An Architecture for the Development of Distributed Analytics based on Polystore Events

Athanasios Zolotas[1], Konstantinos Barmpis[1], Fady Medhat[1], Patrick Neubauer[1], Dimitris Kolovos[1], and Richard F. Paige[1,2]

[1] Department of Computer Science, University of York, York, United Kingdom
{thanos.zolotas, konstantinos.barmpis, fady.medhat, patrick.neubauer, dimitris.kolovos, richard.paige}@york.ac.uk
[2] Department of Computer Science, McMaster University, Hamilton, Canada

Abstract. To balance the requirements for data consistency and availability, organisations increasingly migrate towards hybrid data persistence architectures (called *polystores* throughout this paper) comprising both relational and NoSQL databases. The EC-funded H2020 TYPHON project offers facilities for designing and deploying such polystores, otherwise a complex, technically challenging and error-prone task. In addition, it is nowadays increasingly important for organisations to be able to extract business intelligence by monitoring data stored in polystores. In this paper, we propose a novel approach that facilitates the extraction of analytics in a distributed manner by monitoring *polystore queries* as these arrive for execution. Beyond the analytics architecture, we presented a pre-execution authorisation mechanism. We also report on preliminary scalability evaluation experiments which demonstrate the linear scalability of the proposed architecture.

Keywords: Analytics · Hybrid Databases · Polystores · Queries

1 Introduction

Data managed within an organisation may have significantly variable consistency and availability requirements. For example, in the case of an e-commerce system, data used to provide recommendations of products to users needs to be highly available but the consistency of such data is not critical. By contrast, for other subsets of data, such as data recording customer payments, compromising data consistency to improve availability is not acceptable. As a result, organisations increasingly need to use both relational and non-relational databases.

Nowdays, small businesses to big coorporations use monitoring tools and data analytics to extract business intelligence based on data stored in such hybrid database systems. This can lead to improvement on their systems and business processes enhancing the customer experience. For example, in an e-commerce system retailers often need to identify relationships of interest between products they trade to provide useful recommendations to customers. Such knowledge can be extracted by including analytics logic within the application business logic to store into the database information of interest.

We propose in this work another approach, that of monitoring the polystore queries and extract analytics of interest as queries arrive for execution. This approach comes with the benefit of calculating analytics in real-time without the need of mixing analytics with core business logic. It also offers access to data that may never be stored in a database (e.g., from "select" queries) or data that were later deleted or updated. Consider the following two motivating scenarios in the domain of an e-commerce website. Analytics developers are able to monitor "trending products" by monitoring the number of "select" queries arriving to the database for each product by the web application when users browse to the details page of each product. In a modern large-scale system not all user requests would end up in the database - given that there are HTTP-level caches commonly in place in such systems. In addition, developers can identify products that users *almost* bought by checking pairs of insert and delete queries to each user's basket for the same product.

In this paper we propose an architecture that consumes queries on polystores to facilitate orthogonal real-time monitoring and predictive analytics. To accommodate the large number of events that polystore-backed applications are expected to generate in real-world scenarios, the proposed architecture is implemented on top of proven big-data-capable frameworks such as Apache Flink [5] and Apache Kafka [4]. Flink is used for distributing the processing/execution workload of analytics applications while Kafka stores and dispatches the generated events in a form of a distributed log. Beyond the possibility of producing analytics based on queries, the analytics component also offers a mechanism of blocking the execution of commands that do not meet developer-defined criteria.

2 Background

The EU-funded Horizon 2020 project TYPHON [6] has developed a model-based methodology and integrated technical offering for designing, developing, querying, evolving, analysing and monitoring scalable hybrid data persistence architectures. It is based on three Domain-Specific Languages (DSLs), namely TyphonML, TyphonDL and TyphonQL which facilitate designing, deploying and querying hybrid datastores, respectively.

