Ephemeral Data Handling in Microservices

Saverio Giallorenzo[1], Fabrizio Montesi[1], Larisa Safina[1,2], Stefano Pio Zingaro[3]
[1]University of Southern Denmark [2]Innopolis University [3]Università di Bologna and INRIA
{ saverio, fmontesi, safina } @imada.sdu.dk, stefanopio.zingaro@unibo.it

***Abstract*—Ephemeral data handling, whereby processed data do not persist, is an emerging requirement of connected IT systems, due to storage constraints (IoT) or regulatory demands (eHealth, GDPR). We present ongoing work on TQuery, a query language for ephemeral data handling in microservices.**

I. Introduction

The notion of *data ephemerality* has rapidly gained importance over the last decade [1], [2], due to a steep increase in the number of scenarios that require processed data not to persist in connected IT systems, e.g., due to resource constraints—as in the Internet of Things (IoT)—or regulations—e.g., eHealth [3], GDPR.

Using a general-purpose language to program complex data-handling is time-consuming and error-prone. Thus, developers often prefer to program data handling using a query language, paired with an execution engine [4]. Commonly, the execution engine is an external database management system (DBMS). In that context, the emergence of structured data-formats like XML and JSON pushed for the widespread adoption of DBMSes based on the NoSQL paradigm and tree-shaped data-structures [5]. However, when considering ephemeral data handling, external DBMSes hinder performance, due to *resource bottlenecks* and *persistence-related overheads*. The bottlenecks derive from well-known resource constraints, e.g., database connection pools. Overheads are instead typical of the ephemeral case, where data must be first inserted in the database (consuming time and bandwidth), queried, and finally deleted to ensure ephemerality.

These observations pushed us to formalise in [6] a NoSQL-based, in-memory query language, called TQuery, aimed at minimising bottlenecks and eliminating overheads due to data insertions (there is no DB to populate) or deletions (the data disappears when the process handling it terminates). TQuery is inspired by MQuery [7], a sound variant of the MongoDB Aggregation Framework [8], the most popular NoSQL query language. The reason behind the formalisation is twofold: *i*) we abstract away implementation details and reason on the overall semantics of our model, to avoid counter-intuitive query behaviours of the Aggregation Framework, as pointed out in [7]; *ii*) we provide a general reference for implementors, not tied to a specific technology.

In this paper, we illustrate our implementation of TQuery as an open-source library (https://github.com/jolie/tquery) for the microservice-oriented programming language Jolie [9], [10]. We deem Jolie a suitable choice because: *i*) Jolie programs are natively microservices [11], i.e., state-of-the-art service systems; *ii*) the language is successfully used in contexts typical of ephemeral data-handling [12]; *iii*) Jolie comes with a runtime environment that automatically translates incoming/outgoing data (XML, JSON, etc.) into the native, tree-shaped data values of the language—Jolie values for variables are always trees—and thus it supports *variety by construction* (an important aspect of connected IT systems, where different service might be implemented with different technologies). In section II we illustrate the usage of our library with a comprehensive use case taken from eHealth. In section III we report preliminary benchmarks, we position TQuery wrt related work, and we draw future directions of research and development.

II. A Use Case from eHealth

We draw our use case from [13], where the authors delineate a detection algorithm for encephalopathy. Using TQuery, we follow the principle "data never leave the hospital", in compliance with the GDPR [14]. While the algorithm in [13] considers many clinical tests for encephalopathy, we focus on two early markers: fever in the last 72 hours and lethargy in the last 48 hours. The relevant data (body temperature and sleep quality) are collectible through smart devices. At line 1 of Listing 1 we give, in a JSON-like format, an example of the biometric data from a smartwatch; likewise, at line 2, we show a sleep log example [15]. Both structures are arrays, marked `[ ]`, containing tree-like elements, marked `{ }`. At line 1, for each `date` we have an array of detected temperatures (`t`) and heart rates (`hr`). At line 2, to each year (`y`) corresponds an array of monthly (`M`) measures, to a month (`m`) an array of daily (`D`) logs, and to a day (`d`) an array of logs (`L`), each representing a sleep session with its start (`s`), end (`e`), and quality (`q`).

