2019 IEEE/ACM 11th International Workshop on Modelling in Software Engineering (MiSE)

Domain-specific Languages for the Design, Deployment and Manipulation of Heterogeneous Databases

Dimitrios S. Kolovos*, Fady Medhat*, Richard F. Paige†, Davide Di Ruscio‡,
Tijs van der Storm§, Sebastian Scholze¶ and Athanasios Zolotas*

*Department of Computer Science, University of York, York, United Kingdom,
Email: {dimitris.kolovos, fady.medhat, thanos.zolotas}@york.ac.uk
†Department of Computer Science, University of York & McMaster University,
York, United Kingdom & Hamilton, Canada, Email: richard.paige@york.ac.uk
‡Department of Information Engineering, Computer Science and Mathematics, University of L'Aquila,
L'Aquila, Italy, Email: davide.diruscio@univaq.it
§Centrum Wiskunde & Informatica, Amsterdam, Netherlands, Email: storm@cwi.nl
¶ATB – Institut für angewandte Systemtechnik Bremen, Bremen, Germany, Email: scholze@atb-bremen.de

***Abstract*—The need for levels of availability and scalability beyond those supported by relational databases has led to the emergence of a new generation of purpose-specific databases grouped under the term NoSQL. In general, NoSQL databases are designed with horizontal scalability as a primary concern and deliver increased availability and fault tolerance at a cost of temporary inconsistency and reduced durability of data. To balance the requirements for data consistency and availability, organisations increasingly migrate towards hybrid data persistence architectures comprising both relational and NoSQL databases. The consensus is that this trend will only become stronger in the future; critical data will continue to be stored in ACID (largely relational) databases while non-critical data will be progressively migrated to high-availability NoSQL databases.**

Designing and deploying a hybrid data persistence architecture that involves a combination of relational and NoSQL databases is a complex, technically challenging and error-prone task. In this paper we outline a model-based methodology developed in the context of the EC-funded H2020 TYPHON project for designing, developing, querying and evolving such scalable architectures for persistence, analytics and monitoring of large volumes of hybrid (relational, graph-based, document-based, natural language, etc.) data, in a systematic and disciplined manner.

***Index Terms*—hybrid persistence, relational databases, non-relational databases, domain-specific languages, model-driven engineering**

I. Introduction and Motivation

Up until recently, relational databases were considered as the de facto technology for persisting and managing large volumes of data. This changed with the emergence of enterprises such as Google, Twitter, Facebook, Amazon, etc. which were faced with extremely large datasets and unprecedentedly high availability requirements. The need for levels of availability beyond those supported by relational databases and the challenges involved in scaling such databases horizontally led to the emergence of a new generation of purpose-specific databases grouped under the term NoSQL [1]. NoSQL databases deviate from the long-established relational paradigm in order to address scenarios where very large datasets need to be managed under almost perfect availability. While NoSQL databases have been shown to be powerful enough to support the load of massive social networks such as Facebook and Twitter, high performance and availability typically come at the cost of durability and consistency.

Data managed within an organisation may have significantly variable consistency and availability requirements. For example, in the case of an e-commerce system, data used to provide recommendations of products that a user may be interested in needs to be highly available as they are constantly retrieved and updated as users browse through the system. As the consistency of such data is not critical, a small probability of loss of its integrity can be reasonably exchanged for a significant improvement in availability. By contrast, for other subsets of data in the same system, such as data recording customer orders and payments, compromising data consistency to improve availability is not acceptable.

As a result, organisations increasingly need to use both types of databases in parallel – with an unavoidable overlap between the data stored in these – using ad-hoc architectures. This introduces a number of challenges including ensuring the coherency of the overall design, the assembly and configuration of the different components of the architecture, and the consistency of the overlapping data. Designing and deploying a hybrid data persistence architecture that involves a combination of relational and NoSQL databases, and which can manage different types of structured and textual data (in the remainder of this paper we refer to such hybrid data stores as *polystores* for conciseness), is a complex, technically

This work is funded by the European Union Horizon 2020 research and innovation programme through the Polyglot and Hybrid Persistence Architectures for Big Data Analytics (TYPHON) project (#780251).

2575-4475/19/$31.00 ©2019 IEEE
DOI 10.1109/MiSE.2019.00021

challenging and error-prone task. Also, in order to access data stored in such architectures, developers need to write application code against different types of persistence backends. Unlike relational databases which provide support for SQL and standard APIs such as JDBC/ODBC, and are generally substitutable, each NoSQL database provides its own proprietary application programming interface (API) and query language. As such, exposing application developers directly to the API of a particular NoSQL database can result to high coupling between the application code and that database, which can hinder migration to a different database in the future. Ad-hoc development of such data persistence architectures also introduces data evolution and migration challenges and complicates the development of uniform monitoring and real-time analytics capabilities.