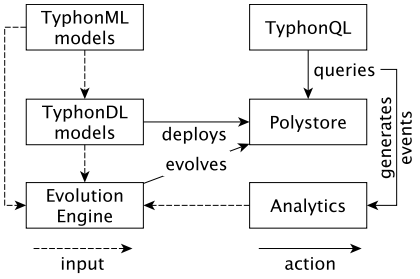

Fig. 1: An overview of the architecture of TYPHON.

Figure 1 shows an overview of the TYPHON architecture and in the following we briefly present the three aforementioned languages. Interested readers can find out about TYPHON and its different components in [6].

TyphonML TyphonML is a textual modelling language that supports the design of hybrid polystores. Developers, using TyphonML, create models that include information regarding the concepts appearing in the polystore, their

fields and their relationships. These models also include information about the databases that are involved in the system. As a result, they represent the high-level infrastructure of a hybrid polystore. An example is shown in Figure 2a.

```
entity Product {
    id : string[64]
    name : string[64]
    description : string[1024]
    category -> Category[1]
}

relationaldb RelationalDatabase{
    tables{
        table {
            ProductDB : Product
            index productIndex{
                attributes ('Product.name')
            }
            idSpec ('Product.name')
        }
    }
}
```

(a) TyphonML syntax example.

```
import EcommerceEnhanced.xmi
import RelationalDatabase.tdl
import DocumentDatabase.tdl
import dbTypes.tdl
containertype Docker
clustertype DockerCompose
platformtype localhost
platform platformName : localhost {
    cluster clusterName : DockerCompose {
        application Polystore {
            container RelationalDatabase : Docker {
                deploys RelationalDatabase
                ports {
                    target = 3306 ;
                }
            }
            container DocumentDatabase : Docker {
                deploys DocumentDatabase
                ports {
                    target = 27017 ;
                }
            }
        }
    }
}
```

(b) TyphonDL syntax example.

Fig. 2: Example of the TyphonML and TyphonDL syntaxes.

TyphonDL Arguably, the abstraction gap between high-level TyphonML models and ready-to-use polystores is not negligible. To bridge this gap, an intermediate polystore deployment modelling language (TyphonDL) is used. TyphonML models are transformed to *TyphonDL models* and are enhanced with more fine-grained database-specific configuration details. TyphonDL models represent the deployment infrastructure of that polystore in terms of the specific cloud platform and deployment tools employed and are used to generate the necessary installation and configuration facilities that, when executed, can assemble the polystore in an automated manner. An example is shown in Figure 2b.

TyphonQL As data in a TYPHON polystore is distributed across a number of heterogeneous databases a common data manipulation language is used. TyphonQL is developed for perfoming data manipulation commands (e.g., insert, delete, etc). Since TyphonQL queries[3] are only executable on polystores precisely specified using TyphonML and TyphonDL, dedicated compilers/interpreters exploit this rich structural and semantical information to type-check and transform TyphonQL queries to high-performance native queries and APIs.

3 Proposed Architecture

An overview of the developed polystore data event publishing and processing architecture is shown in Figure 3. Events undergo nine stages that are distributed among two main interleaved phases; authorisation and analytics. The authorisation phase involves validating if a new incoming (TyphonQL) query will be allowed execution by the polystore or not. The rules defined in authorisation tasks can be based either on hardcoded conditions (e.g., value of a specific field is above a threshold) or on information extracted from the history of previous events processed through the stream processor.

[3] An example TyphonQL "select" query: **from** User u **select** u.age **where** u.id==1

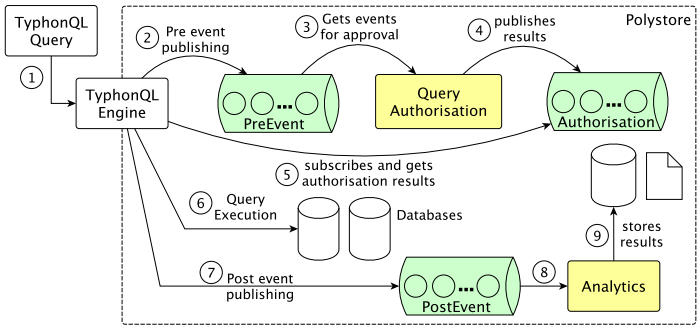

Fig. 3: The proposed data event publishing and processing architecture.

