Ephemeral data handling in microservices with Tquery

Saverio Giallorenzo, Fabrizio Montesi, Larisa Safina, Stefano Pio Zingaro

PeerJ Computer Science

Ephemeral data handling in microservices with Tquery

Saverio Giallorenzo[1,2], Fabrizio Montesi[3], Larisa Safina[4] and Stefano Pio Zingaro[1,2]

[1] Università di Bologna, Bologna, Italy
[2] INRIA, Sophia Antipolis, France
[3] University of Southern Denmark, Odense, Denmark
[4] INRIA, Lille, France

ABSTRACT

The adoption of edge and fog systems, along with the introduction of privacy-preserving regulations, compel the usage of tools for expressing complex data queries in an ephemeral way. That is, queried data should not persist. Database engines partially address this need, as they provide domain-specific languages for querying data. Unfortunately, using a database in an ephemeral setting has inessential issues related to throughput bottlenecks, scalability, dependency management, and security (*e.g.*, query injection). Moreover, databases can impose specific data structures and data formats, which can hinder the development of microservice architectures that integrate heterogeneous systems and handle semi-structured data. In this article, we present Jolie/Tquery, the first query framework designed for ephemeral data handling in microservices. Jolie/Tquery joins the benefits of a technology-agnostic, microservice-oriented programming language, Jolie, and of one of the most widely-used query languages for semi-structured data in microservices, the MongoDB aggregation framework. To make Jolie/Tquery reliable for the users, we follow a cleanroom software engineering process. First, we define Tquery, a theory for querying semi-structured data compatible with Jolie and inspired by a consistent variant of the key operators of the MongoDB aggregation framework. Then, we describe how we implemented Jolie/Tquery following Tquery and how the Jolie type system naturally captures the syntax of Tquery and helps to preserve its invariants. To both illustrate Tquery and Jolie/Tquery, we present the use case of a medical algorithm and build our way to a microservice that implements it using Jolie/Tquery. Finally, we report microbenchmarks that validate the expectation that, in the ephemeral case, using Jolie/Tquery outperforms using an external database (MongoDB, specifically).

Subjects Distributed and Parallel Computing, Theory and Formal Methods, Programming Languages, Internet of Things
Keywords Microservices, Jolie, Semi-structured data, Ephemeral data, Edge computing, Fog computing, Formal methods, Service-oriented computing, Query languages, e-Health

Submitted 3 May 2022
Accepted 22 June 2022
Published 22 July 2022

Corresponding author
Fabrizio Montesi,
fmontesi@imada.sdu.dk

Academic editor
Muhammad Aleem

Additional Information and Declarations can be found on page 35

DOI **10.7717/peerj-cs.1037**

INTRODUCTION

Background

Modern applications that make use of Edge Computing (*Shi et al., 2016*) and the Internet of Things (IoT for short) (*Baker, Xiang & Atkinson, 2017*) are increasingly developed as systems of microservices: independently executable components that communicate

via message passing (*Dragoni et al., 2017*). These systems typically have to deal with the continuous acquisition, processing, and distribution of semi-structured data. Over the last decade, the need for such data handling has contributed significantly to the adoption of document-oriented querying frameworks (*Leavitt, 2010*), like the MongoDB aggregation framework (*MongoDB Inc., 2022*)—and especially so in settings where Cloud Computing (*Armbrust et al., 2010*) is involved as well.

Recently, the necessity for careful data handling and the introduction of data protection regulations like the GDPR (*Van Alsenoy, 2019*) has highlighted the importance of handling ephemeral data (*Shein, 2013*). That is, in order to limit the circulation of data, applications should quickly process information without relying on persistency.

Ephemeral data handling is particularly relevant in scenarios where privacy is important (*Mostert et al., 2016*), for example eHealth (electronic systems that support healthcare) (*Baker, Xiang & Atkinson, 2017*), because it ensures by construction that data is automatically discarded unless the developers manually specifies otherwise. However, collecting and querying data with general-purpose languages in these contexts is often time consuming and error-prone (*Reda, Piccinini & Carbonaro, 2018*; *Ma, Wang & Chu, 2013*). In particular:

1. The implementation of query pipelines can quickly become complicated without proper abstractions.
2. Data might come from heterogeneous sources and in different data formats.

To solve the first issue (querying), developers typically include in their systems components that offer dedicated query languages (*Cheney, Lindley & Wadler, 2013*). For semi-structured data, a popular approach is to store data in a MongoDB instance (*MongoDB Inc., 2018b*), and then to use the MongoDB aggregation framework to perform queries.

As for the second issue (heterogeneity), developers can get support from programming languages or frameworks in which programs abstract from the concrete representation of data on the wire. Then, data is converted into the appropriate format and communicated through the appropriate protocol at runtime. Jolie is a (micro)service-oriented language designed to offer this capability (*Montesi, Guidi & Zavattaro, 2014*). A Jolie service can type, communicate, and manipulate semi-structured data under a unifying model that abstracts from data formats and communication protocols. Then, the program can be reused with different deployment instructions, which inform the Jolie engine of how data should be formatted (binary representations, JSON, XML, *etc.*) and communicated (using HTTP, SOAP, *etc.*) (*Montesi, 2016*). Jolie instructions can further be composed in workflows (*Gabbrielli, Giallorenzo & Montesi, 2014*); a feature that simplifies the programming of data collection and distribution in IoT and edge environments (*Gabbrielli et al., 2019*).

The Problem

Ideally, a tool for ephemeral data handling in microservices would give us the best of the MongoDB aggregation framework and of the Jolie programming language: a query framework designed for semi-structured data and a language for working in heterogeneous environments.

An obvious attempt at achieving what we want is to just "stick together" MongoDB with Jolie, in the sense of deploying a Jolie service in the company of a MongoDB instance. Unfortunately, this approach runs into issues:

Dependency An external Database Management System (DBMS) like MongoDB is an additional standalone component that needs to be installed, deployed, and maintained. As with any software dependency, this exposes the applications to challenges of version incompatibility (*Jang, 2006*).

Security The companion DBMS is subject to weak security configurations (*Brian Krebs, 2017*) and query injections, increasing the attack surface of the application. This is a typical problem in microservices-with-database deployments where usually the microservice composes queries by assembling external inputs as strings, which is the main vector for query injections (*Ron, Shulman-Peleg & Puzanov, 2016*).

Inconsistency The key features of the MongoDB aggregation framework have only recently been formally understood, and some present idiosyncrasies related to implementation that do not make sense for a clean, abstract model (*Botoeva et al., 2018*).

Performance The communication channel between the MongoDB instance and the Jolie service can become a bottleneck, introducing the usual performance issues of database connections (*Visveswaran, 2000*). This is common in microservices-with-database scenarios where the overheads of establishing database connections can limit the performance of the whole component (and techniques, like managing pools of persistent database connections, are partial solutions (*Visveswaran, 2000*) that make the logic of the microservices more involved). Data format conversions in these communications contribute to overhead as well, together with the necessary measures to ensure ephemerality (post-query data deletion).

Our solution

We propose the integration of relevant MongoDB data-query operators in Jolie. Our solution avoids the issues above: **Dependency**, since there is not anymore a database that we need to install and maintain; **Security**, because shedding the database removes risks from weak security configurations and, since the queries are part of the language (and not simply strings that we forward to the database engine), we also lower the exposition to query injections; **Inconsistency**, by building upon previous work on the formalisation of a consistent data-query theory of MongoDB (*Botoeva et al., 2016*); **Performance**, since there is no database involved, we avoid the overhead of: passing the data to and from the database; possible data-format conversions; bottlenecks due to pools of database connection channels (and possible bugs linked to their management), and of ensuring ephemerality.

This article

We present two main contributions. The first one is a formal model of a query language for semi-structured data, called Tquery. The second is an implementation of Tquery, called

Jolie/Tquery, which is the first query framework designed for ephemeral data handling in microservices. Jolie/Tquery addresses the problem by joining the benefits of Jolie and of the MongoDB aggregation framework: data can be collected from heterogeneous sources and then be queried in local memory by using pipelines of operations on semi-structured data.

The development of Jolie/Tquery is inspired by cleanroom software engineering. In particular, we have implemented our framework from scratch, starting from a formal model of its operators and their semantics. Our main contributions are described in the following.

Formal Specification We define Tquery, a theory for querying semi-structured data compatible with Jolie. Tquery provides the key operators of the MongoDB aggregation framework (match, unwind, project, group, and lookup), but reformulated for Jolie data structures and their accompanying syntax of paths for data traversal.

Implementation We develop Jolie/Tquery, an implementation of Tquery in the form of a Jolie package that can be used in services. Jolie/Tquery is lightweight: the entire compiled package consists of less than 100 kb. The implementation consists of two parts: an Application Programming Interface (API) to construct and run query pipelines, which defines the syntax of Tquery operators in terms of Jolie types; and an implementation of the API that follows the semantics given in Tquery. Jolie comes with an engine that supports implementing Jolie APIs with different languages (*Montesi, 2016*). In our case, Jolie/Tquery is implemented in Java. Jolie applications can use Jolie/Tquery by passing data in local memory and using native Jolie structures, which avoids the aforementioned issues. At the same time, Jolie applications can use Jolie's capabilities for integrating with heterogeneous components to collect and distribute data.

Evaluation We illustrate the expressivity of Jolie/Tquery by using it to implement a use case from eHealth: a detection system for encephalopathy based on a proposal by *Vigevano & Liso (2018)*. We then carry out microbenchmarks to validate the expectation that using Jolie/Tquery, being an in-memory query framework, outperforms using an external database management system (MongoDB specifically).

The article is structured as follows. 'Related Work' covers the related work. 'Overview and Running Example' illustrates the Tquery with the running example from the eHealth. 'The Tquery Formalisation' introduces formalisation of Tquery. 'Implementation' presents implementation of Tquery as a microservice written in Jolie programming language. 'Benchmarks' provides the benchmarks comparing the Tquery with the MongoDB. 'Discussion and Conclusion' drives conclusions.

This is the journal version of *Giallorenzo et al. (2019)*, a short conference article where we presented preliminary ongoing work about the implementation of Jolie/Tquery.

RELATED WORK

Jolie/Tquery is the first implementation from scratch of a formally-specified, document-oriented query framework. Our formal model, Tquery, stands on the shoulders of MQuery (*Botoeva et al., 2018*), the first formal model of query operators for JSON data

structures. MQuery formalises the key operators of the MongoDB aggregation framework, dispensing from some unnecessary idiosyncrasies that can lead to counterintuitive behaviour. Tquery inherits this good feature—the reader interested in the technical differences w.r.t. the MongoDB aggregation framework can consult (*Botoeva et al., 2016*, Appendix C). The key difference between this work and *Botoeva et al. (2018)* is that Tquery comes with an implementation, whereas *Botoeva et al. (2018)* investigated the theoretical expressivity of the MQuery operators w.r.t. relational algebra and their complexity. Tquery adopts the same operators but reformulated to be compatible with the Jolie data model (by adopting arrays instead of unordered forests for document collections). The semantics of our operators is also specified differently: while Tquery's operators follow the same intuition of the operators in MQuery, we give our semantics specifying how operations can be computed. For example, we do not rely on existential quantification and all our definitions are given by recursion on the structures of inputs. We believe that formalisation efforts like MQuery and Tquery are important: during the development of our implementation, we found having a reference formal model helpful to clarify the expected behaviour of operators and what tests we should write.

Jolie has been used in several domains that require ephemeral data handling, including smart mobility (*Callegati et al., 2017*), IoT (*Gabbrielli et al., 2018*), integration components in document management systems (*Maschio, 2019*), and media content (*Maschio, 2017*). However, due to the lack of an appropriate query framework, the query logic has been implemented manually with a general-purpose computation language (the computation layer of Jolie). Because it guarantees that data gets discarded (ephemerality) and it provides an expressive set of compositional query operators, Jolie/Tquery offers a better alternative for writing data-intensive Jolie microservices. Moreover, since every Jolie program is a composition of services, adapting a program to offload parts of its computations to remote nodes is simple (it mainly regards the reconfiguration of how services are deployed). This, in unison with the fact that Jolie/Tquery operators are stateless, simplifies the task of splitting Jolie/Tquery heavy-weight or computation-intensive queries over multiple nodes.

Other solutions that offer semi-structured data querying in separate services include MongoDB (*MongoDB Inc., 2018b*) and CouchDB (*Apache, 2005*); however, these are DB-based solutions that fall into the category of deployments that we deem unfit for the case of ephemeral data-handling. Moreover, these do not come with a formal model which, *e.g.*, one can use to reason about the semantics of the implementation and to check its consistency (like *Botoeva et al. (2016)* demonstrated for MongoDB).

There exist works on the integration of relational query frameworks with general-purpose programming languages, including: object-relation mapping frameworks (ORMs), which map objects to database entities (*Fussel, 1997*); Opaleye, a Haskell DSL for generating PostgreSQL commands (*Ellis, 2014*); and LINQ (*Meijer, Beckman & Bierman, 2006*), which provides query operators targeting SQL tables and XML structures for .NET languages. Tquery could be a reference to implement similar frameworks for semi-structured data in these languages. A convenient feature of Jolie/Tquery is that all its queries can work with any data format that Jolie can handle: Jolie automatically converts data in different formats

(including JSON, XML, and some binary formats) to its abstract data model (*Montesi, Guidi & Zavattaro, 2014*; *Montesi, 2016*).

As we are going to exemplify in the next section, a typical use case for semi-structured data handling and Jolie/Tquery is the reactive processing of events. Stream-processing languages have been explored for similar tasks, but they feature different kinds of primitives and are usually not based on semi-structured data.

The landscape of stream-processing languages is quite wide, *e.g.*, data-centric (*Chen et al., 2000*; *Barbieri et al., 2009*), time- or hardware-constrained execution-centric (*Caspi et al., 1987*; *Hirzel, Schneider & Gedik, 2017*; *Tommasini et al., 2019*), focussed on the relational- or document-oriented (*Chen et al., 2000*; *Diao et al., 2002*; *Mendell et al., 2012*) approach. In particular, SQL-based stream-processing languages (*Esteves et al., 2017*; *Babu & Widom, 2001*) recently gained popularity in industry (thanks to the familiarity of programmers with the SQL language), with commercial tools such as Apache Flink (*Apache, 2022a*), Apache Kafka (KSQL) (*Narkhede, 2017*), Apache Samza (*Apache, 2022b*), Apache Storm (*Apache, 2022c*), WSO2 Stream Processor (*WSO2, 2022*), Siddhi (Siddhi Streaming SQL) (*Siddhi, 2022*). We deem StreamQL (*Kong & Mamouras, 2020*) the work closest to Jolie/Tquery. This is a query language for efficiently processing IoT data streams. The StreamQL Engine is implemented as a lightweight Java library and does not depend on the external engine. However, StreamQL is a functional language that is based on formal semantics residing on the class of monotone functions over streams. It works with the typical functional primitives on list-based data, supporting a variety of operators that simplify stream-processing at the level of data aggregation (filtering, windowing, etc.) and data-flow control (*e.g.*, parallel composition). StreamQL does not handle explicitly semi-structured document-oriented data and requires additional processing for data translation, while Jolie/Tquery handles it natively(tree-shaped data simplifies integration with Jolie). Unlike Jolie/Tquery, StreamQL has built-in primitives for temporal control typical for data streaming languages. In Jolie/Tquery time contracts can be implemented by adding information to the data structures and need to be managed explicitly by the programmer. Widening our scope, we deem two works, CQL (*Arasu, Babu & Widom, 2006*) and EQL (*Elasticsearch, 2022*), close to Jolie/Tquery. CQL is a declarative streaming SQL-based query language, implemented in the STREAM DSMS (*Arasu et al., 2016*) with data captured with sliding windows (*Babcock et al., 2002*) based on time-(*e.g.*, update the data every 30 s) and data-related conditions (*e.g.*, capture the data as soon as it arrives). EQL (*Elasticsearch, 2022*) is an event-based data manipulation library developed in Python. Similarly to CQL, EQL expects data to follow an event-oriented schema. Interestingly, EQL provides a query-composition operator similar to the one provided by Jolie/Tquery (see 'Extending Jolie/Tquery with query pipelines'). Both CQL and EQL, being SQL-based, work on tuples of data rather than semi-structured documents as Jolie/Tquery does—e.g., one needs to convert a JSON document into tuples of data before using CQL/EQL.

