an artificial workload that aims at reproducing a real-world scenario for the Petclinic domain. The workload decisions were taken based on characteristics of Petclinic, e.g., the insertion rate into the *pets* table cannot be lower than that of owners (a *pet* cannot exist without an *owner*), and table *visits* must incur the highest access frequency. Note that access frequency was normalized to range from 1 to 100 per interface.

Table coupling identification. The entity-relationship (ER) diagram for the Petclinic application is depicted in Figure 4. As can be seen, the relationships with coupling equal to 1 are: types and pets, owners and pets, and visits and pets.

[2]https://github.com/spring-petclinic/spring-framework-petclinic
[3]https://spring.io
[4]https://zenodo.org/record/3237968

Operation	Interface	Table	Access frequency
GET	/owners/ownerId	owners	10
GET	/owners/ownerId/edit	owners	10
PUT	/owners/ownerId/edit	owners	10
GET	/owners	owners	60
POST	/owners/new	owners	20
GET	/owners/ownerId/pets/new	pets	25
POST	/owners/ownerId/pets/new	pets	25
GET	/owners/ownerId/pets/petId/edit	pets	10
PUT	/owners/ownerId/pets/petId/edit	pets	10
GET	/vets	vets	10
GET	/owners/ownerId/pets/petId/visits/new	visits	100
POST	/owners/ownerId/pets/petId/visits/new	visits	100

Table 1. Workload scenario

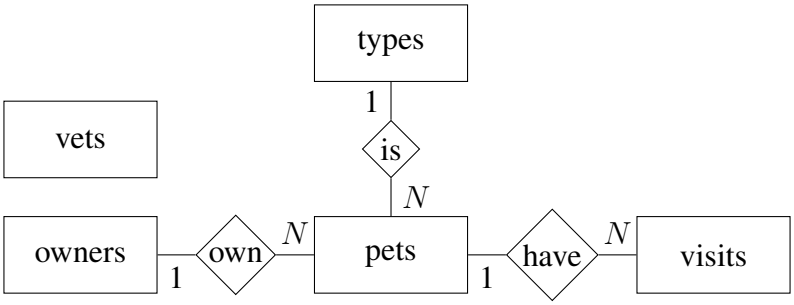

Figure 4. Petclinic ER diagram

Reactor table identification. In order to execute the model, the parameter Q was set to 200, corresponding to the sum for the resource with the highest access frequencies (*visits*). We aim to distribute workload among reactors, avoiding two or more data-intensive entry points to be allocated to the same reactor type. Figure 5 exhibits the result of the optimization allocating tables to clusters. The result of the allocation of interfaces to clusters can be accessed online.[4]

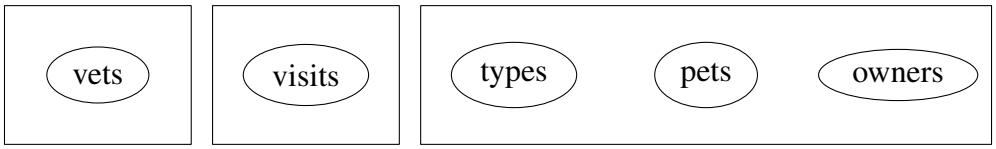

Figure 5. Optimization problem output

Reactor function extraction. For this step, a source code inspection was performed in Petclinic. Based on the strategy discussed in Section 4, we have employed a heuristic to define the thresholds: As the average Petclinic business layer LOC is 1, for each 3 lines of code, $HIGH_1$ is incremented. Also, as the average LOC of the repository layer in Petclinic is 2, DDLOC must be greater than or equal 1. An excerpt of an extracted reactor function based on the resource */visits/new* is shown in Figure 6. The remaining functions are available online.[4]

```java
@Service
public class ClinicServiceImpl implements ClinicService {
   // code ommitted for brevity
   private VisitRepository visitRepository;
   // code ommitted for brevity
   @Override
   @Transactional
   public void saveVisit(Visit visit) throws
         DataAccessException {
      visitRepository.save(visit);
   }
}
```

```java
@Repository
public class JpaVisitRepositoryImpl implements
      VisitRepository {
   // code ommitted for brevity
   @Override
   public void save(Visit visit) {
      if (visit.getId() == null) {
         this.em.persist(visit);
      } else {
         this.em.merge(visit);
      }
   }
}
```

```
void upsert_visit(visit){
   if visit.id IS NULL then
      INSERT INTO
               visits
      VALUES   (visit.date,
               visit.description,
               visit.pet_id);
      return;
   end if;

   SELECT id
   FROM visits
   INTO v_id
   WHERE visit.id = id;

   if v_id IS NULL then
      abort;
   end if;

   UPDATE visits
   SET date = visit.date,
      description = visit.description,
      pet_id = visit.pet_id
   WHERE visit.id = id;
}
```

Figure 6. Application logic (left) extracted to a reactor function (right)

Reactor function allocation. Following the technique, the method depicted in Figure 6, for example, is assigned to the cluster that holds the *visits* table. Information regarding the allocation of all identified methods can be accessed online.[4]

Based on the results, it is possible to correlate the distribution of relational actors and the respective tables and interfaces to the project Petclinic microservices version.[5] Table 2 depicts the tables presented in each microservice in the Petclinic microservices application. We observe that the decomposition of tables provided by the expert developers of Petclinic microservices is the same as the decomposition provided by our technique in terms of the tables and methods selected for each microservice.