Such a query authorisation mechanism would be also possible at the application layer. An advantage though of using query authorisation at the level of the polystore is that many applications that connect to the same polystore can reuse those authorisation rules instead of having to implement and maintain them individually in each application.

The second phase of analytics involves the continuous consumption of TyphonQL queries that were executed. The tasks developed at this stage consume PostEvent objects (the structure of thse objects is described in Section 3.1). Below are the stages an event will go through as it progresses within the proposed architecture (numbers in the list correspond to those shown in Figure 3).

1. A query is passed by a user to the TyphonQL engine for execution.
2. TyphonQL publishes a pre-execution event (PreEvent) for the query, push it to a pre-event queue and waits for an authorisation decision of this event.
3. A stream processor (Apache Flink in the current implementation) dedicated to authorisation, consumes messages from the authorisation queue to apply the configured authorisation checks (presented in Section 3.2) before generating an authorisation decision of an event.
4. Following the application of the required authorisation checks, the stream processor publishes the authorisation decision to an authorisation queue.
5. TyphonQL receives the authorisation decision it was waiting for.
6. Based on the outcome of the authorisation decision, TyphonQL will execute the query received at step 1 or reject it.
7. A PostEvent object is generated and pushed to the analytics queue.
8. The analytics stream processor consumes the (post) event to which the relevant analytics tasks (described in Section 3.3) could be applied.
9. The results of the analytics can be stored/published using different mechanisms (e.g., a database, a filesystem, a web-service).

3.1 Data Event Structure

This section summarises the data analytics events structure metamodel which is presented in Figure 4. Both PreEvents and PostEvents have a unique *id* and store the TyphonQL *query* that generated them. The time when the query arrived for execution to the polystore is stored in the *queryTime* attribute of PreEvents.

A boolean flag, named *authenticated*, stores the result of the decision on if the query is approved for execution or rejected. The *resultSetNeeded* boolean property declares if the polystore needs to store the result of the execution of the commands in the PostEvent object after it is executed. The *slots* list acts as an extension mechanism to be utilised by polystore-backed applications. It hosts key-value pairs of any custom properties that analytics developers need to pass to the analytics workflow to accommodate their requirements when manipulating the events. A use of this feature is demonstrated in the scenario presented in the evaluation (see Section 4).

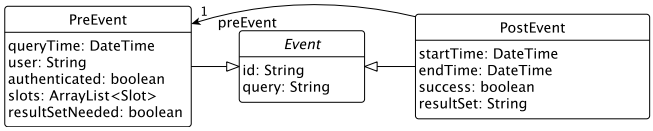

Fig. 4: The event metamodel.

PostEvent instances will point to their corresponding PreEvent instance. PostEvents also hold timestamps of when the execution started (*startTime*) and when it ended (*endTime*). PostEvents store a *success* flag declaring whether the execution of the query was successful or not. Finally, the result set returned from the execution of the command is stored in the *resultSet* attribute.

3.2 Authorisation Tasks

The event authorisation architecture is based on the concept of authorisation tasks. Each task contains logic that decides if a query should be executed or not against the polystore. In the proposed authorisation architecture, all the configured authorisation tasks are part of an authorisation chain. A PreEvent arriving for authorisation, visits authorisation tasks one after the other, unless a previously visited task has already rejected the execution of that event. A query is executed if it has been approved (or ignored) by all the tasks in the chain.

Developers can provide the aforementioned logic by implementing an abstract class (namely *GenericAuthorisationTask*) which is part of the analytics infrastructure. More specifically, they need to implement two methods for each authorisation task: i) the *checkCondition(Event event)* and ii) *shouldIReject(Event event)*. The first method (i.e., *checkCondition(. . .)*), checks if the task is responsible for approving or rejecting a query. The second method (i.e., *shouldIReject(. . .)*), includes the logic that defines if a query should be approved or rejected. The *shouldIReject* method is called if and only if the *checkCondition* method of the task evaluates to true.