Finally, Ballerina (*Oram, 2019*) is a language for the development of microservices close to Jolie, developed by WSO2, that equips SQL-like query operators to process data and events. The differences with Jolie/Tquery include the relational nature of the operators,

which requires the user to translate values between document- and tuple-shaped data when applying/using the data from the queries, and the lack of a formal reference.

OVERVIEW AND RUNNING EXAMPLE

In this section, we illustrate our proposal with an eHealth use case, showing the definition of a diagnostic algorithm as a composition of Tquery's operators. We deem this area of application apt to illustrate Tquery for two main reasons.

First, since medical diagnostic algorithms are usually expressed through declarative or high-level imperative instructions, having high-level, declarative operators for data handling narrows the gap between definitions and implementations and helps in both translating and checking their correctness. Indeed, more and more studies emerged proposing non-intrusive, affordable yet accurate diagnostic systems based on data collected from heterogeneous sources such as user-inputted data, smartphones, wearables, and cameras (*Purohit et al., 2020*). An emblematic example of this phenomenon is the recent proposal by *Hirten et al. (2020)*, where the authors defined and demonstrated the efficacy of a diagnostic algorithm to identify and predict SARS-CoV-2 (aka COVID-19) infections, reporting promising predictive ability to identify infection days before the diagnosis through nasal-swab testing. Here, we focus on a simpler-yet-comprehensive diagnostic algorithm defined by *Vigevano & Liso (2018)* to detect cases of encephalopathy.

Second, the inherent ephemerality of Tquery programs caters to the principles of secrecy and obliviousness of data—the data handled by a Tquery program is automatically deleted from memory—in the healthcare sector. This approach is frequently summarised by the motto "the data never leave the hospital" and it is compliant with the current regulations on data protection (*e.g.*, GDPR (*Rose, 2014*)).

In the remainder of the article, we use the diagnostic algorithm by *Vigevano & Liso (2018)* to illustrate the formal semantics of Tquery. Here, we focus on the overall definition of the parts of the algorithm and how we can map them to a combination of Tquery operators acting on and merging data from different sources. Then, in 'The Tquery Formalisation', we return on the single instructions that make up the algorithm presented here and show the step-by-step output of Tquery operators, following from the specification of their semantics.

An encephalopathy diagnostic algorithm

Taking inspiration from *Vigevano & Liso (2018)*, we focus on the aggregation of two early markers to detect encephalopathy: fever in the last 72 h and lethargy in the last 48 h. Those data are collectable by commercially-available smart-watches and smart-phones (*Bunn et al., 2018*): body temperature and sleep quality.

Tquery defines operators over tree-like data structures, formally defined in 'Data structures: trees and paths'. To keep this example compact, it is sufficient that the reader has some familiarity with data formats like XML (*Bray et al., 2000*) and JSON (*Crockford, 2006*) documents. Specifically, here we use a subset of the JSON format where a tree is represented by a pair of brackets `{ }`, which enclose a set of ordered pairs, each linking a

label (unquoted) to an array, whose content is enclosed within square brackets `[ ]`. Arrays can either contain trees or primitive values (string, integer, *etc.*).

As an example of the format above, we report in Listing 1 code snippets exemplifying the shape of the two data structures used in the example: the first (Lines 2–5) carries the temperature and heart-rates, the second holds the sleep logs (Lines 7–16) (*Thurman et al., 2018*).

At Lines 2–5, for each `date` we have an array of detected temperatures (`t`) and heart-rates (`hr`). At Lines 7–16, to each year (`y`) it 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`).

```
1 // representation of the tmp data structure
2 [{date:[20201127], t:[37], hr:[64]},
3   {date:[20201128], t:[36], hr:[66]},
4   {date:[20201129], t:[36], hr:[65]},
5   {date:[20201130], t:[37], hr:[67]}]
6 // representation of the sl data structure
7 [{y:[2020],
8   M:[{m:[11],
9       D:[{d:[27], L:[{s:['23:33'], e:['07:04'], q:['poor']}]},
10         {d:[28], L:[{s:['21:13'], e:['09:34'], q:['good']}]},
11         {d:[29], L:[{s:['21:01'], e:['03:12'], q:['good']},
12                    {s:['03:36'], e:['09:58'], q:['good']}]},
13         {d:[30], L:[{s:['20:33'], e:['01:14'], q:['poor']},
14                    {s:['01:32'], e:['06:15'], q:['good']}]}
15      ]}]}]
16 }]
```

Listing 1: Snippets of biometric (Lines 2–5) and sleep logs (Lines 7–16) data.

To implement the algorithm for detecting encephalopathy, we need to integrate with two functionalities provided by the Hospital IT infrastructure: `detectFever` and `detectEncephalopathy`. The former accepts data of the shape:

```
[ { t:[ 36, 37, 38 ], patient_id:[ 'xyz' ] } ]
```

where `t` contains the array of measured temperatures in the last three days and `patient_id` the identifier for the patient in the Hospital IT infrastructure. The latter accepts the format:

```
[ { temperatures:[ 36, 37, 38 ], patient_id:[ 'xyz' ],
    quality:[ 'good', 'poor' ] } ]
```

where `temperatures` contains the measured temperatures in the last three days, `patient_id` contains the identifier for the patient in the Hospital IT infrastructure (essentially, this is the same data found, respectively, under `t` and `patient_id` issued to the `detectFever` functionality), and `quality` contains the recorded quality of sleep in the last two days.

Our focus in the last part of this section is to describe—by means of the Tquery operators—a program that manipulates the biometric and sleep logs data in Listing 1 to integrate the functionalities `detectFever` and `detectEncephalopathy` and implement the diagnostic algorithm.

An overview of the Tquery operators

Before presenting the diagnostic algorithm, we give a brief and informal description of the shape and effect of each Tquery operator (presented formally in 'The Tquery Formalisation'), as a reference to integrate the description of the example.

- the *match* operator μ, given an array and a match criterion returns the elements of the array that satisfy the criterion, in their relative order from the input;
- the *unwind* operator ω takes as inputs an array and a path p.[1] The result of the application is a new array containing the "unfolding" of the input array under the path, *i.e.*, where we take each element e from the input array, we find all values under p in e and, for each value, we include in the new array a copy of e except it holds only that single value under p;
- the *project* operator π, given an array and a projection expression, it returns a copy of the original array with each element updated by the projection expression. Projection expressions can move/rename and remove sub-parts from the elements, as well as insert new ones;
- the *group* operator γ takes as inputs an array and two lists of paths: a grouping list and an aggregation list. The result of the application is a new array where each element has two properties: *(i)* it includes the combinations of distinct values from the set of values found under the grouping paths among the elements in the input array; *(ii)* it aggregates all the values found under the aggregation paths among the elements in the input array which have been grouped by the same combination of values;
- the *lookup* operator λ joins two arrays, a "source" and an "adjunct" one, according to the correspondence of values in their elements with respect to a source path and an adjunct path. Besides those inputs, the operator requires a "destination" path. The application of the operator returns a new array that contains all the elements resulting from merging each element e_s in the source array with the elements e_a in the adjunct array such that e_s and e_a hold the same values under the respective source and adjunct paths. The resulting array contains all the elements from the source, each updated to include, under the provided destination path, all path-matching elements from the adjunct array.

[1] Intuitively, a path is a sequence of node labels of the shape A.B.C. Formally, cf. 'Data structures: trees and paths'.

Implementing the diagnostic algorithm with Jolie and Tquery

Here, on the data structures and operators described above, we define a Jolie microservice (reported in Listing 2), which implements the handling of the data and the workflow of the use-case diagnostic algorithm.

The example is broad enough to let us illustrate all the operators in Tquery and to represent a real-world workflow, where, besides implementing the algorithm of interest, we manipulate the data for system integration (*e.g.*, by reshaping the data structures to fit the service APIs we need to invoke). Note that, while in Listing 2 we hard-code some data (*e.g.*, integers representing dates like `20201128`) for presentation purposes, we would normally use parametrised variables.

Since we follow a formalisation-first approach to present Tquery, in Listing 2 we interleave runnable Jolie code with the formal definition of the application of the involved

Tquery operators. When doing so, we use the highlighted, algorithmic notation $\cdots \leftarrow \cdots$. After having defined the formal semantics of the operators in 'The Tquery Formalisation', we will present the actual implementation of the example in 'Implementation' using our implementation of the Tquery operators in Jolie.

Note also that, while variables of the form `patientData` and `tmp` in Listing 2 conveniently resemble variable symbols as found in Java or C, they are actually path applications on the state of a Jolie program, which is a tree. Hence, the meaning of `tmp` reads "get the structure pointed by path *tmp* in the current state of the program". In the example, when assigning and passing values, we use the notation `a` and `b.c` to indicate the path traversal and retrieval of the structure pointed by the respective paths *a* and *b.c* on the state of the Jolie program. We instead use the notation *a* and *b.c* to indicate the passing of paths as parameters of Tquery operators.

We now describe the diagnostic algorithm and how we use the Tquery operators to implement it.

In Listing 2, at Line 1 we find the Jolie code of a request to an external service, provided by the `HospitalIT` infrastructure. The service offers the functionality `getPatientPseudoID` which, given some identifying `patientData` (acquired earlier), provides a pseudo-anonymised identifier—needed to treat sensitive health data—saved in variable `pseudoID`.

At Line 2 we retrieve in the variable `credentials` the keys to access the physiological sensors of the patient to obtain the biometric data (Listing 1, Lines 1–5) from the `SmartWatch` of the patient, by invoking the functionality `getMotionAndTemperature` and storing the result in `tmp`.

At Lines 3–5 we use the Tquery operators μ, γ, and π to extract the recorded temperatures of the patient in the last 3 days/72 h. At Line 3 we use the match operator μ to filter all the entries of the biometric data, keeping only those of the last 72 hours/3 days. At Line 3, we aggregate the result of the μ (which replaced the previous value under `tmp`) under the path *t* and discard the others. At Line 5, we use the project operator to include in `tmp` the identifier of the patient, under *patient_id*.

At Line 6, we call the external functionality `detectFever` to analyse the temperatures and check if the patient manifested any fever, storing the result in `hasFever`.

```
1  getPatientPseudoID@HospitalIT( patientData )( pseudoID )
2  getMotionAndTemperature@SmartWatch( credentials )( tmp )
3  tmp ← μ( tmp, (date=20201128 ∨ date=20201129 ∨ date=20201130) )
4  tmp ← γ( tmp, (t), () )
5  tmp ← π( tmp, ( t, pseudoID ⟩ patient_id ) )
6  detectFever@HospitalIT( tmp )( hasFever )
7  if( hasFever ) {
8    getSleepPatterns@SmartPhone( credentials )( sl )
9    sl ← ω( sl, M.D.L )
10   sl ← π( sl, (y ⟩ year, M.m ⟩ month, M.D.d ⟩ day, M.D.L.q ⟩ quality) )
11   sl ← μ( sl, (year=2020 ∧ month=11 ∧ (day=29 ∨ day=30)) )
12   sl ← γ( sl, (quality), () )
13   sl ← π( sl, (quality, pseudoID ⟩ patient_id) )
```

```
14   bs←λ( sl, patient_id, tmp, patient_id, temperatures )
15   bs←π( bs, (quality, patient_id, temperatures.t⟩temperatures) )
16   detectEncephalopathy@HospitalIT( bs )
17 }
```

Listing 2: Snippet implementing the diagnostic algorithm.

After the analysis on the temperatures, `if` the patient `hasFever` (Line 7), we continue testing for lethargy. To do that, at Line 8 we follow the same strategy described for Line 2 to pass the `credentials` to the functionality `getSleepPatterns`, used to collect the sleep logs of the patient from her `SmartPhone` in `sl`.

Then, 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 ω operator applied on the path *M.D.L* (Line 9). For each nested node (separated by the dot in the path), ω generates a new data structure for each element in the array reached by that node. Concretely, the array returned by the ω operator at Line 9 contains each sleep log associated with the full date of the recording (year, month, and day), as shown below.

```
[{y:[2020],M:[{m:[11],D:[{d:[27],L:[{s:['23:33'],...}]}]}]},
 {y:[2020],M:[{m:[11],D:[{d:[28],L:[{s:['21:13'],...}]}]}]},
 {y:[2020],M:[{m:[11],D:[{d:[29],L:[{s:['21:01'],...}]}]}]},
 {y:[2020],M:[{m:[11],D:[{d:[29],L:[{s:['03:36'],...}]}]}]},
 ...]
```

Given the new shape of `sl`, at Line 10 we modify the data structure with the project operator π to simplify the subsequent commands: we rename the node *y* to *year*, we move and rename the node *M.m* to *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 of the quality of the sleep), renaming it *quality*—*M.D.L.s* and *M.D.L.e*, not included in the parameters of the projection, are discarded.

On the obtained structure, we filter the sleep logs relative to the last 48 h with the match operator at Line 11.

At Line 12 we use the grouping operator γ to aggregate the *quality* of the sleep sessions recorded in the same day and discarding the nodes *day*, *month*, and *year*.

At Line 13 we project within the `sl` data structure the `pseudoID` of the patient under node *patient_id*. That value is used at Line 14 to join, with the lookup operator λ, the obtained sleep logs with the previous values of temperatures (`tmp`). Lastly, we prepare the data structure to be submitted for analysis. We do this at Line 15 by keeping the paths *quality* and *patient_id* in `bs` and by moving the nested temperatures (*temperatures.t*) under the path *temperatures*—this is required by the interface of `detectEncephalopathy`, which we invoke passing the resulting (`bs`) data structure.

THE TQUERY FORMALISATION

In this section, we report the formalisation of Tquery. Besides providing a general, mathematical reference, the formalisation guides the implementation of our Jolie framework, presented in 'Implementation'. Tquery is inspired by MQuery (*Botoeva et*

al., 2018), a sound variant of the MongoDB Aggregation Framework (*MongoDB Inc., 2018a*); the most popular query language for NoSQL data handling.

In our formal development, we favour a theory-to-practice strategy to avoid inconsistent or counter-intuitive query behaviours, which is one of the significant drawbacks of the MongoDB Aggregation Framework implementation (*Botoeva et al., 2018*). Moreover, we consider the formalisation as a blueprint for implementors and thus we strive for a balance between abstraction and technical involvement: *(i)* we adopt constructive semantics definitions rather than declarative ones, since the former are more amenable to imperative implementations, and *(ii)* we define our semantics on trees rather than on sets (as done in *Botoeva et al. (2018)*), since the former is the data structure handled by the developers and their users.

Data structures: trees and paths

We start by defining trees and the primitives on which we define the semantics of Tquery.

We denote trees with t. A tree contains two elements: (i) a *root* value that we denote with b, which holds basic values (Booleans, integers, and strings) or the null value υ; (ii) a set of pairs $\{k : a\}$, where each pair associates a *key* k to an *array* of trees a. Formally:

$$t ::= b\ \{k_i : a_i\}_i \qquad a ::= [t_1, \ldots, t_n]$$

We indicate with $k(t)$ the extraction of the array pointed by the label k in t: if k is present in t we retrieve the related array, otherwise we return the null array α (different from the empty array, instead denoted with $[\,]$). Formally:

$$k(b\ \{k_i : a_i\}_i) = \begin{cases} a & \text{if } (k : a) \in \{k_i : a_i\}_i \\ \alpha & \text{otherwise} \end{cases}$$

We assume the range of a given array a to run from the minimum index (one) to the maximum, that corresponds to its cardinality, denoted with $\#a$. We indicate the extraction of the tree t at index i in array a with the index notation $a[i]$, *i.e.*, $a[i] = t$. In case a contains an element at index i we retrieve it, otherwise, we retrieve the null tree, denoted with τ. Formally:

$$a[i] = \begin{cases} t_i & \text{if } a = [t_1, \ldots, t_n] \wedge 1 \leq i \leq n \\ \tau & \text{otherwise} \end{cases}$$

We define the array concatenation operator, denoted with ::, such that $[t_1, \ldots, t_n] = [t_1] :: \ldots :: [t_n]$. Given two arrays a' and a'', the concatenation $a' :: a''$ returns an array a of size $\#a = \#a' + \#a''$ where elements $a[1], \ldots, a[\#a']$ correspond point-wise to elements $a'[1], \ldots, a'[\#a']$ and elements $a[\#a' + 1], \ldots, a[\#a' + \#a'']$ correspond point-wise to elements $a''[1], \ldots, a''[\#a'']$.