Microservice	Tables
customers-service	owners, types, and pets
vets-service	vets
visits-service	visits

Table 2. Tables presented in each microservice for Petclinic

6. Limitations

One of the limitations of our technique concerns the assumption that the system adopts a three-tiered layered REST-based architecture. However, this architecture is widely adopted in industrial settings. Also, while our technique currently only considers interfaces that enable GET and POST operations, we do not anticipate significant issues in extending it with DELETE and PUT operations.

Regarding the evaluation, the prepared artificial workload may not provide sufficient coverage for all cases in which the technique could be applied. Nevertheless, the workload was defined based on reasoning about the application domain. It is noteworthy to mention that the workload, its limits, and the source code metrics were verified by three independent researchers. Additionally, to test the sensitivity of our model, we have also

[5]https://github.com/spring-petclinic/spring-petclinic-microservices

applied it to a different hypothetical workload, allowing us to observe sensitivity of model output due to changes in the input specifications.

Finally, we chose a specific Java software project for applying our technique. However, we believe the technique is generic enough to be applied to other object-oriented programming languages (e.g., C#) and frameworks (e.g., .NET Core).

7. Concluding Remarks

This study proposes an exact optimal approach for allocating relational tables, REST interfaces, and application logic to clusters, represented by reactors. To the best of our knowledge, there is no identical work in literature. Even though our system formalization is based on the work of Levcovitz et al. [Levcovitz et al. 2016], it goes much beyond in both presenting a MIP-solver-based automatic method for distribution among clusters as well as considering heuristics to identify application logic in source code to be extracted.

For evaluation purposes, the technique was applied to a REST-based application and exhibited appropriate results. Indeed, the output yielded by our technique consisted in a closely related reactor decomposition when compared to a microservices decomposition conducted by the developers of the original application example. As future work, we aim at conducting more in-depth empirical evaluations of our technique to further understand its benefits and limitations, including applying it to real-world systems.

References

Agha, G. (1986). *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, USA.

Bellatreche, L., Karlapalem, K., and Simonet, A. (2000). Algorithms and support for horizontal class partitioning in object-oriented databases. *Distributed and Parallel Databases*, 8(2):155–179.

Bernstein, P. A., Dashti, M., Kiefer, T., and Maier, D. (2017). Indexing in an actor-oriented database. In *CIDR 2017, 8th Biennial Conference on Innovative Data Systems Research, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*.

Boner, J., Farley, D., Kuhn, R., and Thompson, M. (2019). The Reactive Manifesto. `https://www.reactivemanifesto.org/`.

Cheung, A., Madden, S., Arden, O., and Myers, A. C. (2012). Automatic partitioning of database applications. *Proceedings of the VLDB Endowment*, 5(11).

Du, D.-Z. and Pardalos, P. (1998). *Handbook of Combinatorial Optimization. Combinatoral Optimization in Clustering*. Springer, New York, NY.

Fielding, R. (2000). *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine.

Fowler, M. (2015). Presentation domain data layering. `https://martinfowler.com/bliki/PresentationDomainDataLayering.html`.

Gysel, M., Kölbener, L., Giersche, W., and Zimmermann, O. (2016). Service cutter: A systematic approach to service decomposition. In *Service-Oriented and Cloud Computing*, pages 185–200. Springer International Publishing.

Hasselbring, W. and Steinacker, G. (2017). Microservice architectures for scalability, agility and reliability in e-commerce. In *IEEE ICSA Workshops*, pages 243–246.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J. (1997). Aspect-oriented programming. In *ECOOP'97 — Object-Oriented Programming*, pages 220–242, Berlin, Heidelberg. Springer Berlin Heidelberg.

Levcovitz, A., Terra, R., and Valente, M. T. (2016). Towards a technique for extracting microservices from monolithic enterprise systems. *CoRR*, abs/1605.03175.

Mazlami, G., Cito, J., and Leitner, P. (2017). Extraction of microservices from monolithic software architectures. In *International Conference on Web Services*. IEEE.

Olbrich, S. M., Cruzes, D. S., and Sjøberg, D. I. (2010). Are all code smells harmful? a study of god classes and brain classes in the evolution of three open source systems. *IEEE International Conference on Software Maintenance*.

Pavlo, A., Curino, C., and Zdonik, S. B. (2012). Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*, pages 61–72.

Richards, M. (2015). *Software Architecture Patterns*. O'Reilly, 1st edition.

Rowe, L. A. and Stonebraker, M. (1987). The POSTGRES data model. In *VLDB'87, Proceedings of 13th International Conference on Very Large Data Bases, September 1-4, 1987, Brighton, England*, pages 83–96.

Shah, V. and Salles, M. A. V. (2018). Reactors: A case for predictable, virtualized actor database systems. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD, Houston, TX, USA, June 10-15, 2018*, pages 259–274.

Shah, V. and Salles, M. V. (2017). Actor database systems: A manifesto. *CoRR*, abs/1707.06507.

Sutter, H. and Larus, J. R. (2005). Software and the concurrency revolution. *ACM Queue*, 3(7):54–62.

Wang, Y., dos Reis, J. C., Borggren, K. M., Salles, M. A. V., Medeiros, C. B., and Zhou, Y. (2019). Modeling and building iot data platforms with actor-oriented databases. In *Advances in Database Technology - 22nd International Conference on Extending Database Technology, EDBT 2019, Lisbon, Portugal, March 26-29, 2019*, pages 512–523.