On the data structures above, we define a Jolie microservice, reported at lines 4–14, which describes the handling of the data and the workflow of the diagnostic algorithm, using our implementation of TQuery. The example is detailed enough to let us illustrate all the operators in TQuery: `match` , `unwind`, `project`, `group`, and `lookup`. Note that, while in Listing 1 we hardcode some data (e.g., integers representing dates like `20181128`) for presentation purposes, we would normally use parametrised variables.

Listing 1. Snippets of biometric (line 1) and sleep logs (line 2) data structures and TQuery-encoded diagnostic algorithm (lines 4–14).

```
1  [{date:20181129,t:[37,..],hr:[64,..]},{date:20181130,t:[36,..],hr:[66,..]},..]
2  [{y:2018,M:[..,{m:11,D:[{d:29,L:[{s:"21:01",e:"22:12",q:"good"},..]},{d:30,L:[{s:..]},..]},..]},..]
3
4  getPatientPseudoID@HospitalIT( patientData )( pseudoID );
5  credentials
6    |> getMotionAndTemperature@SmartWatch |> match { date == 20181128 || date == 20181129 || date ==20181130}
7    |> project { t in temperatures, pseudoID in patient_id } |> temps;
8  detectFever@HospitalIT( temps )( detectedFever );
9  if( detectedFever )
10   credentials |> getSleepPatterns@SmartPhone |> unwind { M.D.L }
11     |> project{ y in year, M.m in month, M.D.d in day,  M.D.L.q in quality }
12     |> match { year == 2018 && month == 11 && ( day == 29 || day == 30 ) }
13     |> group { quality by day, month, year } |> project { quality, pseudoID in patient_id }
14     |> lookup {patient_id == temps.patient_id in temps } |> detectEncephalopathy@HospitalIT
```

In Listing 1, line 4 defines a request to an external service, provided by the **HospitalIT** infrastructure. The service offers functionality getPatientPseudoID which, given some identifying patientData (acquired earlier), provides a pseudonymised identifier in variable pseudoID.

At lines 5–7 and lines 10–14 we use the Jolie (prototypical) chaining operator |> to define a sequence of calls, either to external services, marked by the @ operator, or to the internal TQuery library. The |> operator takes the result of the execution of the expression at its left and passes it as the input of the expression on the right.

At lines 5–7 we use the TQuery operators match and project to extract the recorded temperatures of the patient in the last 3 days/72 hours. At line 5 we evaluate the content of variable credentials, which holds the certificates to let the Hospital IT services access the physiological sensors of a given patient. In the program, credentials is passed by the chaining operator at line 6 as the input of the external call to functionality getMotionAndTemperature. That service call returns the biometric data (Listing 1, line 1) from the **SmartWatch** of the patient. While the default syntax of a service call in Jolie is the one with the double pair of parenthesis (e.g., at line 4 of Listing 1), thanks to the chaining operator |> we can omit to specify the input of getMotionAndTemperature (passed by the |>) and its output (the biometric data exemplified at Listing 1, passed to the subsequent |>). At line 6 we use the TQuery operator match to filter all the entries of the biometric data, keeping only those collected in the last 72 hours/3 days (i.e., since 20181130). The result of the match is passed to the project operator at line 7, which removes all nodes but the temperatures, found under t and renamed **in** temperatures (this is required by the interface of functionality detectFever , explained below). The projection also includes in its result the pseudoID of the patient, **in** node patient_id. We finally store the prepared data in variable temps (it will be aggregated with the processed sleep logs, at line 14).

At line 8, we call the external functionality detectFever to analyse the temperatures and check if the patient had any fever, storing the result in variable detectedFever.