We define paths to express tree traversal, ranged over by p. Paths are concatenations of expressions, indicated with e (which we omit to define since orthogonal to Tquery), closed by the sequence termination ε. Formally:

$$p ::= \underline{e.p} \mid \varepsilon.$$

When possible, we omit to indicate sequence terminations ε in paths and we slightly abuse the notation by indicating the components of paths like $\underline{e.p}$ as $e.p$ to keep a lightweight

notation—this does not make the notation ambiguous since path concatenation is always contextually distinct.

The application of a path p to a tree t, written $[\![p]\!]^t$, returns an array that contains the sub-trees reached traversing t following p. To define $[\![p]\!]^t$, we introduce the notation $e \downarrow k$, read "e evaluates to k", and use it to indicate that the evaluation of the expression e in a path p results in the label k. Path application $[\![p]\!]^t$ neglects array indexes, *i.e.,* for $p = e.p'$, such that $e \downarrow k$, we apply the sub-path p' to all trees in the array pointed by k in t and concatenate all their results keeping their relative order—the resulting array can concatenate null arrays α too, as a result of applying the path on some (sub)trees that do not contain all nodes present in p.

$$[\![p]\!]^t = \begin{cases} [\![p']\!]^{t_1} :: \ldots :: [\![p']\!]^{t_n} & \text{if } p = e.p' \wedge e \downarrow k \wedge k(t) = \texttt{[}t_1, \ldots, t_n\texttt{]} \\ \texttt{[}t\texttt{]} & \text{if } p = \varepsilon \\ \alpha & \text{otherwise} \end{cases}$$

We illustrate the path application with the example below, where $t_1 = \text{sl}[1]$, *i.e.,* it is the first (and only) element in the `sl` data structure represented at Lines 7–16 of Listing 1. From now on, in the examples, we adopt the formal representation of trees defined at the beginning of the section.

$[[M.m.D.d]]^{t_1}$ = `[27{}, 28{}, 29{}, 30{}]`

In the remainder, to contract empty and null arrays, we assume the following structural equivalences when we perform array concatenations.

$$\alpha \equiv \alpha :: \alpha \qquad \alpha :: \texttt{[]} \equiv \texttt{[]} :: \alpha \equiv \texttt{[]} :: \texttt{[]} \equiv \texttt{[]} \qquad \alpha :: a \equiv a :: \alpha \equiv a :: \texttt{[]} \equiv \texttt{[]} :: a \equiv a$$

Tquery operators

In this section, we present each Tquery operator: examples of its usage, its formal syntax, and its semantics, with examples illustrating relevant steps. For reference, we report in Fig. 1 the syntax of the Tquery operators: *match* (μ), *unwind* (ω), *project* (π), *group* (γ), and *lookup* (λ). In the syntax, a denotes arrays, b denotes primitive values, and p, q, and r are paths. We define the parameters of the operators with four syntactic rules: φ for the match, Π and d for the project, and Γ for the group, explained in their relative sections.

The match operator

$\mu(a, \varphi)$ $\qquad \varphi ::= \texttt{true} \mid \exists p \mid p = a \mid p_1 = p_2 \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi$

The purpose of the *match* operator is to select trees in an array a according to a criterion φ, which can be (from left to right): (i) the Boolean truth, (ii) the existence of a path p in t, (iii) the equality between the application of a path p on t and a given array a, (iv) the equality between the applications of two paths p_1 and p_2 on t, and the logic connectives (v) negation, (vi) conjunction, and (vii) disjunction.

Example Here and in the following sections, we draw our examples from Listing 2. There, we see the match operator used twice: the first at Line 3 and the second at Line 11. Here, we focus on the example at Line 3. We comment the execution of Line 11 in 'The group operator', since we use it to filter out the unnecessary values from the `sl` data structure before the application of the *group*.

$$
\begin{aligned}
\text{operator} &::= \mu(a,\varphi) \mid \omega(a,p) \mid \pi(a,\Pi) \mid \gamma(a,\Gamma,\Gamma') \mid \lambda(a,q,a',r,p)\\
\varphi &::= \texttt{true} \mid \exists p \mid p = a \mid p_1 = p_2 \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi\\
\Pi &::= p \mid d \rangle p \mid p,\Pi \mid d \rangle p,\Pi\\
d &::= b \mid p \mid [d_1,\ldots,d_n] \mid \varphi \mid \varphi?d_1 : d_2\\
\Gamma &::= p \mid p \rangle p' \mid p,\Gamma \mid p \rangle p',\Gamma
\end{aligned}
$$

Figure 1 **Syntax of Tquery.**
Full-size DOI: 10.7717/peerjcs.1037/fig-1

At Line 3, we use a match to filter `tmp` from those trees that do not correspond to the time range of interest. For convenience, we report Line 3 of Listing 2 in the snippet below.

```
3 tmp←μ( tmp, (date=20201128 ∨ date=20201129 ∨ date=20201130) )
```

The execution takes as input the data structure `tmp` presented in Listing 1 and assigns to it the resulting data structure:

```
[ υ {date:[ 20201128{} ], t:[ 36{} ], hr:[ 66{} ] },
  υ {date:[ 20201129{} ], t:[ 36{} ], hr:[ 65{} ] },
  υ {date:[ 20201130{} ], t:[ 37{} ], hr:[ 67{} ] } ]
```

Semantics When applied to an array a, the match operator returns a new array in the shape of a but including only its elements that satisfy φ. If no element matches the criterion (and also in the case that $a = \alpha$), the operator returns an empty array `[ ]`.

$$
\mu(\alpha,\varphi) = [] \qquad \mu([t] :: a,\varphi) = \begin{cases} [t] :: \mu(a,\varphi) & \text{if } t \models \varphi \\ \mu(a,\varphi) & \text{if } \#a > 0 \\ [] & \text{otherwise} \end{cases}
$$

The semantics of $t \models \varphi$ is defined by the Boolean expressions below.

$$
t \models \varphi \text{ holds iff } \begin{cases} \varphi = true \\ \varphi = (\exists p) \wedge [\![p]\!]^t \neq \alpha \\ \varphi = (p = a) \wedge [\![p]\!]^t = a \\ \varphi = (p_1 = p_2) \wedge [\![p_1]\!]^t = [\![p_2]\!]^t \\ \varphi = (\neg\varphi') \wedge t \not\models \varphi' \\ \varphi = (\varphi_1 \wedge \varphi_2) \wedge (t \models \varphi_1 \wedge t \models \varphi_2) \\ \varphi = (\varphi_1 \vee \varphi_2) \wedge (t \models \varphi_1 \vee t \models \varphi_2) \end{cases}
$$

Example: semantics At Line 3 of Listing 2, the match evaluates all trees inside `tmp` and verifies which one of the sub-conditions hold for each element of `tmp`. In the case of `tmp`[1], the criterion is not satisfied and thus the value is discarded. Next, `tmp`[2] satisfies the first criterion $\underline{date} = \texttt{20201128}$, `tmp`[3] satisfies the second criterion $\underline{date} = \texttt{20201129}$, and `tmp`[4] satisfies the third criterion $\underline{date} = \texttt{20201130}$.

The unwind operator

$\omega(a,p)$

The purpose of the *unwind* operator is to unfold the elements of an array a under a given path p.

Example We exemplify the usage of *unwind* reporting Line 9 of Listing 2 in the snippet below and later showing the result of its application.

```
9 sl←ω( sl, M.D.L )
```

The unwind operator takes as input the sleep logs `sl` (as retrieved from the invocation of the `getSleepPatterns` operation at Line 8, and represented at Lines 7–16 of Listing 1). In the snippet, we update the content of `sl` to contain the new data structure, shown below.

```
[ υ {  y:[ 2020{} ],M:[ υ {m:[ 11{} ],D:[ υ {d:[ 27{} ],
   L:[ υ {  s:[ '23:33'{} ],e:[ '07:04'{} ],q:[ 'poor'{} ] }] }] }] },
  υ {  y:[ 2020{} ],M:[ υ {m:[ 11{} ],D:[ υ {d:[ 28{} ],
   L:[ υ {  s:[ '21:13'{} ],e:[ '09:34'{} ],q:[ 'good'{} ] }] }] }] },
  υ {  y:[ 2020{} ],M:[ υ {m:[ 11{} ],D:[ υ {d:[ 29{} ],
   L:[ υ {  s:[ '21:01'{} ],e:[ '03:12'{} ],q:[ 'good'{} ] }] }] }] },
  ... ]
```

Semantics To define the semantics of the unwind operator ω, we introduce an auxiliary operator, called *unwind expansion operator* and we indicate it with $\mathsf{ueo}(t,a,k)$ (read "unwind t on a under k"). Informally, $\mathsf{ueo}(t,a,k)$ returns an array of trees with cardinality $\#a$ where each element has the shape of t except that label k points to the corresponding index-wise element in a.

Formally, given a tree t, an array a, and a key k:

$$\mathsf{ueo}(t,a,k) = \begin{cases} [b((\{k_i : a_i\}_i \setminus \{k : k(t)\}) \cup \{k : [t']\})] :: \mathsf{ueo}(t,a',k) & \text{if } a = [t'] :: a' \\ & \wedge\, t = b\,\{k_i : a_i\}_i \\ [] & \text{otherwise} \end{cases}$$

Then, the formal definition of $\omega(a,p)$ is

$$\omega(a,p) = \begin{cases} \mathsf{ueo}(t, \omega([\![\underline{k.\varepsilon}]\!]^t, p'), k) :: \omega(a',p) & \text{if } p = e.p' \wedge e \downarrow k \wedge a = [t] :: a' \\ a & \text{if } p = \varepsilon \\ [] & \text{otherwise} \end{cases}$$

Essentially, the semantics of the unwind operator follows two inductive directions: one on arrays and the other on paths. Hence, to simplify the explanation of the semantics, we describe it following a spatial interpretation of the two directions: the induction on arrays is the "breadth" of the expansion while the induction on paths represents its "depth".

The first part of the breadth expansion corresponds to the induction over the array a, which results in the concatenation of the inductive application of the depth expansion of p over each element t of a. In turn, the depth expansion consists of a nested depth expansion with a breadth one. The depth expansion is represented by $\omega([\![\underline{k.\varepsilon}]\!]^t, p')$, which corresponds to the application of the unwind operator with path p'—the suffix of k in p—and on the array of subtrees found in t under the current path fragment k. The breadth expansion (which complements the breadth expansion on the array a) uses the unwind expansion operator (ueo) to apply the result of the nested depth expansion on all elements found under k in t.

Example: semantics We now report excerpts of the execution of the unwind operator at Line 9 of Listing 2 to exemplify both the unfolding of the breadth and depth expansions.

We remind that sl has the shape reported in Line 7 in Listing 1 and that the application at Line 9 of Listing 2 "unwinds" the sl data structure with path $\underline{M.D.L}$.

The first expansion we perform is the breadth expansion over the array sl. Since sl just contains one tree, *i.e.,* that for sleep logs of 2020, we just have one application of the ueo operator (the empty array `[]` at the right of the concatenation operator :: results from the "otherwise" branch of the definition of the unwind and from sl being structurally equivalent to sl[1] :: `[]`).

$$\mathsf{ueo}(\mathtt{sl}[1], \omega([\![\underline{M.\varepsilon}]\!]^{\mathtt{sl}[1]}, \underline{D.L}), M) :: \mathtt{[]}$$

Then, we show the "depth" part of the expansion, by focusing on the terminal part of the application of the ueo operator. Specifically, we concentrate on the tree corresponding to the sleep logs of day 2020-11-29, found at Line 11 of Listing 1 and aliased with the tree t_{29}. Formally, the expansion corresponds to the application $\mathsf{ueo}(t_{29}, [\![\underline{L.\varepsilon}]\!]^{t_{29}}, L)$ of the terminal node L in path $\underline{M.D.L}$.

```
ueo(t29,⟦L.ε⟧^t29, L) ⇒
[(ν{d:[29{}],L:[...]}\ν{L:[...]})∪ν{L:[ν{s:['21:01'{}],e:['03:12'{}],q:['good'{}
    ]}]}] ::
[(ν{d:[29{}],L:[...]}\ν{L:[...]})∪ν{L:[ν{s:['03:36'{}],e:['09:58'{}],q:['good'{}
    ]}]}]
```

Above, for each element of the array pointed by L, we create a new structure where we replace the original array associated with the key L with a new array containing only one element. For instance, the first element of the result takes the original structure found under D (`[`ν `{`d : `[29{}]`, L : `[... ] } ]`) and updates it to contain only the element υ `{` s: `[` `'21:01'` `{ } ]`, e: `[` `'03:12'` `{ } ]`, q: `[` 'good' `{ } ] }` associated to the node L.

The project operator

$\pi(a, \Pi) \quad \Pi ::= p \mid d \rangle p \mid p, \Pi \mid d \rangle p, \Pi \quad d ::= b \mid p \mid$ `[`$d_1, \ldots, d_n$ `]` $\mid \varphi \mid \varphi ? d_1 : d_2$

The purpose of the *project* operator is to modify the trees in an array a by projecting nodes, renaming node labels, or introducing new nodes, as described in the sequence of elements Π, which are either a path p or an injection ($\rangle$) of a *value definition* d into a path.

A value definition d can be (in the grammar, from left to right): (i) a value, (ii) a path, (iii) an array of value definitions, (iv) a criterion (φ) (cf. 'The match operator") or (v) a ternary expression on a criterion and two value definitions.

Example As done for the other operators, we draw our examples from Listing 2, where we have four usages of the project operator, the first at Line 5, the second at Line 10, the third at Line 13, and the fourth at Line 15. Here, we focus on the second example, at Line 10, reported in the snippet below. We comment on the others when exemplifying the *lookup* operator in 'The lookup operator'.

```
10 sl ← π( sl , (y ⟩ year , M.m ⟩ month , M.D.d ⟩ day , M.D.L.q ⟩ quality ) )
```

The projection at Line 10 takes the sl data structure resulting from the application of the unwind at Line 9 and performs a sequence of renaming over all tress within sl. For each tree, we perform the rename of the node y in *year* by moving the content of path $\underline{y}$

into the node corresponding to path *year*, represented by the fragment $y \rangle year$. Similarly, we move the content of $M.m$ under *month*, of $M.D.d$ under *day*, and of $M.D.L.q$ under *quality*. The result of the projection is the following flattened structure:

```
[ υ {year:[ 2020{} ], month:[ 11{} ], day:[ 27{} ], quality:[ 'poor'{} ] },
  υ {year:[ 2020{} ], month:[ 11{} ], day:[ 28{} ], quality:[ 'good'{} ] },
  υ {year:[ 2020{} ], month:[ 11{} ], day:[ 29{} ], quality:[ 'good'{} ] },
 ...]
```

Semantics We start by defining the auxiliary operators we use in the definition of the project. Auxiliary operators $\pi(a,p)$ and $\pi(t,p)$ formalise the application of a branch selection over a path p respectively over an array and a tree. Then, we define the auxiliary operator $\mathsf{eval}(d,t)$, which returns the array resulting from the evaluation of a value definition d over a tree t. Finally, we report the projection of an injection of a value definition d into a path p over a tree t, *i.e.*, $\pi(t, d \rangle p)$.