The authorisation chain is built using the Flink's concept of side outputs [11]. Each stream of data in Flink can be transformed to another stream in which the data is grouped using tags based on some logic defined in the transformation operator. The analytics architecture automatically tags PreEvents into specific groups that facilitate the orchestration of the flow of events within the authorisation chain. More specifically, all rejected events, no matter which task rejected them, end up in a group tagged "Rejected". The events that were either approved

or not checked (because of the "checkCondition" method returning "false") by a task are placed under the group which is tagged by the name of the Task. Those are given as input to the next task in the chain where the process is repeated.

An orchestrator application is automatically generated (in Java) using a purpose-built code generator. The orchestrator subscribes to the pre-event queue, consumes the event stream and is responsible for re-directing the events to the appropriate tasks based on the results of each task's *checkCondition* and *shouldIReject* methods.

3.3 Analytics Tasks

Analytics tasks are implemented as individual Flink jobs. Each analytics task needs to implement the *analyze* method of an *IAnalyzer* interface provided by the architecture. The *analyze* method automatically subscribes to the post-event queue and provides a Flink datastream of PostEvent objects to the method. Developers are able to define their scenarios by using Flink's the built-in stream operators (e.g., map, process, aggegate, sum, etc.). A provided class that includes the main method for calling the analytics tasks is then used to deploy the scenario in a Flink execution enviorment.

3.4 Deployment

The analytics and authorisation tasks can be deployed in a Flink/Kafka infrastructure. This can be achieved by using one of the available containerized deployments (i.e., Docker and Kubernetes). TyphonDL (see Section 2) generates the necessary deployment scripts. For Docker, we use the wurstmeister Zookeeper[4] and Kafka[5] DockerHub images. The Kubernetes deployment is based on the Strimzi [9] package. Flink cluster deployment is achieved by using the official Apache Flink cluster deployment scripts [10].

4 Scalability Evaluation

The evaluation of the scalability of the proposed architecture requires ingestion of large volumes of data. In order to evaluate our work we developed an e-shop simulator that produce large volumes of synthetic, but realistic, data. The e-shop simulator is based on the notion of "Agents". An agent simulates the behaviour of one type of shopper (i.e., a User) in an e-shop. Developers can use either the *executeQuery(...)* method to execute a query against the polystore or the *createAndPublishPostEvent(...)/createAndPublishPreEvent(...)* to skip the execution of the command against the polystore and create directly a PostEvent/PreEvent object in the relevant analytics queues. To be able to evaluate the scalability of the proposed architecture, we opted for the latter option avoiding the overhead of having to wait for the execution of the actual command against the database in order to produce the Pre/PostEvent object.

[4] https://hub.docker.com/r/wurstmeister/zookeeper/
[5] https://hub.docker.com/r/wurstmeister/kafka/

4.1 Authorisation Chain Scalability Evaluation

In order to test the scalability of the authorisation chain, we produced an increasing number of events which were given as input into the Pre-Event queue. More specifically, users (agents) were simulating the placement of orders in the e-shop. TyphonQL "insert" commands (see Listing 1.1) were generated for the placed orders including details of the credit card used to pay the order. Three authorisation tasks were created, applying different validation rules on the credit card used. The first task checks the existence of a credit card in the query, the second if the credit card has expired and the third if the credit card number was valid.

Listing 1.1: An example TyphonQL insert command used in the experiment.

```
insert Order {id:'...', date:'...', total:'...', products:[...], user:'...',
paidWith: CreditCard {id:'...', number:'6007-2216-3740-9000',
expiryDate:'2021-06-25T08:36:13.656'}}
```

As described in Section 3, if an event is rejected by one authorisation task, it is not passed to the following task(s) in the chain but is directed automatically to the authorisation queue as rejected. In the simulator, the agents were producing orders that had always a credit card assigned to them, so they were approved by the first task. From those, half (50%) were having an expired credit card attached to them thus, they were rejected from the second task. Those passed successfully from the second task have a 50% chance of having an invalid credit card number. Following this pattern, we increased the variability as some of the events will be passing the whole chain, while some will be rejected earlier.