After the analysis on the temperatures, **if** detectedFever is true, we continue testing for lethargy. To do that, at line 10, we follow the same strategy described for lines 5–6 to pass the credentials to functionality getSleepPatterns, used to collect the sleep logs of the patient from her **SmartPhone**. Since the sleep logs are nested under years, months, and days, to filter the logs relative to the last 48 hours/2 days, we first flatten the structure through the unwind operator applied on nodes M.D.L (end of line 10). For each nested node, separated by the dot (.), the unwind generates a new data structure for each element in the array reached by that node. Concretely, the array returned by the unwind operator contains all the sleep logs in the shape:

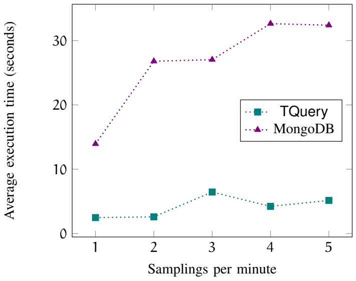
```
[{year:2018,M:[{m:11,D:[{d:29,L:[{s:"21:01",e:"22:12",q:"good"}]}]}]}
 {year:2018,M:[{m:11,D:[{d:29,L:[{s:"22:36",e:"22:58",q:"good"}]}]}]}]
```

where there are as many elements as there are sleep logs and the arrays under M, D, and L contain only one sleep log. Once flattened, at line 11 we modify the data-structure with the project operator to simplify the subsequent chained commands: we rename the node y **in** year, we move and rename the node M.m **in** month (bringing it at the same nesting level of year); similarly, we move M.D.d, renaming it day, and we move M.D.L.q (the log the quality of the sleep), renaming it quality — M.D.L.s and M.D.L.e, not included in the project, are discarded. On the obtained structure, we filter the sleep logs relative to the last 48 hours with the match operator at line 12. At line 13 we use the group operator to aggregate the quality of the sleep sessions recorded in the same day (i.e., grouping them **by** day, month, and year) and use the projection to keep only the aggregated values of quality (getting rid of day, month, and year); we also include under node patient_id the pseudoID of the patient. That value is used at line 14 to join, with the lookup operator, the obtained sleep logs with the previous values of temperatures (temps). The resulting, merged data-structure is finally passed to the **HospitalIT** services by calling the functionality detectEncephalopathy.

III. Benchmarks and Discussion

Benchmarks. As a preliminary result, we benchmarked the query at lines 6–7 of Listing 1 against a comparable architecture based on MongoDB. We programmed two microservices: QS contains the implementation at lines 6–7 of Listing 1; MS implements the same logic in terms of MongoDB queries: *i*) we insert the data in the database, *ii*) we send the query (match and project) as one instruction to the database, and *iii*) we delete the inserted data to ensure ephemerality. To run our tests, we use 5 instances of the data structure at line 1 of Listing 1. All instances cover 365 days of recordings but at increasing sampling rates, i.e., 1 per minute (1440 samplings per day), and then 2, 3, 4, and 5. We simulate bursts of requests in 4 subsequent batches, each with 10 concurrent requests (40 requests in total). A third microservice loads the data and sends 10 separate requests to QS (resp. MS) at a time. We draw our benchmarks in the figure below, reporting the average time over the 40 requests, for each sampling. In QS we start the timer before executing the first query instruction (match) and we stop it after we obtain the result of the last (project). In MS we start the timer before executing the insertion in the database and stop it after we queried and deleted the data. We run our benchmarks on a machine equipped with a 2.6GHz quad-core Intel Core i7 processor and 16GB RAM, running macOS 10.14, Java 11, Jolie 1.8-beta, and MongoDB 4.

In all cases, TQuery is faster than its MongoDB alternative. Intuitively, this is justified by fewer data transmissions, *ii*) no disk writings to ensure data persistence, and *iii*) no overhead due to database connections.