The projection $\pi(a,p)$ for a path p over an array a results in an array whose elements are the projection for p of the elements of a:

$$\pi(a,p) = \pi(\texttt{[}t_1,\ldots,t_n\texttt{]},p) = \texttt{[}\pi(t_1,p),\ldots,\pi(t_n,p)\texttt{]}$$

The projection $\pi(t,p)$ for a path p over a tree t implements the actual semantics of branch selection, where, given a path $e.p'$ with $e \downarrow k$, we remove all the branches k_i in $t = b\,\texttt{\{}k_i : a_i\texttt{\}}_i$ but k (if $k \in \{k_i\}_i$) and continue to apply the projection for the continuation p' over the (array of) sub-trees under k in t (*i.e.*, $[\![\underline{k.\epsilon}]\!]^t$). Formally:

$$\pi(t,p) = \begin{cases} \upsilon\,\texttt{\{}k : \pi([\![\underline{k.\epsilon}]\!]^t, p')\,\texttt{\}} & \text{if } [\![p]\!]^t \neq \alpha \wedge p = e.p' \wedge e \downarrow k \\ t & \text{if } p = \varepsilon \\ \tau & \text{otherwise} \end{cases}$$

The operator $\mathsf{eval}(d,t)$ evaluates the value definition d over the tree t and returns an array containing the result of the evaluation. Formally:

$$\mathsf{eval}(d,t) = \begin{cases} \texttt{[}d\,\texttt{\{}\texttt{\}}\texttt{]} & \text{if } d \in V \\ [\![d]\!]^t & \text{if } d \in P \\ \mathsf{eval}(d,t) :: \mathsf{eval}(d',t) & \text{if } d = \texttt{[}d\texttt{]} :: d' \\ \texttt{[}t \models \varphi\,\texttt{\{}\texttt{\}}\texttt{]} & \text{if } d = \varphi \\ \mathsf{eval}(d_1,t) & \text{if } d = \varphi ? d_1 : d_2 \wedge t \models \varphi \\ \mathsf{eval}(d_2,t) & \text{if } d = \varphi ? d_1 : d_2 \wedge t \not\models \varphi \\ \alpha & \text{otherwise} \end{cases}$$

The projection $\pi(t, d \rangle p)$ of the injection of the evaluation of a value definition d on a tree t into a path p results in a new tree where we find the evaluation of d on t under p.

$$\pi(t, d \rangle p) = \begin{cases} \upsilon\,\texttt{\{}k : \texttt{[}\pi(t, d \rangle p')\texttt{]}\,\texttt{\}} & \text{if } p = e.p' \wedge e \downarrow k \wedge \mathsf{eval}(d,t) \neq \alpha \\ \upsilon\,\texttt{\{}k : \mathsf{eval}(d,t)\,\texttt{\}} & \text{if } p = e.\varepsilon \wedge e \downarrow k \wedge \mathsf{eval}(d,t) \neq \alpha \\ \tau & \text{otherwise} \end{cases}$$

Before formalising the projection, we report the auxiliary operator $\oplus$ to merge arrays and trees—we use the operator to merge the result of sequences of projections in the definition of $\pi(t,\Pi)$.

$$(\texttt{[}t\texttt{]} :: a) \oplus (\texttt{[}t'\texttt{]} :: a') = \texttt{[}t \oplus t'\texttt{]} :: a \oplus a'$$

$$t \oplus \tau = t \qquad a \oplus \texttt{[]} = \texttt{[]} \oplus a = a \oplus \alpha = \alpha \oplus a = a$$

$$\frac{t = b\ \{k_i : a_i\ \}_{i\in I} \wedge t' = b\ \{k_j : a_j\ \}_{j\in J}}{t \oplus t' = b\ \{k_h : k_h(t) \oplus k_h(t')\ \}_{h\in I\cup J}} \qquad \frac{b \neq b'}{b\ \{k_i : a_i\ \}_i \oplus b'\ \{k_j : a_j\ \}_j = \tau}$$

To conclude, we first report the application of the projection to a tree t, $\pi(t, \Pi)$, which merges the results of projections in Π over t into a single tree. Second, we report the application of the projection to an array a, $\pi(a, \Pi)$, which corresponds to the application of the projection to all elements of a. Respectively, we formally write:

$$\pi(t,\Pi) = \begin{cases} \pi(t,p) \oplus (\pi(t,\Pi')) & \text{if } \Pi = p, \Pi' \\ \pi(t, d \rangle p) \oplus (\pi(t,\Pi')) & \text{if } \Pi = d \rangle p, \Pi' \\ \pi(t,p) & \text{if } \Pi = p \\ \pi(t, d \rangle p) & \text{if } \Pi = d \rangle p \end{cases}$$

and

$$\pi(a,\Pi) = \pi(\ [t_1,\ldots,t_n\ ],\Pi) = [\pi(t_1,\Pi),\ldots,\pi(t_n,\Pi)\ ] \qquad \pi(\ [\ ],\Pi) = \pi(\alpha,\Pi) = [\ ]$$

Example: semantics We report the execution of the project at Line 10 of Listing 2. We take `sl` as returned after the application of the unwind operator described in 'The unwind operator'. For brevity, we represent the `sl` data structure as the concatenation of its elements, *i.e.*, $\texttt{sl} = \texttt{sl}[1] :: \texttt{sl}[2] :: \cdots$.

$$\begin{aligned} &\pi(\texttt{sl}[1] :: \texttt{sl}[2] :: \cdots,\ (\underline{y} \rangle \underline{year},\ \underline{M.m} \rangle \underline{month},\ \underline{M.D.d} \rangle \underline{day},\ \underline{M.D.L.q} \rangle \underline{quality})) \Rightarrow \\ &\quad [\ \pi(\texttt{sl}[1], (\underline{y} \rangle \underline{year},\ \underline{M.m} \rangle \underline{month},\ \underline{M.D.d} \rangle \underline{day},\ \underline{M.D.L.q} \rangle \underline{quality})), \\ &\qquad \pi(\texttt{sl}[2], (\underline{y} \rangle \underline{year},\ \underline{M.m} \rangle \underline{month},\ \underline{M.D.d} \rangle \underline{day},\ \underline{M.D.L.q} \rangle \underline{quality})),\ \ldots\ ] \end{aligned}$$

We continue showing the projection of the first element in a, $\texttt{sl}[1]$ (the projection on the other elements follows the same structure)

$$\begin{aligned} &\pi(\texttt{sl}[1], (\underline{y} \rangle \underline{year},\ \underline{M.m} \rangle \underline{month},\ \underline{M.D.d} \rangle \underline{day},\ \underline{M.D.L.q} \rangle \underline{quality})) \Rightarrow \\ &\quad \pi(\texttt{sl}[1], \underline{y} \rangle \underline{year}) \ \oplus\ \pi(\texttt{sl}[1], \underline{M.m} \rangle \underline{month}) \ \oplus\ \pi(\texttt{sl}[1], \underline{M.D.d} \rangle \underline{day}) \ \oplus\ \pi(\texttt{sl}[1], \underline{M.D.L.q} \rangle \underline{quality}) \end{aligned}$$

Finally, we show the unfolding of the first two projections from the left, above, *i.e.*, those for $\underline{y} \rangle \underline{year}$ and for $\underline{M.m} \rangle \underline{month}$, and their merge $\oplus$ (the remaining ones unfold similarly).

$$\begin{aligned} &\pi(\texttt{sl}[1], \underline{y} \rangle \underline{year}) \oplus \pi(\texttt{sl}[1], \underline{M.m} \rangle \underline{month}) \\ &\Rightarrow \upsilon\ \{\ \texttt{year:}\ \pi(\texttt{sl}[1], \underline{y})\ \} \oplus \upsilon\ \{\ \texttt{month:}\ \ \pi(\texttt{sl}[1], \underline{M.m})\ \} \\ &\Rightarrow \upsilon\ \{\ \texttt{year:}\ \mathsf{eval}(\underline{y}, \texttt{sl}[1])\ \} \oplus \upsilon\ \{\ \texttt{month:}\ \ \mathsf{eval}(\underline{M.m}, \texttt{sl}[1])\ \} \\ &\Rightarrow \upsilon\ \{\ \texttt{year:}\ [\![\underline{y}]\!]^{\texttt{sl}[1]}\ \} \oplus \upsilon\ \{\ \texttt{month:}\ [\![\underline{M.m}]\!]^{\texttt{sl}[1]}\ \} \\ &\Rightarrow \upsilon\ \{\ \texttt{year:}\ [\texttt{2020\{\}}]\ \} \oplus \upsilon\ \{\ \texttt{month:}\ [\texttt{11\{\}}]\ \} \\ &\Rightarrow \upsilon\ \{\ \texttt{year:}\ [\texttt{2020\{\}}],\ \texttt{month:}\ [\texttt{11\{\}}]\ \} \end{aligned}$$

The group operator

$$\gamma(a, \Gamma, \Gamma') \qquad \Gamma ::= p \mid p \rangle p' \mid p, \Gamma \mid p \rangle p', \Gamma$$

The purpose of the *group* operator is to group the trees in an array a according to a specification Γ' and to aggregate the values of the grouped trees according to the specification Γ. Both Γ and Γ', respectively the *aggregation* and the *grouping* set, are sequences of elements of the form $p \rangle p'$ where p is a path in the input trees, and p' a path in the output trees.

Note that Γ includes both fragments of the shape p and $p \rangle p'$. Here, the former is syntactic sugar for the latter, where both paths are the same. Therefore, we assume to apply the semantics of the group operator only with the de-sugared form $\gamma(a, \Gamma, \Gamma') = \gamma(a, \mathsf{exp}(\Gamma), \mathsf{exp}(\Gamma'))$, where

$$\mathsf{exp}(\Gamma_1, \Gamma_2) = \mathsf{exp}(\Gamma_1), \mathsf{exp}(\Gamma_2) \qquad \mathsf{exp}(p) = p \rangle p \qquad \mathsf{exp}(q \rangle p) = q \rangle p$$

Example Drawing from Listing 2, we have two applications of the group operator, one at Line 4 and the second at Line 12. Since the two applications are similar, we just focus on the latter (reported below), leaving the comment on the second to 'The lookup operator'.

```
12 sl←γ( sl, (quality), () )
```

As stated above, the aggregation set expands from *quality* to the de-sugared form *quality* ⟩ *quality*.

The group operator applies on the data structure in `sl` which, at Line 11, we filtered with the match operator to only contain values corresponding to the dates 2020-11-29 and 2020-11-30. The new data structure, copied into `sl` and reported below, is essentially the aggregation under the node *quality* of the filtered sleep recordings.

```
[ υ {quality:[ 'good'{}, 'good'{}, 'poor'{}, 'good'{} ] } ]
```

To make for a more comprehensive illustration, in this section we consider an alternative version of the example above, where we want to use the group operator to group the values by *day*, *month*, and *year* and aggregate the values of the sleep *quality*. Concretely, we do this by updating the command found at Line 12 with the sequence of paths replacing the third parameter, which in the original we left empty.

```
sl←γ( sl, (quality), (day, month, year) )
```

As stated, the paths *quality*, *day*, *month*, and *year* respectively expand to *quality* ⟩ *quality*, *day* ⟩ *day*, *month* ⟩ *month*, and *year* ⟩ *year*.

The main detail we want to notice here is that, by grouping the values by *year*, *month*, and *day*, we only aggregate logs relative to the same day.

```
[ υ {year:[2020{}],month:[11{}],day:[29{}],quality:['good'{},'good'{}]},
  υ {year:[2020{}],month:[11{}],day:[30{}],quality:['poor'{},'good'{}]}
]
```

Semantics We start by reminding the shape of the de-sugared syntax of the group operator.

$$\gamma(a, \Gamma, \Gamma') = \gamma(a, \mathsf{exp}(\Gamma), \mathsf{exp}(\Gamma')) = \gamma(a\ ,\ \underbrace{q_1 \rangle p_1, \ldots, q_n \rangle p_n}_{\text{aggregation set } \mathcal{A}}\ ,\ \underbrace{s_1 \rangle r_1, \ldots, s_m \rangle r_m}_{\text{grouping set } \mathcal{G}})$$

Intuitively, the group operator performs the following actions:

(a) it groups together those trees in a that (1) have the maximal number of existing paths from the grouping set $s_1, \ldots, s_m$ and (2) whose values under those paths coincide;

(b) it projects the values in the grouped trees from $s_1, \ldots, s_m$ to the corresponding paths $r_1, \ldots, r_m$;

(c) it aggregates all the values in the grouped trees found under the paths $q_1, \ldots, q_n$ from the aggregation set;

(d) it projects the aggregated values from $q_1, \ldots, q_n$ into the corresponding paths $p_1, \ldots, p_n$.

Formally, let $S = \{s_1, \ldots, s_m\}$ be the set of left elements in the injections of the sequence in the grouping set and let Σ be the power-set 2^S of paths in S so that

$$\Sigma = \{\varnothing, \{s_1\}, \{s_2\}, \{s_3\}, \ldots, \{s_1, s_2\}, \{s_1, s_3\}, \ldots, \{s_1, \ldots, s_m\}\} = \{\sigma_1, \ldots, \sigma_k\}$$

We define the auxiliary operator exists which takes S and an element $\sigma \in \Sigma$ and builds the *existence-match-query* formula of the paths in S w.r.t. the combination identified by σ.

$$\mathsf{exists}(\sigma, S) = \begin{cases} \texttt{true} & \text{if } S = \varnothing \\ \exists s \wedge \mathsf{exists}(\sigma, S \setminus \{s\}) & \text{let } s \in S \text{ and } s \in \sigma \\ \neg\exists s \wedge \mathsf{exists}(\sigma, S \setminus \{s\}) & \text{let } s \in S \text{ and } s \notin \sigma \end{cases}$$

We use the exists operator to perform part 1) of Item (a), *i.e.,* grouping those trees in a so that the trees in the same group have the same set of existing and non-existing paths from $s_1, \ldots, s_m$. The part operator (presented below) performs part 2) of Item (a), which is the partition of the trees grouped by the exists operator so that the values in their existing paths in $s_1, \ldots, s_m$ coincide.

We now define the semantics of the group operator and then present the semantics of the part operator. In the remainder, to make the definitions more intuitive, we alias the aggregation set with $\mathcal{A}$ and the grouping set with $\mathcal{G}$. Let, $k = |\Sigma|$, we write

$$\gamma(a, \mathcal{A}, \mathcal{G}) = \mathsf{part}(\mu(a, \mathsf{exists}(\sigma_1, S)), \sigma_1, \mathcal{A}, \mathcal{G}) :: \cdots :: \mathsf{part}(\mu(a, \mathsf{exists}(\sigma_k, S)), \sigma_k, \mathcal{A}, \mathcal{G})$$

As mentioned, the part operator finds the elements of a which should be grouped together according to $\mathcal{G}$ (among those selected through σ). In the definition, we delegate the actual grouping to the other auxiliary operator group, which (as hinted in Item (b)) projects the partitioned values from S into the corresponding destination path $r_1, r_2, \ldots$ in $\mathcal{G}$. The group operator also performs the aggregation of the values found in $q_1, q_2, \ldots$ (Item (c)) and it projects them under the corresponding destination path $p_1, p_2, \ldots$ (Item (d)).