The chain was deployed in a cluster of three machines; one acting as the master and the other two as the workers[6]. In our experiment, the master was also hosting the relevant Kafka topics (PRE and AUTH). We restricted the Flink deployment to allocate and use only 8GB of the available 64GB for each worker.

Figure 5 shows the total execution time (in seconds) for processing all the event and posting the (rejected/approved) PreEvent in the authorisation queue. The graph shows linear scalability which confirms our expectation as the analytics architecture is built atop tools such as Apache Flink and does not add any bottleneck. The master node's average memory increases steadily and averages between 450 and 650MB as shown in Figure 5. The CPU is around 50% for all the experiments. The master node in this experiment was hosting the Kafka queue and more significantly the AUTH topic in which the workers were publishing the results, thus, the CPU utilisaton is justified by having the master node writing these events in the authorisation queue.

The average memory consumption and CPU utilisation for the workers is shown in Figure 6. In this scenario, both workers requiring increasing amount of memory for each scenario from the operating system while the CPU utilisation is between 60-70%. The CPU utilisation and memory consumption is similar across the cluster's workers which shows even distribution of work.

[6] AMD Opteron(tm) Processor 4226 – 6-cores @ 2.7Ghz, 4x16GB DD3 1066MHz RAM

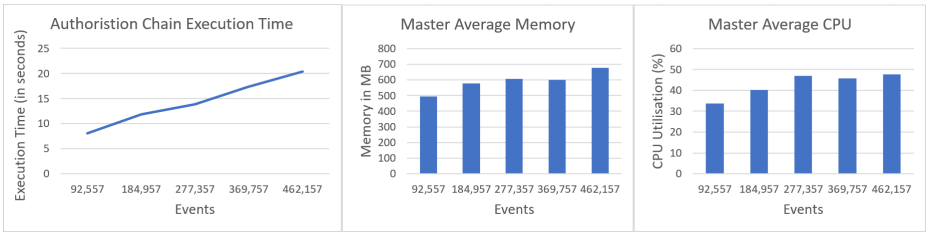

Fig. 5: Execution time, master's memory and CPU utilisation for authorisation.

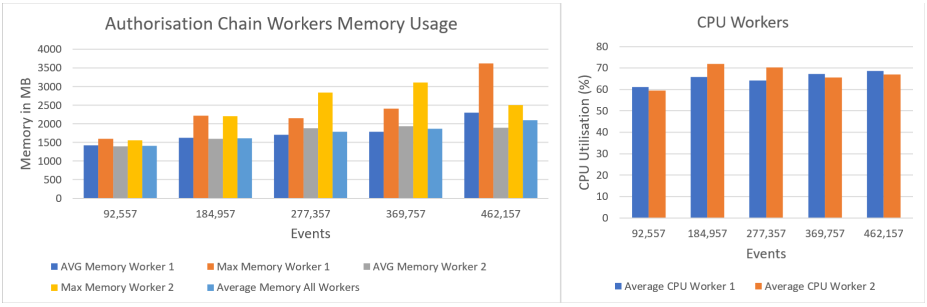

Fig. 6: Workers' memory consumption and CPU utilisation for authorisation.

4.2 Analytics Scalability Evaluation

We implemented a scenario in which a list of the top products that users browsed within a specific time window is produced. The simulator was instantiated with a varying number of users each of which was randomly navigating a number of products. Navigation of the catalogue has a result of generating one TyphonQL "select" query (e.g., ***from*** *Product p* ***select*** *p* ***where*** *p.id = '...'*) each time a product page was visited.