In this paper we outline a model-based methodology and integrated technical offering for designing, developing, querying, evolving, analysing and monitoring scalable hybrid data persistence architectures that will meet the growing scalability and heterogeneity requirements of organisations, which is currently being developed as part of the TYPHON EU-funded Horizon 2020 project. We focus on the model-driven aspects of the project and more specifically the three Domain-specific Languages (DSLs) proposed, namely TyphonML, TyphonDL and TyphonQL which help in designing, deploying and querying hybrid datastores, respectively. We will also present an architecture that uses these languages to facilitate the evolution of polystores and the extraction of various forms of analytics to provide to the reader the whole picture of the project, but we will not discuss details of the evolution and analytics components as these are outside of the scope of this paper.

The paper is structured as followed. Section II provides a high-level overview of the proposed methodology and architecture. Section III presents the three DSLs while Section IV concludes the work and outlines plans for future work.

II. Architecture Overview

Figure 1 shows an overview of the proposed architecture. The process starts with the creation of a model of the polystore. Developers, using a textual and graphical DSL called TyphonML, create models that include information regarding the concepts appearing in the polystore, their attributes and their relationships. These models, labelled as *TyphonML models* in Figure 1, also include information about the databases that are involved in the system. As a result, they represent the high-level infrastructure of a hybrid polystore.

Arguably, the abstraction gap between high-level TyphonML models and ready-to-use polystores is not negligible. To bridge this gap, an intermediate polystore deployment modelling language (TyphonDL) is used. TyphonDL provides concepts that lie at an abstraction level between that of TyphonML, and that of specific data stores and virtual machine configuration technologies. TyphonML models are transformed to *TyphonDL models* and are enhanced with more fine-grained database-specific options. TyphonDL models represent the deployment infrastructure of that polystore

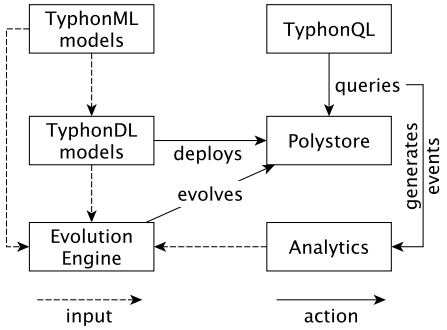

Fig. 1. An overview of the architecture of TYPHON.

in terms of the specific cloud platform and deployment tools employed and are used to generate the necessary installation and configuration facilities that, when executed, can assemble the polystore in an automated manner.

As data will be distributed across a number of heterogeneous databases a common data manipulation language is used. *TyphonQL* is developed for perfoming data manipulation commands (e.g., insert, delete, etc). Since TyphonQL queries are only executable on polystores precisely specified using TyphonML and TyphonDL, dedicated compilers/interpreters exploit this rich structural and semantical information to type-check and transform TyphonQL queries to high-performance native queries, and APIs that support advanced features such as prefetching and lazy loading to accommodate different usage scenarios. More details about the three DSLs are given in Section III.

The execution of TyphonQL queries will lead to the generation of events (also referred as *triggers* in the databases domain). These events are consumed by a high-performance framework for processing data access/update events to facilitate orthogonal real-time monitoring and predictive *analytics*.

Finally, information gathered from TyphonML, TyphonDL and the analytics components are used as input to the *evolution engine* which is responsible for evolving the organisation and distribution of data in hybrid polystores, as well as providing tools for monitoring the use of polystores to inform the evolution process.

III. Domain-specific Languages

A. TyphonML

TyphonML is a modelling language that supports the design of hybrid polystores. Using TyphonML, engineers are able to model the data that needs to be persisted in a homogeneous manner, abstracting over the specificities of the underlying technologies. Most NoSQL data stores are schemaless, i.e., there is no explicit schema specifying the internal structure of the data they manage [2]. Instead the schema is implicit. Support for schemaless data is particularly useful for systems that involve non-uniform data or whose structure changes often. On the other hand, the lack of an explicit schema can introduce challenges in data integration scenarios, where at least a partial understanding of the data structure is required. TyphonML supports the modelling of the implicit schema of schemaless

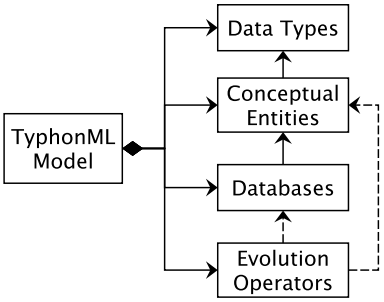

Fig. 2. High level architecture of TyphonML.

```
entity Order {
    date : Date
    totalAmount : double
    products -> Product."Product.orders" [*]
    paidWith -> CreditCard
    user -> User."User.orders"
}

...

relationaldb Orders {
    table {
        OrderTB:Order
        index OrderIndex { "Order.date" }
        id {"Order.date"}
    }

}
```

Fig. 3. An example of TyphonML syntax for an e-commerce scenario.

datasets and therefore it enables the integration of open and closed datasets (i.e. datasets, which conform to an open or strict schema respectively) in a seamless manner. Moreover, TyphonML provides facilities for capturing availability, consistency and partitioning requirements for different subsets of the modelled data, as well as the available infrastructure on top of which the hybrid polystore will be deployed.