In the semantics of the part operator, we assume to extend the set difference $\setminus$ to arrays, so that $a \setminus a'$ returns a copy of a without the elements found in a' (preserving their relative order). We also assume to have a variant of the match operator $\mu^{id}(a, \varphi)$ that, instead of returning the array of trees in a that match the criterion φ, it returns the array of their indexes in a.

$$\mathsf{part}(a, \sigma, \mathcal{A}, \mathcal{G}) = \begin{cases} a & \text{if } a = [\,] \\ \mathsf{group}(a, \sigma, \mathcal{A}, \mathcal{G}) & \text{if } \sigma = \varnothing \\ \mathsf{group}([a[j], \ldots, a[k]], \sigma, \mathcal{A}, \mathcal{G}) & \text{otherwise, let } \sigma = \{s_1, \ldots, s_i\}, \\ \quad :: \mathsf{part}([a[f], \ldots, a[g]], \sigma, \mathcal{A}, \mathcal{G}) & \mu^{id}(a, \bigwedge_{j=1}^{i} s_j = [\![s_j]\!]^{a[1]}) = [j, \ldots, k], \\ & [f, \ldots, g] = [1, \ldots, \#a] \setminus [j, \ldots, k] \end{cases}$$

Finally, we report below the definition of the group operator. There, the last case is where we aggregate the values found in the array a following the paths in $\mathcal{A}$, and we combine them with the grouped values from $\mathcal{G}$ by using the project operator. The aggregation of the values in a is done by invoking the group operator on the second case. The second case applies when $\sigma = \varnothing$ (*i.e.,* when no path S is selected for grouping). The result of the

application of the second case is an array containing one tree that combines the values of the array a following the paths in $\mathcal{A}$. To aggregate the values, we use the auxiliary tree variant of the project operator ($\pi(t,\Pi)$, cf. 'The project operator') to project each value for a given path q into its corresponding path p in $\mathcal{A}$.

$$\mathsf{group}(a,\sigma,\mathcal{A},\mathcal{G}) = \begin{cases} a & \text{if } a = \texttt{[]} \\ \texttt{[}\underbrace{\pi(\tau,\eta_1\rangle p_1,\ldots,\eta_n\rangle p_n)}_{\textit{aggregation}}\texttt{]} & \text{if } \sigma=\varnothing, \quad \text{let} \quad \mathcal{A}=q_1\rangle p_1,\ldots,q_n\rangle p_n, \\ & \eta_j=\pi(a,q_j), \quad j\in[1,n] \\ \pi(a',\underbrace{[\![s_i]\!]^{a[1]}\rangle r_i,\ldots,[\![s_j]\!]^{a[1]}\rangle r_j}_{\textit{grouping}}) & \text{otherwise, let } a'=\mathsf{group}(a,\varnothing,\mathcal{A},\mathcal{G}) \\ & \mathcal{G}=s_1\rangle r_1,\ldots,s_m\rangle r_m, \\ & \sigma=\{s_i,\ldots,s_j\}, 1\le i\le j\le m \end{cases}$$

Example: semantics To illustrate the semantics of the group operator, we consider the alternative version of the code shown at Line 12 (and presented as a second example at the beginning of this section), where we want to aggregate for *quality* but we also want to keep those values grouped by *year*, *month*, and *day*.

```
sl ← γ( sl, (quality), (day, month, year) )
```

In the semantics, the first thing we do is the de-sugaring of paths—namely *quality*, *day*, *month*, and *year*, which respectively expand to *quality* ⟩ *quality*, *day* ⟩ *day*, *month* ⟩ *month*, and *year* ⟩ *year*—and then we apply the de-sugared group operator on `sl` (which, we remind, contains only values corresponding to the dates 2020-11-29 and 2020-11-30, represented by the trees $t^1_{29}, t^2_{29}, \ldots$ below).

$$\begin{aligned}
&\gamma(\texttt{sl},(\textit{quality}),(\textit{day},\textit{month},\textit{year})) \\
&\Rightarrow \gamma(\texttt{sl},(\textit{quality}\rangle\textit{quality}),(\textit{day}\rangle\textit{day},\textit{month}\rangle\textit{month},\textit{year}\rangle\textit{year})) \\
&\text{let} \quad S = \{\textit{day},\textit{month},\textit{year}\}, \\
&\qquad \mathcal{A} = \textit{quality}\rangle\textit{quality}, \text{ and} \\
&\qquad \mathcal{G} = \textit{day}\rangle\textit{day},\textit{month}\rangle\textit{month},\textit{year}\rangle\textit{year} \\
&\Rightarrow \mathsf{part}(\mu(\texttt{sl},\mathsf{exists}(\varnothing,S)),\varnothing,\mathcal{A},\mathcal{G}) :: \cdots :: \mathsf{part}(\mu(\texttt{sl},\mathsf{exists}(S,S)),S,\mathcal{A},\mathcal{G}) \\
&\Rightarrow \mathsf{group}(\texttt{[]},\varnothing,\mathcal{A},\mathcal{G}) :: \cdots :: \mathsf{group}(\texttt{[}t^1_{29},t^2_{29}\texttt{]},S,\mathcal{A},\mathcal{G}) :: \mathsf{group}(\texttt{[}t^1_{30},t^2_{30}\texttt{]},S,\mathcal{A},\mathcal{G}) \\
&\Rightarrow \texttt{[]} :: \cdots \\
&\qquad :: \pi(\texttt{[}\upsilon\ \texttt{\{quality:['good'\{},\texttt{'good'\{}\texttt{]\}]},(\texttt{[29\{}\texttt{]}\rangle\textit{day},\texttt{[11\{}\texttt{]}\rangle\textit{month},\texttt{[2020\{}\texttt{]}\rangle\textit{year})) \\
&\qquad :: \pi(\texttt{[}\upsilon\ \texttt{\{quality:['poor'\{},\texttt{'good'\{}\texttt{]\}]},(\texttt{[30\{}\texttt{]}\rangle\textit{day},\texttt{[11\{}\texttt{]}\rangle\textit{month},\texttt{[2020\{}\texttt{]}\rangle\textit{year})) \\
&\Rightarrow \texttt{[}\upsilon\texttt{\{year:[2020\{}\texttt{]},month:[11\{}\texttt{]},day:[29\{}\texttt{]},quality:['good'\{},'good'\{}\texttt{]\}},} \\
&\qquad \upsilon\texttt{\{year:[2020\{}\texttt{]},month:[11\{}\texttt{]},day:[30\{}\texttt{]},quality:['poor'\{},'good'\{}\texttt{]\}]}
\end{aligned}$$

The lookup operator

$\lambda(a,q,a',r,p)$

The purpose of the *lookup* operator is to join the trees in a source array a with the trees in an adjunct array a'. For those values obtained by applying the path q on a, the lookup operator pairs them with the equivalent values obtained by applying r on the adjunct array a' and it projects the latter under path p in the paired trees of a.

Example Before commenting on the application of the lookup in Listing 2, we describe the results of the group at Line 4 and of the two projections, respectively at Line 5 and Line 13. At Line 4, we aggregate the temperatures in the `tmp` data structure, which results into

```
[ υ {t:[ 36{}, 36{}, 37{} ] } ]
```

The projection at Line 5 performs two actions over the `tmp` data structure. First, it keeps only the node t (holding the temperatures filtered for the days of interest). Second, it projects into the filtered data structure the pseudo-identifier (`pseudoID`) under the node *patient_id*.

```
[ υ {t:[ 36{}, 36{}, 37{} ], patient_id:[ 'id_xxx'{} ]} ]
```

The projection at Line 13, similar to the one above, keeps only the node *quality* (holding the quality of the sleep for the days of interest) and it projects the `pseudoID` under the node *patient_id*.

```
[ υ {quality:[ 'good'{}, 'good'{}, 'poor'{}, 'good'{} ],
     patient_id:[ 'id_xxx'{} ] } ]
```

We can now comment on the lookup at Line 14, which we report below for convenience.

```
14 bs←λ( sl, patient_id, tmp, patient_id, temperatures )
```

The instruction joins the data structures `tmp` and `sl` by pairing the values under the path *patient_id* (this is a special case where the left and right paths of the join coincide, *i.e.,* the path *patient_id*). The last path in the application, *i.e.,* *temperatures*, indicates where the values from the right data structure (`tmp`) should be projected in the paired values of the left one (`sl`).

At Line 14, we store the result of the application of the lookup into a new variable `bs` (standing for bio-signals).

```
υ {quality:[ 'good'{}, 'good'{}, 'poor'{}, 'good'{} ],
   temperatures:[ υ {t: [ 36{}, 36{}, 37{} ],
                    patient_id: [ 'id_xxx'{} ] } ],
   patient_id: [ 'id_xxx'{} ] }
```

For completeness, we report the result of the last step of Listing 2, at Line 15, where we apply the project operator to reshape the data structure for the invocation of the `detectEncephalopathy` functionality at Line 16.

```
  υ {quality:[ 'good'{}, 'good'{}, 'poor'{}, 'good'{} ],
     temperatures:[ 36{}, 36{}, 37{} ],
     patient_id: [ 'id_xxx'{} ] }
```

Semantics In the semantics of the lookup, for each element $a[i]$ $(1 \leq i \leq \#a)$, we use the tree version of the project operator ($\pi(t, \Pi)$, cf. 'The project operator') to merge the element $a[i]$ with the paired values from a' under r. Since, by its definition, $\pi(t, \Pi)$ corresponds to the merging ($\oplus$) of the single applications of each component in the sequence Π, we use this to merge the source tree $a[i]$ with the paired elements in a'. Hence, for each element $a[i]$, we define Π_i as the sequence ε, $\mu(a', \varphi_i) \rangle p$. The projection for the first component (ε) returns the original tree ($a[i]$). The projection for the second component ($\mu(a', \varphi_i) \rangle p$) injects the result of the match $\mu(a', \varphi_i)$ into the path p, where the criterion φ_i, equal to $r = [\![q]\!]^{a[i]}$, selects those values in a' that under r coincide with the array found under q in $a[i]$.

Note that when for some i we have q not present in $a[i]$ (*i.e.*, $[\![q]\!]^{a[i]} = \alpha$) the lookup operator joins $a[i]$ with those trees in a' where r does not exist (*i.e.*, $\mu(a', r = \alpha)$).

$$\lambda(a, q, a', r, p) = \texttt{[}\pi(a[1], \Pi_1)\texttt{]} :: \cdots :: \texttt{[}\pi(a[\#a], \Pi_{\#a})\texttt{]} \quad \text{where} \quad \begin{cases} 1 \leq i \leq \#a \\ \Pi_i = \varepsilon, \mu(a', \varphi_i) \rangle p \\ \varphi_i = (r = [\![q]\!]^{a[i]}) \end{cases}$$

Example: semantics Below, we report the unfolding of the execution of the lookup at Line 14. Since we have one value in `sl`, we do not perform a concatenation of arrays but we just apply the projection for `sl`[1]. In the three reductions below, first, we retrieve the content of $[\![\underline{patient\_id}]\!]^{\texttt{sl[1]}}$, then, we execute the match (which essentially returns the whole content of the `tmp` variable), and, finally, we merge `sl`[1] (obtained by the projection under ε) with the result of the match projected under path *temperatures*.

```
[π(sl[1],(ε,μ(tmp,patient_id = [[patient_id]]^sl[1])) ⟩ temperatures))]
⇒ [π(sl[1],(ε,μ(tmp,patient_id = ['id_xxx'{}] )) ⟩ temperatures))]
⇒ [π(sl[1],(ε,[υ {t:[36{},36{},37{}],patient_id:['id_xxx'{}]}]) ⟩ temperatures))]
⇒ [υ {quality:['good'{},'good'{},'poor'{},'good'{}],
      patient_id:['id_xxx'{}]} ⊕ υ {temperatures:[υ {t:[36{},36{},37{}],
                                    patient_id:['id_xxx'{}]}]}]
```

IMPLEMENTATION

We now present Jolie/Tquery, our implementation of Tquery as a Jolie microservice. Specifically, we chose to release Jolie/Tquery as a library that users can include and invoke locally in their Jolie projects—as an npm package (https://www.npmjs.com/package/@jolie/tquery.). However, thanks to Jolie's module system, users can also expose Jolie/Tquery as an independent service, *e.g.*, as a RESTful service (*Montesi, 2016*) as well as a publish/subscribe MQTT worker (*Gabbrielli et al., 2018*) (as briefly detailed in 'The implementation of Jolie/Tquery').

In this section, first, we describe the main components of Jolie/Tquery, specified through the abstractions provided by the Jolie language (which follow the typical partition of microservice components (*Giallorenzo et al., 2021*)), namely: its Application Programming Interfaces (API), its access points, and its logic/behaviour. In particular, APIs and access points[2] describe how users interact with Jolie/Tquery, while the behaviour implements the semantics of Tquery (cf. 'The Tquery Formalisation').

Then, we slightly extend the API and behaviour of Jolie/Tquery to support query *pipelines*, *i.e.,* multi-stage queries where (*a*) the first stage uses the data provided as input, (*b*) each other stage transforms the data from the proceeding stage, and (*c*) the last stage returns its output back to the invoker. We have two main reasons for extending Jolie/Tquery with pipelines: (*i*) for efficiency, since it removes the overhead of data transmission between sequential stages (as, *e.g.*, in Listing 2 at Lines 3–5 and Lines 9–15); (*ii*) for familiarity with the MongoDB Aggregation Framework (*MongoDB Inc., 2022*), where users express queries as multi-stage transformations.

[2] These specify the network, transport, and application protocols, *e.g.*, HTTP/TCP/IP.

Finally, we show the implementation of the example from 'Overview and Running Example' in Jolie/Tquery, both using the original sequence of operators (cf. 'Overview and Running Example') and as a combination of multi-stage pipelines.

The implementation of Jolie/Tquery

We start from the API of Jolie/Tquery and then present how Jolie allows us to provide the microservice as a library and to also have an efficient implementation of its engine.

The Jolie/Tquery API Simplifying (*Giallorenzo et al., 2021*; *Montesi, Guidi & Zavattaro, 2014*), in Jolie, the API of a microservice corresponds to an `interface`, which is a named collection of resources, called operations, each defined by a name, an interaction modality—i.e., asynchronous invocations or synchronous request responses (*W3c, 2001*)—and schemas of their expected inbound and outbound data, called `type`s. Thus, in Fig. 2, we report the API of Jolie/Tquery expressed as a Jolie `interface`, with its associated `type`s.

The code in Fig. 2 is a fragment of the `main.ol` [3] executable Jolie file from Jolie/Tquery. In Fig. 2, we stylise the code omitting `void` root types (described in the following paragraph) and naming `type`s using the symbols from the formalisation. These conventions help keeping the code compact and also ease the comparison with Tquery, in unison with the boxed fragments reporting the Tquery syntax in Fig. 2.

[3] Available at https://github.com/jolie/tquery/blob/master/main.ol.

We briefly introduce the main elements of Jolie APIs and we comment on the choices that drove the design of the Jolie/Tquery API. At Lines 1–8 of Fig. 2, we find the definition of `TqueryInterface`, the Jolie/Tquery `interface`. The keyword `requestResponse` indicates that the operations associated to it (as a comma-separated list) are synchronous invocations, where the caller waits for the callee (here, the Jolie/Tquery service) to reply with the computed response. We defined all the operations of Jolie/Tquery as `requestResponse`s since this interaction modality matches the invocation semantics of the Tquery operators.[4]

[4] A possible alternative, here, is using asynchronous `oneWay`s and either choose a pull or push semantics to retrieve the results of the queries. We did not pursue this direction, since this modality would sensibly diverge from that of Tquery.

In the syntax of operations, *e.g.*, `match(μType)(QueryResponse)` at Line 3, we find the name of the operation (`match`), the request `type` between the first parenthesis (`μType`), and the response `type` between the second parenthesis (`QueryResponse`).

A Jolie `type` has a name, *e.g.*, `QueryResponse` at Line 9, and a shape similar to that of the trees described in 'The Tquery Formalisation': a root that contains a value (*e.g.*, `bool`, `int`, `string`, as well as the empty value, `void`) and sub-nodes that point to quantified arrays of typed trees, *e.g.*, the `QueryResponse` `type` has a `void` root (omitted) and a sub-node named `result` which points to an unbounded array (`*`) of elements that can assume any shape (`undefined`).

Jolie `type`s can be further refined, *e.g.*, at Line 10, we restrict the set of strings that the root of the `type` `Path` can assume to those matching the regular expression within the `regex` predicate, following the definition of paths from Tquery.

