Stream and Historical Data Integration using SQL as Standard Language

Jefferson Amará[1], Victor Ströele[1], Regina Braga[1], Mário Dantas[1], Michael Bauer[2]

[1]Department of Computer Science – Federal University of Juiz de Fora (UFJF)
Juiz de Fora – Brazil

[2]Department of Computer Science – University of Western Ontario (UWO)
London – Canada

Abstract. *The complexity imposed by data heterogeneity makes it difficult to integrate 'streaming x streaming' and 'streaming x historical' data types. For practical analysis, the enrichment and contextualization process based on historical and streaming data would benefit from approaches that facilitate data integration, abstracting details and formats of the primary sources. This work presents a framework that allows the integration of streaming data and historical data in real-time, abstracting syntactic aspects of queries through the use of SQL as a standard language for querying heterogeneous sources. The framework was evaluated through an experiment using a relational database and real data produced by sensors. The results point to the feasibility of the approach.*

1. Introduction

The world and its relationship with data are migrating from data islands to a global data space paradigm. If a few years ago the term Big Data was exclusive in the scientific works and not well known in the daily life of civil society, nowadays it is a reality that permeates the routines of people [Abu-Salih et al. 2021, Barros 2020]. It is now of paramount importance in decision-making in the most diverse areas of knowledge and the global economy [Ghasemaghaei and Calic 2020, Wang et al. 2020].

In recent years, organizations have been dedicating themselves to leveraging the intelligent use of the vast amount of data produced [Mikalef et al. 2020, Shan et al. 2019]. The ability to manipulate efficiently this information and extract knowledge is now seen as a key factor in gaining a competitive advantage [María Cavanillas et al. 2016]. In addition to traditional data sources, which are modeled through persistent relations, applications with transient relations have become increasingly common. Also known as Data Streaming Applications, systems like IoT, sensor networks, mobile applications and social networks add, among others features, volume and heterogeneity to this global space of data [Akanbi and Masinde 2020].

Based on data of such different origins, an evident need is to integrate these sources [Asano et al. 2019, Tatbul 2010]. Several works have been developed in order to promote query mechanisms capable of integrating streaming data, addressing aspects of semantic optimization [Cappuzzo et al. 2020, Alkhamisi and Saleh 2020], continuous queries with sliding windows [Shein and Chrysanthis 2020], time alignment of queries [Tu et al. 2020] and aspects related to scalability [Stonebraker and Ilyas 2018].

In the context of *Industrial IoT (IIoT)* applications for example, [Costa et al. 2020] present a solution for real-time integration of data produced by *IoT* devices. In their

approach, data from intelligent devices, sensors and robots are extracted, processed, and stored in diverse, independent, and heterogeneous repositories. In that work, however, there is no proposal for the integration of data for monitoring in a unified and simplified way, that is, the complexities and features intrinsic to each data repository have their treatments delegated to the solution's consumers. Thus the integration for monitoring these repositories maintains the complexity at the level of user queries according to the characteristics of each repository.

It is worth noting that when it comes to data integration, the solutions that have had an impact are those that can be explained and easily comprehended by a human [Miller 2018]. Furthermore, the effort involved in querying this data can create barriers for consumers [Wang et al. 2018]. A central problem is a semantic gap between the way users express their queries and the different ways that data is represented internally [Freitas and Curry 2014].

With this context in mind, the objective of this work is to present a framework proposal that allows the integration for monitoring streaming data and historical data in real-time, abstracting syntactic aspects from the user, through the use of SQL as a standard language for querying heterogeneous data sources. For this purpose, this article followed four main steps: (i) review of related literature; (ii) definition of an architecture for executing SQL queries for data integration, and selection from heterogeneous sources; (iii) implementation of abstraction classes for the internal characteristics of data sources; and (iv) evaluation of results.

The following research questions were derived: **Q1)** Can the framework be used as a tool for joining stream and historical data described by heterogeneous data formats and models?; **Q2)** Is the solution extensible, that is, is it possible to add other data sources (stream or historical) in a simplified way?

The literature review includes articles related to streaming data integration. The architecture was proposed in order to enable the selection of data from heterogeneous sources and the execution of queries in standard SQL language. The architecture was developed based on the Apache Calcite[1] technology, and stream sensor and historical weather data were used to evaluate the proposal.

This article is organized as follows: Section 2 provides an overview of some related work; Section 3 describes the approaches and methods, including an overview of the proposal's architecture and infrastructure; In the Section 4 we present a feasibility study; Finally, the Section 5 presents conclusion and future directions.

2. Related Work

Considering the research areas on which this paper draws, we selected articles addressing data integration in streaming applications and historical data, and query mechanisms for real-time monitoring. The purpose was to understand important aspects of the theme and limitations of existing solutions.

In their work, [Asano et al. 2019] introduce Dejima, a framework focused on system aspects for data integration and control of update propagation of multiple databases.

[1]https://calcite.apache.org/

According to the authors, their solution combines two previous approaches to data integration; the first based on the global data schema, in which the data is integrated among a few databases using a single global schema, and the other based on the concept of 'peer' where the propagation of updates is cascaded through the peer networks. The authors do not present structural aspects of the queries.

In the research of [Brown et al. 2019], the focus is on ensuring data integrity during the data migration process, presenting CQL (Categorical Query Language) as an intuitive language to allow the movement and integration of data with complex schemas. They also point to the need for tools to combine heterogeneous datasets.