A high-level overview of TyphonML is shown in Figure 2. TyphonML models include the conceptual entities and the different types of databases appearing in the polystore. To specify the attributes of the concepts in TyphonML, one can use primitive types, natural language processing (NLP) enabler types (which enable the application of natural language analysis) or define custom data types. Finally, modelers are provided with change operators that can be used to specify how already deployed TyphonML models have to be evolved and how the already stored data have to be consistently migrated.

B. TyphonDL

Models captured with TyphonML can be used to automate the process of assembling virtual machine (VM) images which contain configured installations of the required relational and NoSQL data stores on top of an operating system and other standard facilities, and which are ready for deployment on diverse cloud computing infrastructures. To bridge the abstraction gap between high-level TyphonML models and ready-to-use polystores an intermediate polystore deployment language (TyphonDL) is proposed. TyphonDL provides concepts lying at an abstraction level between that of TyphonML, and that of specific data stores and VM configuration technologies.

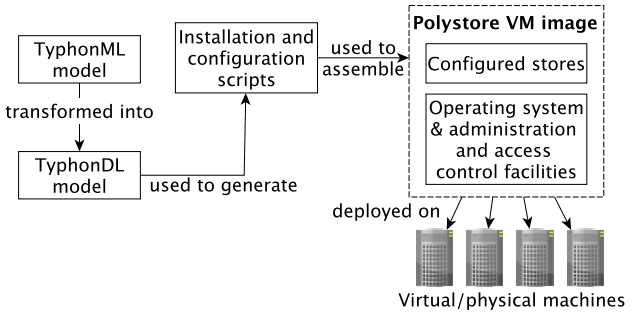

Fig. 4. Overview of the automated polystore assembly process.

The overall process is illustrated in Figure 4. A TyphonDL model takes two sources of input; The TyphonML model which includes database specific information (e.g., which are the database systems that are used to manage the modeled data entities and relationships) and deployment specific configuration parameters. There are different cofigurations parameters need to be set by developers. These include the cloud platform provider (e.g., Amazon Web Services (AWS), etc.), the container format (e.g., Docker, etc.), the specific database system for the type provided in the TyphonML model (e.g., MySQL, etc.), the storage space for each cluster, other platform dependent configuration options (e.g., amount of memory, etc.) and database specific variables (e.g., username/password for the administrator, etc.). The configuration parameters can be supplied by the developers by editing manually the TyphonDL model itself, via a textual editor that has been developed in Xtext [3] or a GUI editor.

Figure 5 shows an example for the definition of the configuration parameters using TyphonDL textual syntax. This TyphonDL model uses AWS as the platform provider, Docker as the containerization technology and a relational database. The AWS platform is then specified. It consists of a cluster named myAWSCluster, which consists of an application named myApplication. Inside the application a Docker container is modelled which consists of a relational database. Configuration parameters for the database and the container are also provided. A TyphonDL editor with graphical user interface (GUI) facilitates the definition of TyphonDL models by retrieving a list of needed uniquely named databases and their types (e.g., relational, etc.) from the TyphonML model. In a wizard, the specific DBMSs can be chosen from a list of supported systems for each required type. The technology for deployment (e.g., Docker) has to be selected from a list while other required configuration parameters can be provided from a configuration-file that is readable by the editor, a GUI with DBMS- and technology-specific text fields and/or by directly editing the generated TyphonDL model.

A model-to-text generator written in Xtend parses the created TyphonDL model and produces the appropriate installation and configuration scripts needed for each type of platform/container (e.g., a docker-compose.yml file).

```
platformtype AWS
containertype Docker
dbtype relationalDB

platform myAWSPlatform : AWS {
    cluster myAWSCluster {
        application myApplication {
            container mariadb : Docker {
                myDB : relationalDB
                image = gitlab.atb-bremen.de:5555/docker/...
                environment {
                    "MYSQL_ROOT_PASSWORD=12345"
                    "MYSQL_DATABASE=myDb"
                    "MYSQL_USER=admin"
                    "MYSQL_PASSWORD=123456"
                }
                volumes = /opt/admin/db:/var/lib
                networks = admin
            }
            ...
        }
    }
}
```

Fig. 5. An example of the TyphonDL syntax.