Jolie `type`s support sum types (*Pierce, 2002*, Chapter 11) (*Safina et al., 2016*) of the shape `type Name: LeftType|RightType`. Here, we use sum types to keep the syntax of Tquery and the structure of Jolie/Tquery `type`s close. For example, at Line 12, we specify that the `type` φ can either be a `bool`ean, the `type` $\exists p$, etc..

The Jolie/Tquery access points and behaviour We now move to the description of the access points and the behaviour of Jolie/Tquery, reported in Listing 3. In Jolie, a microservice

```
interface TqueryInterface {
  requestResponse:
  match  ( μType )( QueryResponse ),
    unwind ( ωType )( QueryResponse ),
    project( πType )( QueryResponse ),
    group  ( γType )( QueryResponse ),
    lookup ( λType )( QueryResponse )
}
type QueryResponse: { result*: undefined }
type Path: string(regex("([A-Za-z_]\\w*\\.)*([A-Za-z_]\\w*)"))
type μType: { data*: undefined   query: φ }
type φ: bool | ∃p | EQ_Exp | NOT | AND | OR
    type ∃p: {exists : Path }
    type EQ_Exp: { equal: EQ_Data | EQ_Path }
    type EQ_Data: { data*: undefined   path: Path }
    type EQ_Path: { left: Path right: Path }
    type NOT: { not: φ }
    type AND: { and: { left: φ  right: φ }}
    type OR: { or: { left: φ  right: φ }}
type ωType: {data*: undefined   query: Path }
type πType: { data*: undefined   query[1,*]: Π }
  type Π: Path | Inject
  type Inject: { dstPath: Path   value[1,*]: d }
  type d: any | ValuePath | ValueMatch | ValueTernary
    type ValuePath: { path: Path }
    type ValueMatch: { match: φ }
    type ValueTernary: {
      condition: φ
      ifTrue[1,*]: d
      ifFalse[1,*]: d
    }
type γType: {data*: undefined   query: Group_Exp }
  type Group_Exp: {
    aggregate*: Γ
    groupBy*: Γ
  }
    type Γ: {dstPath: Path   srcPath: Path }
type λType: {
  leftData*: undefined
  leftPath: Path
  rightData*: undefined
  rightPath: Path
  dstPath: Path
}
```

For reference, we report the syntax of each Tquery operator next to its types.

$p ::= e.p \mid \varepsilon$

$\mu(a,\varphi)$
$\varphi ::= \texttt{true} \mid \exists p \mid p = a \mid p_1 = p_2 \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi$

$\omega(a,p)$

$\pi(a,\Pi)$
$\Pi ::= p \mid d \rangle p \mid p,\Pi \mid d \rangle p,\Pi$
$d ::= b \mid p \mid [d_1,\ldots,d_n] \mid \varphi \mid \varphi ? d_1 : d_2$

$\gamma(a,\Gamma,\Gamma')$
$\Gamma ::= p \mid p \rangle p' \mid p,\Gamma \mid p \rangle p',\Gamma$

$\lambda(a,q,a',r,p)$

Figure 2 **Mapping between the Tquery operators and Jolie/Tquery API.**
Full-size DOI: 10.7717/peerjcs.1037/fig-2

is identified by the keyword `service` associated with a name (in Listing 3, `Tquery`), a set of access points (in Listing 3, the `inputPort` at Lines 46–49), and a set of behaviours (in Listing 3, defined through the `foreign` language (`java`) at Lines 51–53).

```
service Tquery {
  inputPort IP {
    location: "local"
    interfaces: TqueryInterface
```

```
  }

  foreign java {
    class: "joliex.tquery.engine.TqueryService"
  }
}
```

Listing 3: The Tquery service.

Concerning access points, Jolie provides `inputPort`s to specify ingress gates, which define how a service expects clients to invoke its operations, and `outputPort`s (absent in Listing 3), which specify outbound egress gates for invoking other services (*Montesi, Guidi & Zavattaro, 2014*). At Line 46 of Listing 3, we define an `inputPort` (its name is immaterial here) with `location "local"` and `interfaces TqueryInterface` (cf. Fig. 2). By specifying an inbound access point with a `"local" location`, we indicate that our service accepts in-memory invocations from another Jolie service that runs Jolie/Tquery as an internal library—through a mechanism called "embedding" (*Montesi, Guidi & Zavattaro, 2014*).[5]

Regarding behaviours, Jolie provides a high-level language (akin to process calculi (*Montesi, Guidi & Zavattaro, 2014*)) to specify the composition of sophisticated workflows (*Gabbrielli, Giallorenzo & Montesi, 2014*) through a clean and minimal syntax. Jolie also supports the specification of behaviours through lower-level languages, like Java and Javascript, which are useful when integrating/exposing existing libraries as services or to manage lower-level abstractions like threads and pointers for performance. Jolie/Tquery falls in the latter category and we implemented its behaviour (and, thus, the Tquery semantics) using Java. This is visible at Lines 51–53 of Listing 3, where we declare the usage of the `foreign` language `java` to specify the `service` behaviour (implemented within the `TqueryService` class under the class-path `joliex.tquery.engine`). We omit the presentation of the Java code, since it closely follows the logic presented in 'The Tquery Formalisation'.

Extending Jolie/Tquery with query pipelines

Besides providing a faithful implementation of Tquery, we decided to extend Jolie/Tquery to support multi-stage queries both for reasons of performance and familiarity with the MongoDB Aggregation framework (*MongoDB Inc., 2022*).

The extension is minimal and provides an interesting point for showcasing the flexibility of the Jolie language in evolving existing projects.

Namely, the extension regards the API and the behaviour. We report in Listing 5 the changes to the Jolie/Tquery API and we omit, as done above, to present the Java code of the implementation, which is a straightforward sequentialisation of calls to the other implemented operators.

In the API, we add the `pipeline` operation among the operations in the `TqueryInterface` `interface`. The new operation requires an associated request `type` that contains the specification of the multi-stage queries. Having defined the `type`s of the other operations as independent components comes in handy. Indeed, the `Pipeline` `type` defines its multi-stage query as an array (under the sub-node `pipeline`) of subtrees specified through the `type`s of the other operations. For instance, at Line 11 in Listing 5, a `match` (`Query`) stage has the structure of the φ `type`, which is also the one used by the `match` operation (in the μ`Type` `type`).

[5] Jolie access points simplify the definition of alternative service configurations. For instance, to expose Jolie/Tquery as a RESTful service, we need to add a new `inputPort` (or change the one already defined) setting its `location` to a socket address (*e.g.*, `"socket://localhost:8080"`) and its `protocol` to `http` (*Montesi, 2016*). In general, `protocol`s in Jolie specify the mapping between protocol-specific resources and Jolie operations and their data serialisation. Since the `"local"` `location` transfers in-memory data structures, the definition of a `protocol` is unnecessary.

Here, the only exception is the `type` `λType`, which we did not use for the node `lookupQuery`, since the `leftData` sub-node is absent as the pipeline provides the (left-side) data.

```
interface TqueryInterface {
  RequestResponse:
  match ( μType )( QueryResponse ),
  //...
  pipeline( Pipeline )( QueryResponse )
}

type Pipeline: {
  data*: undefined
  pipeline[1,*]:
    { matchQuery          : φ }
    | { projectQuery[1,*] : Π }
    | { unwindQuery       : Path  }
    | { groupQuery        : Group_Exp }
    | { lookupQuery       : {
        leftPath  : Path
        rightData*: undefined
        rightPath : Path
        dstPath   : Path
    }
  }
}
```

Listing 4: Pipeline support extension (fragments).

The curious reader could wonder why we did not specify the whole Jolie/Tquery interface through the single `pipeline` operation. Our point is that, by having both possibilities, users can opt for the modality that best suits their scenario. For instance, when developing and debugging a query, it is useful to look at the shape of the single invocations and responses. Moreover, while pipelines help to make local sequential invocations efficient, they make the code harder to distribute, since the query now lives as an indivisible data structure. On the contrary, if we found out that a specific stage of a query, *e.g.*, the match at Line 3 or the unwind at Line 9 of Listing 2, would benefit from scaling it over multiple copies, we could do that by isolating each operation into a dedicated service and redirecting their inputs/outputs to perform our original local query as a distributed one. In that case, despite the architectural change, the logic of the query would remain intact.

The Running Example written in Jolie/Tquery

We conclude this section by presenting the implementation of our running example from 'Overview and Running Example', Listing 2. Specifically, we present two alternatives: a more faithful one in Fig. 3, where we have a one-to-one correspondence between Tquery operators and Jolie/Tquery operations, and one in Fig. 4 that obtains the same result by using Jolie/Tquery pipelines.

While the code in Fig. 3 fulfills the promise made in 'Overview and Running Example' to show the implementation of the example in Listing 2, we take the chance to illustrate, in Fig. 4, how one can transition between a composition of single-stage queries to multi-stage, pipelined ones. Moreover, Fig. 4 is a reference for the actual Jolie/Tquery code used in 'Benchmarks' to benchmark our implementation.

Translating Tquery operator calls into Jolie/Tquery ones is straightforward, *e.g.*, the match at Line 3 of Listing 2 corresponds to Lines 4–11 of Fig. 3. As expected, the main

```
getPatientPseudoID@HospitalIT(pData)(pID)
getMotionAndTemperature@SmartWatch(creds)(_tmp)

match@Tquery({
 data << _tmp
 query.or << {
  left <<   { equal << { path = "date" data = 20201128 } }
  right.or << {
   left <<  { equal << { path = "date" data = 20201129 } }
   right << { equal << { path = "date" data = 20201130 } }
}}})( resp )

group@Tquery({
 data << resp.result
 query.aggregate << { dstPath = "t" srcPath = "t" }
})( resp )

project@Tquery({
 data << resp.result
 query[0] << { dstPath = "t" value.path = "t" }
 query[1] << { dstPath = "patient_id" value = pseudoId }
})( tmp )

detectFever@HospitalIT( tmp )( hasFever )

if( hasFever ){
 getSleepPatterns@SmartPhone( credentials )( _sl )

 unwind@Tquery( { data << _sl query = "M.D.L" } )( resp )

 project@Tquery( {
  data << resp.result
  query[0] << {dstPath = "year"    value.path = "y"       }
  query[1] << {dstPath = "month"   value.path = "M.m"     }
  query[2] << {dstPath = "day"     value.path = "M.D.d"   }
  query[3] << {dstPath = "quality" value.path = "M.D.L.q" }
 })( resp )

 match@Tquery( {
  data << resp.result
  query.and << {
   left.equal    << { path = "year"  data = 2020 }
   right.and << {
    left.equal   << { path = "month" data = 11 }
    right.or << {
     left.equal  << { path = "day"   data = 29 }
     right.equal << { path = "day"   data = 30 }
 }}}})( resp )

 group@Tquery( {
  data << resp.result
  query.aggregate <<{dstPath="quality" srcPath="quality"}
 })( resp )

 project@Tquery( {
  data << resp.result
  query[0] << {dstPath = "quality" value.path = "quality"}
  query[1] << {dstPath = "patient_id" value = pseudoId}
 })( sl )

 lookup@Tquery( {
  leftData  << sl.result   leftPath  = "patient_id"
  rightData << tmp.result  rightPath = "patient_id"
  dstPath = "temperatures"
 })( resp )

 project@Tquery( {
  data << resp.result
  query[0] <<
   {dstPath="quality"      value.path = "quality"}
  query[1]<<
   {dstPath="temperatures" value.path = "temperatures.t"}
  query[2] <<
   {dstPath="patient_id"   value.path = "patient_id"}
 })( bs )

 detectEncephalopathy@HospitalIT( bs )
}
```

Figure 3 **Single-stage implementation of Listing 2.**

Full-size DOI: 10.7717/peerjcs.1037/fig-3

```
getPatientPseudoID@HospitalIT(pData)(pID)
getMotionAndTemperature@SmartWatch(creds)(_tmp)

pt[i++] << { matchQuery.or << {
 left.equal   << { path = "date" data = 20201128 }
 right.or << {
  left.equal << { path = "date" data = 20201129 }
  right.equal  << { path = "date" data = 20201130 }
 }
}

pt[i++] << {
 groupQuery.aggregate << { dstPath = "t" srcPath = "t" }
}

pt[i++] << {
 projectQuery[0] << { dstPath = "t" value.path = "t" }
 projectQuery[1] <<
  { dstPath = "patient_id" value = pseudoId }
}

pipeline@Tquery({data<<_tmp pipeline<<pt})(tmp)

detectFever@HospitalIT( tmp )( hasFever )

if( hasFever ){
 getSleepPatterns@SmartPhone( credentials )( _sl )

 ps[j].unwindQuery = "M.D.L"

 ps[j++] << { projectQuery[0] <<
   { dstPath = "year" value.path = "y" }
  projectQuery[1] <<
   { dstPath = "month" value.path = "M.m" }
  projectQuery[2] <<
   { dstPath = "day" value.path = "M.D.d" }
  projectQuery[3] <<
   { dstPath = "quality" value.path = "M.D.L.q" }
}

 ps[j++] << { matchQuery.and << {
  left.equal    << { path = "year"  data = 2020 }
  right.and << {
   left.equal   << { path = "month" data = 11 }
   right.or << {
    left.equal  << { path = "day"   data = 29 }
    right.equal << { path = "day"   data = 30 }
 }}}}

 ps[j++].groupQuery.aggregate <<
  { dstPath = "quality" srcPath = "quality" }

 ps[j++] << {
  projectQuery[0] <<
   { dstPath = "quality" value.path = "quality" }
  projectQuery[1] <<
   { dstPath = "patient_id" value = pseudoId }
 }

 ps[j++].lookupQuery << {
  rightData << tmp.result  rightPath = "patient_id"
  leftPath = "patient_id"
  dstPath = "temperatures"
 }

 ps[j++] << {
  projectQuery[0] <<
   { dstPath = "quality" value.path = "quality" }
  projectQuery[1] <<
   {dstPath="temperatures" value.path="temperatures.t"}
  projectQuery[2] <<
   { dstPath = "patient_id" value.path = "patient_id" }
 }

 pipeline@Tquery({ data << _sl pipeline << ps })( bs )

 detectEncephalopathy@HospitalIT( bs )
}
```

Figure 4 **Multi-stage implementation of Listing 2.**

Full-size DOI: 10.7717/peerjcs.1037/fig-4

difference is that we need to map the elements of the criterion φ from Line 3 of Listing 2 into a Jolie tree that follows the shape of type φ (cf. Lines 11–19 of Fig. 2).

The reuse of the `type`s of the single-stage operators in the definition of the `pipeline` helps migrating between the two modalities. For example, at Lines 4–10 of Fig. 4, we find that the definition of the match stage under the `pt` data structure follows the one at Lines 4–11 of Fig. 3.

We finally show how our implementation can interact with different services and heterogeneous data sources. In particular, we assume that the service offered by the hospital communicates XML messages over HTTP, and that smart-watches instead use an efficient binary protocol—SODEP (*Montesi, Guidi & Zavattaro, 2014*). These assumptions are coded in Jolie for our example with appropriate `outputPort`s that allow our implementation to contact these other components by using the right transports and data formats, as follows (we parameterise our code on the locations of these components, which are provided externally).

```
1 outputPort HospitalIT {
2   location: params.hospitalLocation
3   protocol: http { format = "xml" }
4   interfaces: HospitalInterface
5 }
6
7 outputPort SmartWatch {
8   location: params.smartWatchLocation
9   protocol: sodep
10  interfaces: SmartWatchInterface
11 }
```

Listing 5: Collecting data from heterogeneous sources.

The rest of our implementation is modular to these details: changing locations, protocols, or data formats does not require changing the code shown in Figs. 3 and 4.