The implemented analytics scenario, consumes only those events (i.e., select events on the table Product). As it might be the case that users in real deployments might exploit such an analytics scenario to promote their products (i.e., by visiting their product page repeatedly), we were amending the *slots* attribute of the PreEvent object linked to the PostEvent object that our simulator generated with the id of the user that requested the execution of the command. Such information can be taken for example from the query where the session user id is passed as a comment to the produced query. We produced an increasing number of events which were given as input into the analytics architecture. The analytics code was deployed in the same cluster configuration as described in the evaluation of the authorisation chain.

The time needed is shown in Figure 7. The graph shows again linear scalability and our architecture does not add any bottleneck. The CPU utilisation and memory consumption for the master node (see Figure 7) remain quite low as in this experiment the workers are only reading from the POST queue hosted in the master and thus the master is not required to perform any writes to the authorisation queue.

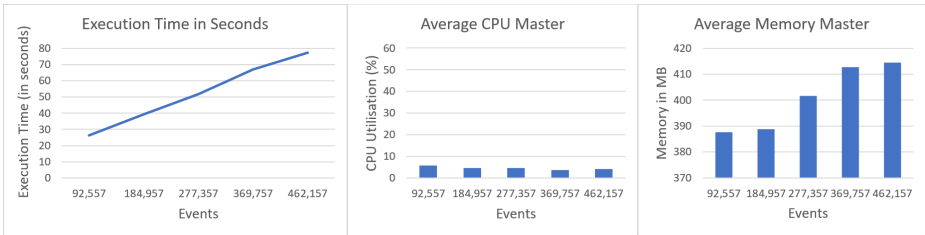

Fig. 7: Execution time, master's memory and CPU for the analytics experiment.

The workers' average and max memory consumption and the average CPU utilisation for the five simulated scenarios are shown in Figure 8. The workers are using above 80% of the available processing power on average across the five different scalability scenarios. Also, the JVM is claiming all the necessary memory (especially in the last 4 of the five scenarios) but is not running out of memory which is explained by the Java garbage collector replacing unnecessary memory when needed. The load balance is equally split among the workers both in terms of CPU utilisation and memory usage which demonstrates that the workload is shared equally in the cluster.

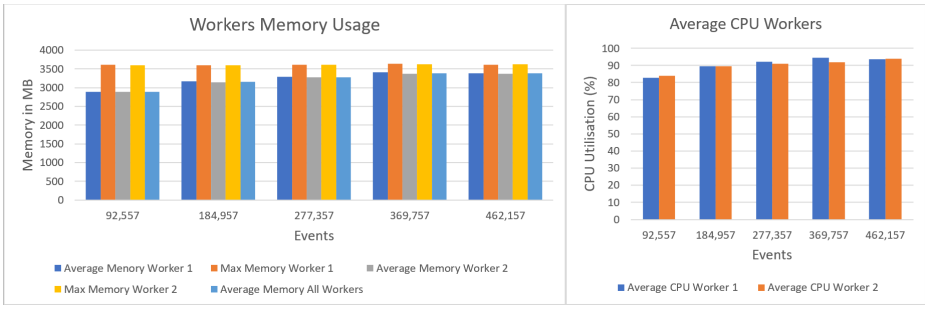

Fig. 8: Workers' memory consumption in the analytics scalability experiment.

5 Related Work

Different systems [8] have been proposed to capture database related evens to mostly allow replication or migration of databases. Connectors (e.g., KafkaConnect [2]) are registered to databases' specific mechanisms to extract already stored data and identify changes. Approaches like Maxwell's Deamon [12] and Oracle GoldenGate [7] monitor the database's log (i.e., binlog) to extract events but these are restricted to use only on relational databases. Debezium [3] offers an event capturing mechanism that supports both relational and non-relational databases. However, it only captures changing commands (i.e., insert, update, delete) and not select queries while it supports a limited number of databases. The Confluent platform [1] is a real-time event streaming application. It supports over 100 connectors to databases and filesystems each of which support different level of granularity of the events that can be captured.