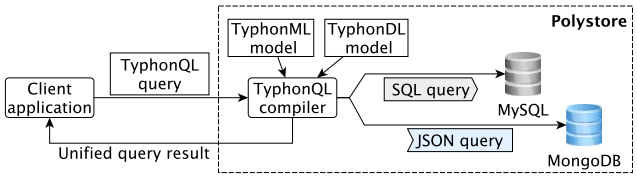

Fig. 6. Overview of the TyphonQL query execution process.

C. TyphonQL

TyphonQL is a language for querying heterogeneous data distributed across diverse databases. As a motivating example, consider the following scenario with reference to an e-commerce system, which uses a relational database to store orders and a document store to persist reviews and comments from customers. In this scenario, the merchant would like to hand out loyalty vouchers to customers who 1) have spent over $1000 in the e-shop and 2) have contributed high-quality reviews to the system (e.g. reviews that attracted at least 20 comments). Using TyphonQL, an engineer could write this query and the TyphonQL compiler/interpreter will then be responsible for breaking it down in two parts, executing these natively on the two different back-ends and returning the results in a uniform representation. An overview of the TyphonQL execution process in presented in Figure 6.

TyphonQL's concrete syntax is created using the syntax definition formalism of the Rascal language workbench [4] which automatically generates a parser from such declarative grammars. For reasons of brevity the grammar of TyphonQL is not presented in this paper - instead we present the basic Data Manipulation Language (DML) operators of the language.

The attributes of each conceptual entity are known to the TyphonQL compiler as they are provided in the TyphonML model. The querying mechanism breaks down the query (if needed), executes each part against the appropriate database (using Java APIs for each type of database) and creates Java objects for the results, populating them with the retrieved data. Let's assume that an "Order" entity has a reference to another entity, "User", representing the user who placed the order (as this is defined in the TyphonML model in the example of Figure 3). The date of the order and the name of the user could be retrieved using the query presented in Listing 1. As we are interested in retrieving these two attributes only, two *partially* filled Java objects will be created (one for each entity).

Listing 1. An example of a "Select" query in TyphonQL.
```
from Order o, User u
select o.date, u.name
where o.user == u
```

However, when arbitrary expressions (or aggregated results) are queried for, the result will contain anonymous entities. For instance, Listing 2 counts the number of orders placed in a specific date. In this case, we use the "as" keyword to name the expression count(o) as it does not correspond to a declared attribute of any entity. The result is stored in an anonymous entity which has one attribute named "numOfOrders".

Listing 2. An example of a "Count" query in TyphonQL.
```
from Order o
count(o) as numOfOrders
where o.date == "28-01-2019"
```

Finally, Listing 3 presents examples of the remaining DML commands supported by TyphonQL.

Listing 3. Examples of an "Insert", "Update" and "Delete" command in TyphonQL.
```
insert Order {date: "29-01-2019", totalAmount: 150.0,
products:{p1, p2} paidWith: CreditCard1, user: buyer1}

update from Order o select o where o.user == buyer1
set totalAmount = 100.0

delete from Order o select o
where o.name == "Nick Black"
```

IV. Conclusions and Future Work

In this paper we propose a methodology for designing, deploying, querying, evolving and analysing hybrid persistence architectures that fulfill growing scalability and heterogeneity requirements of organisations. We described three new domain-specific languages, named TyphonML, TyphonDL and TyphonQL giving some insights on their underlying principles and the current status of implementation. In the future, the constructs and the tools of all the languages need to be finalised and refined based on feedback while working on real case scenarios from the industrial partners involved in the project. Among others we plan to work on the extensibility mechanism of TyphonQL and the execution of Data Definition Language commands, support for the definition of individual nodes and of standard configuration concepts (e.g. DB master-slave nodes, Elastic instances, etc.) using TyphonDL.

References

[1] C. Strauch, U.-L. S. Sites, and W. Kriha, "NoSQL databases," *Lecture Notes, Stuttgart Media University*, vol. 20, 2011.

[2] K. Kaur and R. Rani, "Modeling and querying data in NoSQL databases," in *2013 IEEE International Conference on Big Data*. IEEE, 2013, pp. 1–7.

[3] S. Efftinge and M. Völter, "oAW xText: A framework for textual DSLs," in *Workshop on Modeling Symposium at Eclipse Summit*, vol. 32, 2006, p. 118.

[4] P. Klint, T. v. d. Storm, and J. Vinju, "RASCAL: A domain specific language for source code analysis and manipulation," in *SCAM'09*. IEEE Computer Society, 2009, pp. 168–177.