BENCHMARKS

We now present the method we followed to benchmark Jolie/Tquery and our experimental results. Specifically, we concentrate on the main application scenario of Tquery, *i.e.*, that of ephemeral data-handling, exemplified in 'Overview and Running Example' with the query logic presented in Listing 2. In 'The Running Example written in Jolie/Tquery' we showed two possible concrete realisations of the logic in Listing 2, developed using Jolie/Tquery. Here, we use Listing 2 as use case for our benchmarks and, as motivated below, the pipeline Jolie/Tquery realisation of Listing 2 (from Fig. 4), as the reference implementation to run our experiments.

To obtain a baseline against which to contrast the performance of Jolie/Tquery, we develop an alternative implementation of the example at 'Overview and Running Example' that uses MongoDB. This alternative implementation is the closest we can obtain to the logic expressed in 'Overview and Running Example', since *i*) the MongoDB query language (*MongoDB Inc., 2022*) inspired (*via* (*Botoeva et al., 2018*)) the design of Tquery and *ii*) the former supports a superset of the operators of the latter. As a confirmation of this fact, we implemented the logic of Listing 2 using MongoDB as a

sequence of two, multi-stage queries, issued through the "aggregate" MongoDB API (https://docs.mongodb.com/manual/aggregation/). The resulting implementation follows the same invocation pattern as the one presented in 'The Running Example written in Jolie/Tquery', which uses the pipeline API extension of Jolie/Tquery, thus, motivating our choice to use this variant.

We remark that MongoDB provides an "in-memory" modality that avoids the overhead of making the data persistent on disk. Using this modality would likely give us baseline values closer to the in-memory performance profile of Tquery. Unfortunately, this modality is accessible only through the paid MongoDB Enterprise Advanced Subscription. Since using a paid-only feature would hinder the reproducibility of our experiments, we do not consider it. Here, we consider three configurations for MongoDB. First, the default one, tailored for persistency, that writes logs of transactions and data on disk. The second one is the MongoDB in the "no journal" modality, which avoids to write a log of the transactions on disk. The third one is an ephemeral configuration taken from grey literature (*Girbal, 2021*) that combines the "no journal" modality with the usage of a `tmpfs` (*Snyder, 1990*) disk as the one where MongoDB stores its data, to avoid the latencies of writing on non-volatile storage.

Below, we report the respective performance of the four configurations—one for Jolie/Tquery and three for MongoDB—in terms of the delay between when the engine receives a request and when it is ready to send back the response. Hence, we avoid recording the time spent transmitting the data between the invoker and the data-handling engine, which is orthogonal to the engine's performance.

To run our benchmarks, we developed two Jolie microservices: one, called `TqueryService`, which contains the implementation in Fig. 4 and the other, called `MongoService`, which implements the following behaviour: (i) insert the data in MongoDB, (ii) perform the queries through MongoDB, and (iii) drop the data from MongoDB, to ensure ephemerality. When recording the performance of `MongoService`, we include the deletion (drop) time, before issuing back the response. To let `MongoService` and MongoDB interact, we use the synchronous version of the MongoDB Java Drivers[6] and we implement its behaviour in Java, similarly as done in 'The implementation of Jolie/Tquery' for Jolie/Tquery.

We synthetically generate 5 tiers of data for the benchmarks. Specifically, we generate 5 pairs of JSON files, each including one file for the temperatures and one for the sleep logs, following the structures from Listing 1. Each tier covers one year of recordings and it includes a number of samplings per day that doubles from a tier to the next: for the temperatures, the first tier contains one sampling per minute (1440 samplings per day), the second contains two samplings per minute (2880), and so on; for the sleep logs, the first tier contains 16 samplings per day, the second contains 32, and so on.

Our benchmark architecture includes a third Jolie microservice, called `DataLoader`, which we use to implement the high-level benchmark logic reported in Algorithm 1. Essentially, given the number of invocations to perform (*min_total_calls*), the number of requests in a batch (*batch_size*), and the set of data tiers (*tiers*), the service sends a sequence of *min_total_calls*/*batch_size* batches (rounded up to the next largest integer, to make sure

[6] Through the `jolie-mongodb-driver` library, available at https://github.com/szingaro/jmdb, which uses the MongoDB synchronous Java library, available at https://docs.mongodb.com/drivers/java/sync/current/.

to issue at least *min_total_calls* invocations). In Algorithm 1, the call *invokeTargetService* performs, in parallel, as many queries as indicated by the *batch_size*, where "*Target*" is one of the four configurations of our benchmark.

Algorithm 1: The `DataLoader` service logic.

```
Input: min_total_calls, batch_size, tiers
begin
    for tier in tiers do
        for batch ← 0 to ceil( min_total_calls / batch_size ) do
            invokeTargetService ( tier, batch_size )
        end
    end
end
```

We execute our benchmarks on a machine equipped with an Intel Xeon Silver 4208 CPU @ 2.10 GHz (32 CPUs), 96GB RAM, and a Dell FH49G SSD. The machine runs CentOS 7 (Kernel 3.10.0 x86_64), Java 11 (with maximal heap size of 32GB), Jolie 1.10.5, Jolie/Tquery 0.4.10, the MongoDB Synchronous Driver 4.2.3, and MongoDB Community Server 4.4.6.

We report in Fig. 5 our benchmarks of Jolie/Tquery and MongoDB, aggregated per batch size (from the top-left corner, for 5, 8, 10, 12, 15, and 20 parallel requests): each plot represents the relation between the data-tier size and the average execution time, maintaining constant the number of parallel invocations. The experimental results show that Jolie/Tquery performs consistently faster than MongoDB (all configurations). Since in the test cases with MongoDB we record the request-to-response delay of the database, the higher execution times of these cases correspond to both the overhead of the communication and the possible bottlenecks due to establishing connections to it. We notice a slight decrease in the relative distance between Jolie/Tquery and MongoDB at the increase of batch and data-tier sizes (in particular, the fourth and fifth tiers and the 15- and 20-sized batches). Our intuition of the phenomenon is that, on the given machine, when exceeding those thresholds, the Jolie execution runtime and the Jolie/Tquery engine undergo overhead due to resource contention. As expected, the default configuration of MongoDB is the one that performs the worst. The other two configurations ("no journal" and "no journal in-memory") perform slightly better than the default and the difference between them is negligible—our intuition is that writing on disk is the driving factor that determines the drop in performance.

For completeness, we report in Fig. 6 the benchmarks aggregated by engine, which confirm the observations above: Jolie/Tquery consistently outperforms MongoDB over the different batches, where the degree of parallelism and the size of data are the main factors that determine changes in the performance trend.

Besides the direct results commented above, the performance behaviour plotted in Fig. 5 and Fig. 6 indicate that, when reaching some empirical threshold values, the system

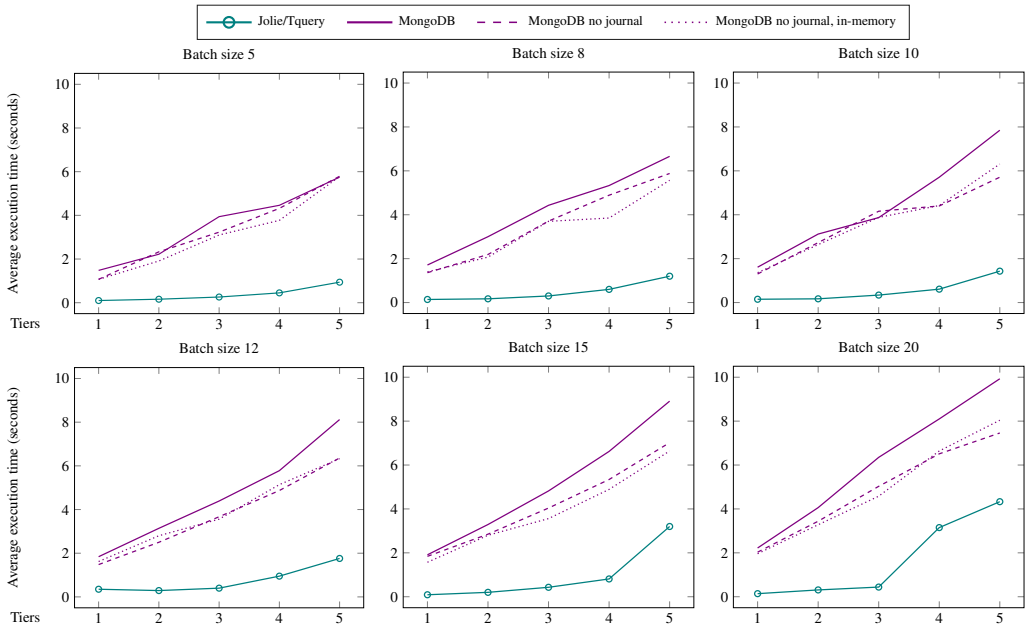

Figure 5 **Batch-wise benchmarks for Jolie/Tquery and MongoDB.**
Full-size DOI: 10.7717/peerjcs.1037/fig-5

would benefit from scaling-up, either by distributing the query over multiple nodes or by having multiple copies of the same service and balancing the requests. Here, the flexibility of Jolie/Tquery can help the user to attain those configurations by minimising the footprint of the migration on both the system (no need to deploy additional database instances) and the codebase (cf. 'Extending Jolie/Tquery with query pipelines').

DISCUSSION AND CONCLUSION

In this article, we presented Tquery, which is a theory for querying semi-structured data, compatible with Jolie. While Tquery is a formal model for general reference, we also presented Jolie/Tquery, which we showed to be especially suitable in the context of ephemeral data-handling. However, Jolie/Tquery is useful in general, for example in big-data analytics scenarios, where developers can specifies queries in a single node and then easily distribute it over different nodes.

Looking at future extensions, a natural evolution of this work is to perform a more complete evaluation of the expressivity of Jolie/Tquery by implementing well-known data-flow patterns (*Hohpe & Woolf, 2004*). A useful by-product of that endevour is the collection of a library of data-flow patterns implemented in Jolie/Tquery, available to developers. A complementary contribution to the above proposal is to perform an exhaustive study and benchmarking of the technologies for ephemeral data-handling. In that work, we would start by collecting real-world use cases of ephemeral data-handling and by selecting the most representative ones into a library of test scenarios. Then, we would collect the main tools used in ephemeral data-handling contexts (including Jolie/Tquery) and compare them from the different points of view of the features they have and their efficiency (*e.g.*, in

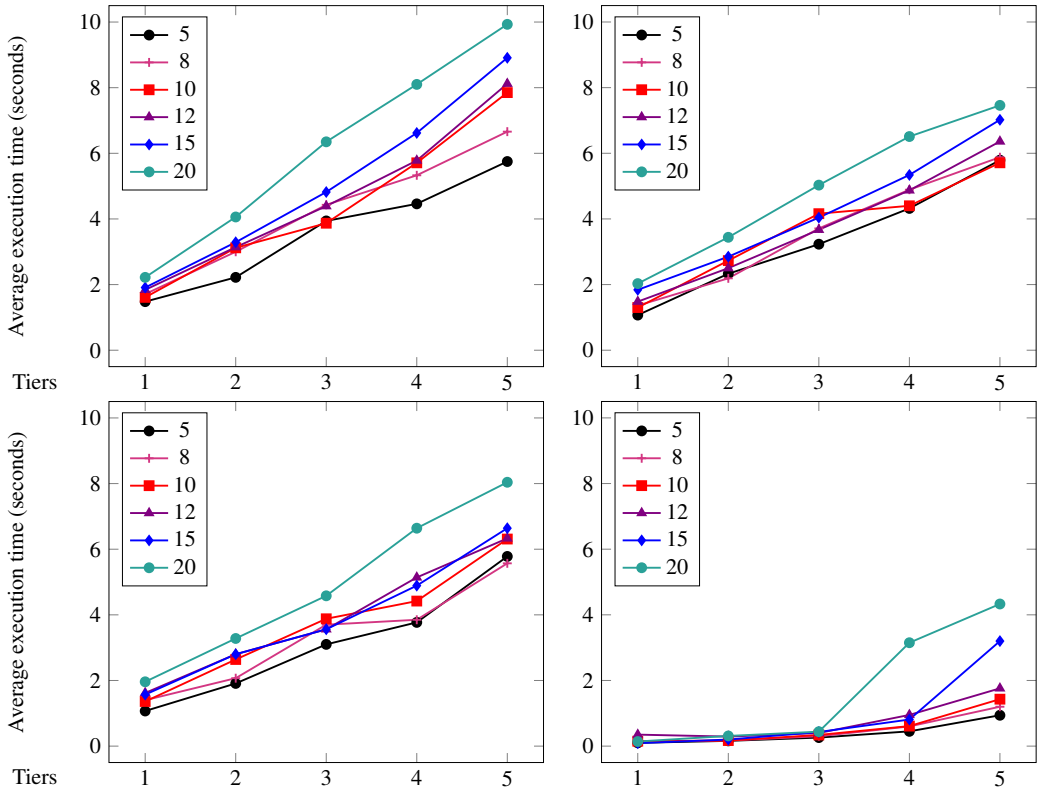

Figure 6 Engine-wise benchmarks of MongoDB (top-left), MongoDB without journaling (top-right), MongoDB without journaling and in-memory (bottom-left), and Jolie/Tquery (bottom-right). The lines represent the different batches of requests.

Full-size DOI: 10.7717/peerjcs.1037/fig-6

terms of program size) and performance as obtained through the implementation of our library of tests.

Another direction is widening the scope of application of Jolie/Tquery with case studies and experiments where data queries are performed by low-power devices in IoT environments. This would entail building topologies of nodes with different tasks—e.g., gatherers (*e.g.*, edge devices equipped with sensors), collectors (*e.g.*, fog nodes that use Jolie/Tquery to aggregate and forward the gathered data to more powerful nodes), and crunchers (*e.g.*, cloud nodes where Jolie/Tquery would manage the high amount of data coming from the edge and fog layers)—and benchmarking their performance (possibly in comparison with alternative technologies for ephemeral data handling). Querying data on devices with low power and memory would likely require implementing strategies for distributing Jolie/Tquery pipelines over networks; future work in this direction will be able to benefit from the native support for services in heterogeneous environments offered by Jolie, which was another reason for developing a querying framework for Jolie.

We think that the above studies, besides providing us with the necessary material to guide us in evolving Jolie/Tquery—e.g., indicating the need for the inclusion of new operators—,

would generate useful references for researchers to orient themselves in the growing field of ephemeral data-handling.

While studying the Tquery operators, we noticed and reported on how the shape of the data impacts on the possibility to distribute the stages of the query pipeline. To the best of our knowledge, this is a design space that did not receive a lot of attention in the literature and, yet, we deem it fundamental to provide further means for improving the performance of ephemeral data-handling systems. Here, our intuition is that Jolie types can help in providing a model that we can use to reason on the shape of the data and their interplay with the operators in a given query. Possible outcomes of this study include giving guidelines to developers to maximise the flexibility of their queries, as well as implementing tools that automatise the optimal distribution of query pipelines.

Finally, since Jolie/Tquery come as a library for the Jolie language, by implementing the support for new data formats in Jolie we would make them automatically available for Jolie/Tquery users.

ACKNOWLEDGEMENTS

The authors thank Claudio Guidi and Balint Maschio for useful discussions on the practical motivation of our framework.

ADDITIONAL INFORMATION AND DECLARATIONS

Funding

This work was sponsored by Villum Fonden, grant no. 29518, by Independent Research Fund Denmark, grant no. 0135-00219, and by Horizon2020, grant no. 825619. The funders had no role in study design, data collection and analysis, decision to publish, or preparation of the manuscript.

Grant Disclosures

The following grant information was disclosed by the authors:
Villum Fonden: 29518.
Independent Research Fund Denmark: 0135-00219.
Horizon2020: 825619.

Competing Interests

The authors declare there are no competing interests.

Author Contributions

- Saverio Giallorenzo conceived and designed the experiments, performed the experiments, analyzed the data, performed the computation work, prepared figures and/or tables, authored or reviewed drafts of the article, and approved the final draft.
- Fabrizio Montesi conceived and designed the experiments, performed the experiments, analyzed the data, performed the computation work, authored or reviewed drafts of the article, and approved the final draft.