The aforementioned approaches are either limited by the support for a specific set of database types or the amount of processible information. In addition, some of them require duplication of data or storing of unrelated events (e.g., the SQL

binlog stores, except the DML commands, DDL commands, too). All of the approaches require the use (and development in case it is not available) of a custom connector for every database and database type in the system. Except for the fact that such connectors might not be possible to be implemented if the database does not offer a related mechanism, the different connectors can acquire different levels of information based on what information the database can offer. In addition, these connectors act separately in each database. If a single TyphonQL command affects more than one database, a common scenario in polystores, then the matching of these events is a difficult - if possible at all - task. Finally, to the best of our knowledge, none of the approaches offer a pre-execution event capturing and authorisation mechanism.

The latter can be achieved with the use of database triggers. However, not all databases allow the execution of custom logic *before* the execution of the commands thus such a feature can be used with some of the databases in the polystore. Most allow the use of triggers after the actual execution of the command. However, this comes with the drawback of having to define specific triggers for each type of command and table/document affected separately which does not allow the creation of a single event that contains all the information needed for the extraction of analytics if a single polystore command affects multiple tables/documents within the same database and across the different databases.

Our approach offers both a before and after execution event capturing mechanism. Authorisation and analytics tasks have access to the data and databases the command affects, no matter if the latter had impact on multiple entities and different types of databases as it is based on a unified syntax (that of TyphonQL). Also, the latter allows future support of event capturing for any new database added to the polystore without requiring developers to implement specific database event capturing/triggering mechanism. Finally, in the case of migration of data from one type of database to another, the authorisation/analytics tasks do not need to be redeveloped to use the database-specific event triggering syntax.

6 Conclusions and Future Work

In this paper we presented a distributed architecture for analytics based on polystore queries. We also presented a pre-execution authorisation mechanism. Finally, the scalability of both the authorisation and the analytics components of the proposed architecture is evaluated. In future work, we will explore if authorisation tasks can be re-arranged automatically in the chain. Tasks that reject a higher proportion of events or require less time to execute could be positioned earlier in the chain. Machine learning can be used to identify the most efficient chains based on different features among those described (i.e., execution time and rejection rate). Finally, applying capture data change (CDC) mechanisms to further facilitate the extraction of analytics would be of interest.

Acknowlegements This work is funded by the European Union Horizon 2020 TYPHON project (#780251).

References

1. Confluent, Inc: Confluent: Apache Kafka and Event Streaming Platform for Enterprise, https://www.confluent.io/
2. Confluent.io: Kafka Connect, https://docs.confluent.io/current/connect/index.html
3. Debezium Community: Debezium, https://debezium.io/
4. Garg, N.: Apache Kafka. Packt Publishing Ltd (2013)
5. Hueske, F., Kalavri, V.: Stream Processing with Apache Flink: Fundamentals, Implementation, and Operation of Streaming Applications. O'Reilly Media (2019)
6. Kolovos, D., Medhat, F., Paige, R., Di Ruscio, D., Van Der Storm, T., Scholze, S., Zolotas, A.: Domain-specific languages for the design, deployment and manipulation of heterogeneous databases. In: 2019 IEEE/ACM 11th International Workshop on Modelling in Software Engineering (MiSE). pp. 89–92. IEEE (2019)
7. Oracle Corporation: Real-time access to realtime Information, Oracle White Paper (2015)
8. Rooney, S., Urbanetz, P., Giblin, C., Bauer, D., Froese, F., Garcés-Erice, L., Tomić, S.: Kafka: the database inverted, but not garbled or compromised. In: 2019 IEEE International Conference on Big Data (Big Data). pp. 3874–3880. IEEE (2019)
9. Strimzi: Strimzi - Apache Kafka on Kubernetes, https://strimzi.io/
10. The Apache Software Foundation: Apache Flink Clusters and Deployment, https://ci.apache.org/projects/flink/flink-docs-release-1.11/ops/deployment/
11. The Apache Software Foundation: Apache Flink Side Ouputs, https://ci.apache.org/projects/flink/flink-docs-stable/dev/stream/side_output.html
12. ZenDesk: Maxwell's Daemon, https://maxwells-daemon.io/