Discussion. We present ongoing work on and illustrate the usage of our implementation of TQuery as a library for the Jolie Service-Oriented language. Thanks to Jolie, our implementation of TQuery enjoys variety-by-construction, i.e., providing a consistent interface to query any data-format supported by the Jolie runtime (JSON, XML, etc.).

Regarding related work, we do not compare with DBMS systems in general, since we rule out their usage in the context of ephemeral data-handling, with the exception of ArangoDB [16]; an in-memory DBMS that can support JSON-like data-structures. The main differences with TQuery are: *i*) there is still overhead due to moving the data in memory between the database and the host program (assuming the in-memory database instance vanishes with the program running it) and *ii*) since ArangoDB supports multiple data-models, it comes with a query language that is not specifically suited for tree-shaped data-structures. Another solution close to ours is LINQ [17]. Similarly to TQuery, LINQ is a integrated, in-memory query language but, similarly to ArangoDB, it provides SQL-like operators not specifically purposed for tree-shaped data-structures.

As future work, we plan to *i*) evaluate the performance of TQuery wrt different application contexts, *ii*) to benchmark our implementation against other alternatives (e.g., SQL databases, AragoDB, LINQ, etc.), and to *iii*) expand the set of operators of TQuery, to capture more complex queries.

Acknowledgements. This work was partially supported by the Independent Research Fund Denmark, grant no. DFF-7014-00041.

References

[1] O. Tene and J. Polonetsky, “Big data for all: Privacy and user control in the age of analytics”, *Nw. J. Tech. & Intell. Prop.*, 2012.

[2] E. Shein, “Ephemeral data”, *Comm. of the ACM*, 2013.

[3] H. Oh *et al.*, “What is ehealth (3): A systematic review of published definitions”, *JMIR*, 2005.

[4] J. Cheney *et al.*, “A practical theory of language-integrated query”, *ACM SIGPLAN Notices*, 2013.

[5] D. P. Mehta and S. Sahni, *Handbook of data structures and applications*. Chapman and Hall/CRC, 2004.

[6] S. Giallorenzo *et al.*, *Ephemeral data handling in microservices - technical report*, 2019. eprint: arXiv: 1904.11327.

[7] E. Botoeva *et al.*, “Expressivity and complexity of mongodb queries”, in *ICDT*, Schloss Dagstuhl - LZI, 2018.

[8] *MongoDB Aggregation Framework*, https://docs.mongodb.com/manual/aggregation/, 2018.

[9] F. Montesi *et al.*, “Service-oriented programming with jolie”, in *Web Services Foundations*, A. Bouguettaya *et al.*, Eds., Springer, 2014.

[10] *Jolie Website*, https://www.jolie-lang.org/, 2018.

[11] N. Dragoni *et al.*, “Microservices: Yesterday, today, and tomorrow”, in *PAUSE*, Springer, 2017.

[12] M. Gabbrielli *et al.*, “A language-based approach for interoperability of IoT platforms”, in *HICSS*, AIS Electronic Library (AISeL), 2018.

[13] F. Vigevano and P. D. Liso, “Chapter 11 - differential diagnosis”, in *Acute Encephalopathy and Encephalitis in Infancy and Its Related Disorders*, H. Yamanouchi *et al.*, Eds., Elsevier, 2018.

[14] N. Rose, “The human brain project: Social and ethical challenges”, *Neuron*, 2014.

[15] S. M. Thurman *et al.*, “Individual differences in compliance and agreement for sleep logs and wrist actigraphy: A longitudinal study of naturalistic sleep in healthy adults”, *PLOS ONE*, Jan. 2018.

[16] *ArangoDB*, https://www.arangodb.com, 2014.

[17] E. Meijer *et al.*, “Linq: Reconciling object, relations and xml in the. net framework”, in *SIGMOD*, ACM, 2006.