- Larisa Safina conceived and designed the experiments, performed the experiments, analyzed the data, performed the computation work, prepared figures and/or tables, authored or reviewed drafts of the article, and approved the final draft.
- Stefano Pio Zingaro conceived and designed the experiments, performed the experiments, analyzed the data, performed the computation work, prepared figures and/or tables, authored or reviewed drafts of the article, and approved the final draft.

Data Availability

The following information was supplied regarding data availability:

The code (test data is included in the tests directory) is available at GitHub: Available at https://github.com/jolie/tquery.

Supplemental Information

Supplemental information for this article can be found online at http://dx.doi.org/10.7717/peerj-cs.1037#supplemental-information.

REFERENCES

Apache. 2005. Apache CouchDB. *Available at https://couchdb.apache.org*.

Apache. 2022a. Apache Flink. *Available at https://flink.apache.org*.

Apache. 2022b. Apache Samza. *Available at https://samza.apache.org*.

Apache. 2022c. Apache Storm. *Available at https://storm.apache.org*.

Arasu A, Babcock B, Babu S, Cieslewicz J, Datar M, Ito K, Motwani R, Srivastava U, Widom J. 2016. STREAM: the stanford data stream management system. In: Garofalakis M, Gehrke J, Rastogi R, eds. *Data stream management: processing high-speed data streams*. Berlin, Heidelberg: Springer Berlin Heidelberg, 317–336 DOI 10.1007/978-3-540-28608-0_16.

Arasu A, Babu S, Widom J. 2006. The CQL continuous query language: semantic foundations and query execution. *The VLDB Journal* **15(2)**:121–142 DOI 10.1007/s00778-004-0147-z.

Armbrust M, Fox A, Griffith R, Joseph AD, Katz R, Konwinski A, Lee G, Patterson D, Rabkin A, Stoica I. 2010. A view of cloud computing. *Communications of the ACM* **53(4)**:50–58.

Babcock B, Babu S, Datar M, Motwani R, Widom J. 2002. Models and issues in data stream systems. In: *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on principles of database systems, PODS '02*. New York, NY, USA: Association for Computing Machinery, 1–16 DOI 10.1145/543613.543615.

Babu S, Widom J. 2001. Continuous queries over data streams. *SIGMOD Record* **30(3)**:109–120 DOI 10.1145/603867.603884.

Baker SB, Xiang W, Atkinson I. 2017. Internet of Things for smart Healthcare: technologies, challenges, and opportunities. *IEEE Access* **5**:26521–26544 DOI 10.1109/ACCESS.2017.2775180.

Barbieri DF, Braga D, Ceri S, Della Valle E, Grossniklaus M. 2009. C-SPARQL: sPARQL for continuous querying. In: *Proceedings of the 18th international conference on world*

wide web, WWW '09. New York, NY, USA: Association for Computing Machinery, 1061–1062 DOI 10.1145/1526709.1526856.

Botoeva E, Calvanese D, Cogrel B, Rezk M, Xiao G. 2016. A formal presentation of MongoDB (Extended Version), CoRR. ArXiv preprint. arXiv:1603.09291.

Botoeva E, Calvanese D, Cogrel B, Xiao G. 2018. Expressivity and complexity of MongoDB queries. In: Kimelfeld B, Amsterdamer Y, eds. *21st International conference on database theory, ICDT 2018, March 26-29, 2018, Vienna, Austria, volume 98 of LIPIcs*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 9:1–9:23 DOI 10.4230/LIPIcs.ICDT.2018.9.

Bray T, Paoli J, Sperberg-McQueen CM, Maler E, Yergeau F. 2000. Extensible markup language (XML) 1.0, W3C recommendation October. Cambridge, MA, USA: W3C.

Brian Krebs. 2017. Extortionists wipe thousands of databases, victims who pay up get stiffed. *Available at https://krebsonsecurity.com/2017/01/extortionists-wipe-thousands-of-databases-victims-who-pay-up-get-stiffed*.

Bunn JA, Navalta JW, Fountaine CJ, Reece JD. 2018. Current state of commercial wearable technology in physical activity monitoring 2015-2017. *International Journal of Exercise Science* **11(7)**:503–515.

Callegati F, Gabbrielli M, Giallorenzo S., Melis A, Prandini M. 2017. Smart mobility for all: a global federated market for mobility-as-a-service operators. In: *20th IEEE international conference on intelligent transportation systems, ITSC 2017, Yokohama, Japan, October 16-19, 2017*. Piscataway: IEEE, 1–8 DOI 10.1109/itsc.2017.8317701.

Caspi P, Pilaud D, Halbwachs N, Plaice JA. 1987. LUSTRE: a declarative language for real-time programming. In: *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on principles of programming languages, POPL '87*. New York, NY, USA: Association for Computing Machinery, 178–188 DOI 10.1145/41625.41641.

Chen J, DeWitt DJ, Tian F, Wang Y. 2000. NiagaraCQ: a scalable continuous query system for internet databases. In: *Proceedings of the 2000 ACM SIGMOD international conference on management of data, SIGMOD '00*. New York, NY, USA: Association for Computing Machinery, 379–390 DOI 10.1145/342009.335432.

Cheney J, Lindley S, Wadler P. 2013. A practical theory of language-integrated query. *ACM SIGPLAN Notices* **48(9)**:403–416.

Crockford D. 2006. The application/json media type for javascript object notation (json). *Available at http://www.ietf.org/rfc/rfc4627.txt*.

Diao Y, Fischer P, Franklin M, To R. 2002. YFilter: efficient and scalable filtering of XML documents. In: *Proceedings 18th international conference on data engineering*. 341–342 DOI 10.1109/ICDE.2002.994748.

Dragoni N, Giallorenzo S, Lluch-Lafuente A, Mazzara M, Montesi F, Mustafin R, Safina L. 2017. Microservices: yesterday, today, and tomorrow. In: *Present and ulterior software engineering*. Berlin, Germany: Springer, 195–216 DOI 10.1007/978-3-319-67425-4_12.

Elasticsearch. 2022. Elasticsearch event query language. *Available at https://www.elastic.co/blog/introducing-event-query-language*.

Ellis T. 2014. Opaleye. *Available at https://github.com/tomjaguarpaw/haskell-opaleye*.

Esteves S, Janssens N, Theeten B, Veiga L. 2017. Empowering stream processing through edge clouds. *SIGMOD Rec.* **46(3)**:23–28 DOI 10.1145/3156655.3156661.

Fussel M. 1997. Foundations of object-relational mapping. *Available at http://markfussell.emenar.com/blog/object-relational/*.

Gabbrielli M, Giallorenzo S, Lanese I, Zingaro SP. 2018. A language-based approach for interoperability of IoT platforms. *Available at https://scholarspace.manoa.hawaii.edu/server/api/core/bitstreams/573255ff-bc3a-4928-9f5b-3809a37745c3/content*.

Gabbrielli M, Giallorenzo S, Lanese I, Zingaro SP. 2019. Linguistic abstractions for interoperability of IoT platforms. In: Majchrzak T, Mateos C, Poggi F, Grønli TM, eds. *Towards integrated web, mobile, and IoT technology*. *Lecture notes in business information processing*, vol. 347. Cham: Springer DOI 10.1007/978-3-030-28430-5_5.

Gabbrielli M, Giallorenzo S, Montesi F. 2014. Service-oriented architectures: from design to production exploiting workflow patterns. In: Omatu S, Bersini H, Corchado J, Rodríguez S, Pawlewski P, Bucciarelli E, eds. *Distributed computing and artificial intelligence, 11th international conference. Advances in intelligent systems and computing, vol 290*. Cham: Springer, DOI 10.1007/978-3-319-07593-8_17.

Giallorenzo S, Montesi F, Peressotti M, Rademacher F, Sachweh S. 2021. Jolie and LEMMA: Model-Driven Engineering and Programming Languages Meet on Microservices. In: Damiani F, Dardha O, eds. *Coordination models and languages. COORDINATION 2021. Lecture notes in computer science(), vol 12717*. Cham: Springer, DOI 10.1007/978-3-030-78142-2_17.

Giallorenzo S, Montesi F, Safina L, Zingaro SP. 2019. Ephemeral data handling in microservices. In: Bertino E, Chang CK, Chen P, Damiani E, Goul M, Oyama K, eds. *2019 IEEE international conference on services computing, SCC 2019, Milan, Italy, July 8-13, 2019*. Piscataway: IEEE, 234–236 DOI 10.1109/SCC.2019.00048.

Girbal A. 2021. How to use MongoDB as a pure in-memory DB. *Available at https://edgystuff.tumblr.com/post/49304254688/how-to-use-mongodb-as-a-pure-in-memory-db-redis* (accessed on 20 September 2021).

Hirten RP, Danieletto M, Tomalin L, Choi KH, Zweig M, Golden E, Kaur S, Helmus D, Biello A, Pyzik R. 2020. Longitudinal physiological data from a wearable device identifies SARS-CoV-2 infection and symptoms and predicts COVID-19 diagnosis. *MedRxiv* DOI 10.1101/2020.11.06.20226803.

Hirzel M, Schneider S, Gedik B. 2017. SPL: an extensible language for distributed stream processing. *ACM Transactions on Programming Languages and Systems* **39(1)**:5 DOI 10.1145/3039207.

Hohpe G, Woolf B. 2004. *Enterprise integration patterns: designing, building, and deploying messaging solutions*. Boston: Addison-Wesley Professional.

Jang M. 2006. *Linux annoyances for geeks: getting the most flexible system in the world just the way you want it*. Sebastopol, CA, USA: O'Reilly Media.

Kong L, Mamouras K. 2020. StreamQL: a query language for processing streaming time series. *Proceedings of the ACM on Programming Languages* **183**:1–32 DOI 10.1145/3428251.

Leavitt N. 2010. Will NoSQL databases live up to their promise? *Computer* **43**:12–14 DOI 10.1109/mc.2010.58.

Ma M, Wang P, Chu C-H. 2013. Data management for internet of things: challenges, approaches and opportunities. In: *2013 IEEE International conference on green computing and communications and IEEE Internet of Things and IEEE cyber, physical and social computing*. Piscataway: IEEE, 1144–1151.

Maschio B. 2017. The use of microservices to implement cross process integration and data sharing. *Available at https://www.conf-micro.services/2017/papers/Maschio.pdf* .

Maschio B. 2019. Updating the current Jolie microservices based Document Management System to include electronic invoicing. In: *Proceedings of the 2th international conference on microservices*. *Available at https://www.conf-micro.services/2019/papers/Microservices_2019_paper_15.pdf* .

Meijer E, Beckman B, Bierman G. 2006. Linq: reconciling object, relations and xml in the. net framework. In: *Sigmod*. 706–706.

Mendell M, Nasgaard H, Bouillet E, Hirzel M, Gedik B. 2012. Extending a general-purpose streaming system for XML. In: *Proceedings of the 15th international conference on extending database technology*. 534–539.

MongoDB Inc. 2018a. MongoDB aggregation framework. *Available at https://www.mongodb.com/developer/products/mongodb/aggregation-framework/*.

MongoDB Inc. 2018b. MongoDB website. *Available at https://www.mongodb.com/*.

MongoDB Inc. 2022. Aggregation pipeline operators in MongoDB. *Available at https://docs.mongodb.com/manual/reference/operator/aggregation/*.

Montesi F. 2016. Process-aware web programming with Jolie. *Science of Computer Programming* **130**:69–96 DOI 10.1016/j.scico.2016.05.002.

Montesi F, Guidi C, Zavattaro G. 2014. Service-oriented programming with Jolie. In: Bouguettaya A, Sheng Q, Daniel F, eds. *Web services foundations*. New York: Springer DOI 10.1007/978-1-4614-7518-7_4.

Mostert M, Bredenoord AL, Biesaart MC, Van Delden JJ. 2016. Big Data in medical research and EU data protection law: challenges to the consent or anonymise approach. *European Journal of Human Genetics* **24(7)**:956–960 DOI 10.1038/ejhg.2015.239.

Narkhede N. 2017. Introducing KSQL: streaming SQL for Apache Kafka. *Available at https://www.confluent.io/blog/ksql-streaming-sql-for-apache-kafka/*.

Oram A. 2019. *Ballerina: a language for network-distributed applications*. Sebastopol, CA, USA: O'Reilly Media, Incorporated.

Pierce BC. 2002. *Types and programming languages*. Cambridge: MIT Press.

Purohit B, Kumar A, Mahato K, Chandra P. 2020. Smartphone-assisted personalized diagnostic devices and wearable sensors. *Current Opinion in Biomedical Engineering* **13**:42–50 DOI 10.1016/j.cobme.2019.08.015.

Reda R, Piccinini F, Carbonaro A. 2018. Towards consistent data representation in the IoT healthcare landscape. In: *DH '18: Proceedings of the 2018 international conference on digital health*. DOI 10.1145/3194658.3194668.

Ron A, Shulman-Peleg A, Puzanov A. 2016. Analysis and mitigation of NoSQL injections. *IEEE Security & Privacy* **14(2)**:30–39 DOI 10.1109/MSP.2016.36.

Rose N. 2014. The human brain project: social and ethical challenges. *Neuron* **82(6)**:1212–1215 DOI 10.1016/j.neuron.2014.06.001.

Safina L, Mazzara M, Montesi F, Rivera V. 2016. Data-driven workflows for microservices: genericity in Jolie. In: Barolli L, Takizawa M, Enokido T, Jara AJ, Bocchi Y, eds. *30th IEEE international conference on advanced information networking and applications, AINA 2016, Crans-Montana, Switzerland, 23-25 March, 2016*. Piscataway: IEEE Computer Society, 430–437 DOI 10.1109/aina.2016.95.

Shein E. 2013. Ephemeral Data. *Communications of the ACM* **56(9)**:20–22.

Shi W, Cao J, Zhang Q, Li Y, Xu L. 2016. Edge computing: vision and challenges. *IEEE Internet of Things Journal* **3(5)**:637–646 DOI 10.1109/JIOT.2016.2579198.

Siddhi. 2022. Siddhi Streaming SQL. *Available at https://siddhi.io/en/v4.x/docs/query-guide/*.

Snyder P. 1990. tmpfs: a virtual memory file system. In: *Proceedings of the autumn 1990 European UNIX Users' group conference*. 241–248.

Thurman SM, Wasylyshyn N, Roy H, Lieberman G, Garcia JO, Asturias A, Okafor GN, Elliott JC, Giesbrecht B, Grafton ST, Mednick SC, Vettel JM. 2018. Individual differences in compliance and agreement for sleep logs and wrist actigraphy: a longitudinal study of naturalistic sleep in healthy adults. *PLOS ONE* **13(1)**:e0191883 DOI 10.1371/journal.pone.0191883.

Tommasini R, Sakr S, Balduini M, Valle ED. 2019. An outlook to declarative languages for big steaming data. In: *Proceedings of the 13th ACM international conference on distributed and event-based systems, DEBS '19*. New York, NY, USA: Association for Computing Machinery, 199–202 DOI 10.1145/3328905.3332462.

Van Alsenoy B. 2019. General data protection regulation. In: *Data Protection Law in the EU: roles, responsibilities and liability*. Cambridge: Intersentia, 279–324.

Vigevano F, Liso PD. 2018. Chapter 11 - differential diagnosis. In: *Acute encephalopathy and encephalitis in infancy and its related disorders*. Amsterdam: Elsevier, 81–85.

Visveswaran S. 2000. Dive into connection pooling with J2EE. *Available at https://www.infoworld.com/article/2076221/dive-into-connection-pooling-with-j2ee.html*.

W3c. 2001. Transport message exchange pattern: single-Request-Response. *Available at https://www.w3.org/2000/xp/Group/1/10/11/2001-10-11-SRR-Transport_MEP*.

WSO2. 2022. WSO2 stream processor. *Available at https://wso2.com/integration/streaming-integrator/*.