[Tian et al. 2013] present QODI (Query-driven Ontology-based Data Integration), an algorithm for dynamic mapping and query reformulation. In their research, they demonstrate QODI as a solution for data integration in heterogeneous distributed database systems. The queries are performed by ontology users through queries in the SPARQL language and translated to their destination databases. Although QODI is designed to integrate RDF data, its main motivation is the integration of relational data.

To support the query process with context enrichment, [Cavallo et al. 2018] present a semantic labeling module for performing queries in RDF statements with a query engine that combines SPARQL and SQL queries. They introduce the syntax of the query language with context enrichment SESQL (Semantically Enriched SQL).

Considering the works mentioned above, their gaps and limitations, as well as the problems identified by the authors themselves, the approach proposed in this present work presents itself as a feasible solution to the highlighted points. The main contributions of this work are: (i) allow the integration of data from distributed repositories, (ii) allow the integration of data between heterogeneous data sets, structured or not, relational or not, (iii) promote the execution of real-time queries in streaming and historical data, and (iv) provide the abstraction of syntactic aspects of the queries at the model level of the datasets, providing the possibility of queries through the use of the SQL standard.

3. Material and Methods

This work proposes a framework to facilitate the integration of streaming data produced by sensors with historical data. In this framework, the SQL language is used as the standard language, since it is the most used language for database queries [Toman 2017]. We assume some sensors produce a stream of data that needs to be ingested and analyzed together with historical data. This integration allows for a better understanding of the data and the detection of new knowledge, since the value of data increases when it can be linked and fused with other data [Analytics 2016].

Structured, semi-structured, and unstructured data, such as sensor data and any type of logs, business events, and user activities, are produced in large volume and must be processed by data streaming tools. We assume that these tools are configured in such a way as to have enough computational power to process and capture data streams.

Upon receiving data from the streaming tools, data are analyzed and stored for future analysis. Data that may not be attractive for analysis today may be important further, so it is necessary not to discard data that could generate relevant information in the future. Once ingested and stored, data needs to be analyzed continuously. Monitoring tools are

used to gain insights at a very high speed through near real-time analytic dashboards.

We believe that the use of the SQL language allows users to access data from different sources, transparently, regardless of the data model of that source (files, relational or non-relational databases, etc.). Thus, monitoring tools, external APIs, and users, can have easier access to different data sources through the use of this framework.

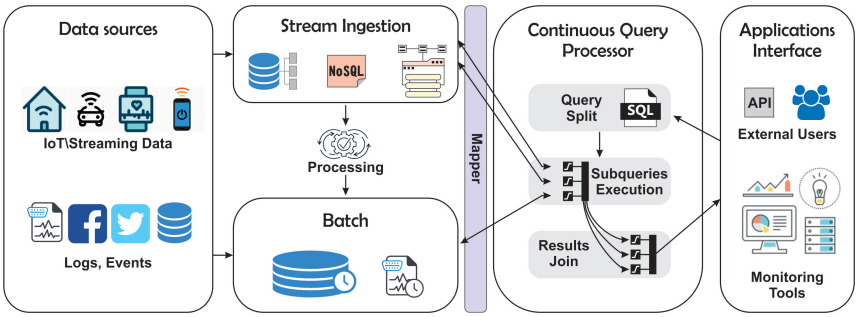

Figure 1. Framework Architecture

Figure 1 provides an overview of the framework components, defined to support continuous monitoring of data streaming, enabling integration with historical data repositories. The components are described below.

The **Data sources** component is responsible for monitoring the data produced by different devices, which can be sensors, IoT devices, logs, social networks, among others. In this component, two types of applications are expected: applications dedicated to ingesting raw stream data, focusing on high throughput and low latency; and applications aimed at scheduled data ingestion, in which data extraction and processing routines are performed periodically.

In the **Stream Ingestion** component, streaming data processing tools are configured to support applications such as Flink, Kafka, Spark, Storm, etc. These tools have several operators, such as varied windowing, join of streams, and pattern detection, being able to process and manipulate streaming data in repositories with diversified data models, respecting the defined windows for streaming processing [Garofalakis et al. 2016]. Windowing is the technique of executing aggregates over streams, being classified as Tumbling windows (no overlap) and Sliding windows (with overlap). Data processed in the windows are stored for further analysis.

The data processed by the Stream Ingestion component and the data ingested by schedule are stored in the **Batch** component. Dedicated repositories for storing historical data are also defined in this component. We designed these components based on the Lambda Architecture, which, in general, has three layers, represented in our framework by the components: Stream Ingestion (Speed), Batch, and Continuous Query Processor (Serving) [Kiran et al. 2015].

In the component **Continuous Query Processor** the core of the solution is defined. SQL queries submitted to these components are pre-processed to identify the data repositories involved in the query. If it is identified that the query must be executed in more than one repository with different data models, a set of subqueries is generated. The *Mapper* receives these subqueries, transforms them to the target repository's query

language, and submits them for execution. Subqueries run in parallel to optimize data fetching. The combination of the subqueries' results is done considering the criteria for creating the subqueries, respecting the filters originally defined in the main query.

In the **Applications Interface** component, a set of applications can be used to consume both historical and stream data. These applications submit SQL queries and receive the query result in a tabular format, standard SQL.

3.1. Infrastructure

In this paper, we focus on the development of the **Continuous Query Processor** component. This component was developed using the Java language and is an extension of Apache Calcite. The component has three main functions: monitor the user's interaction with the system; query validation considering the data sources configuration; and orchestrate the services communication, considering the multiple threads approach. Figure 2 presents a view of this component, as well as the technologies used in its development.

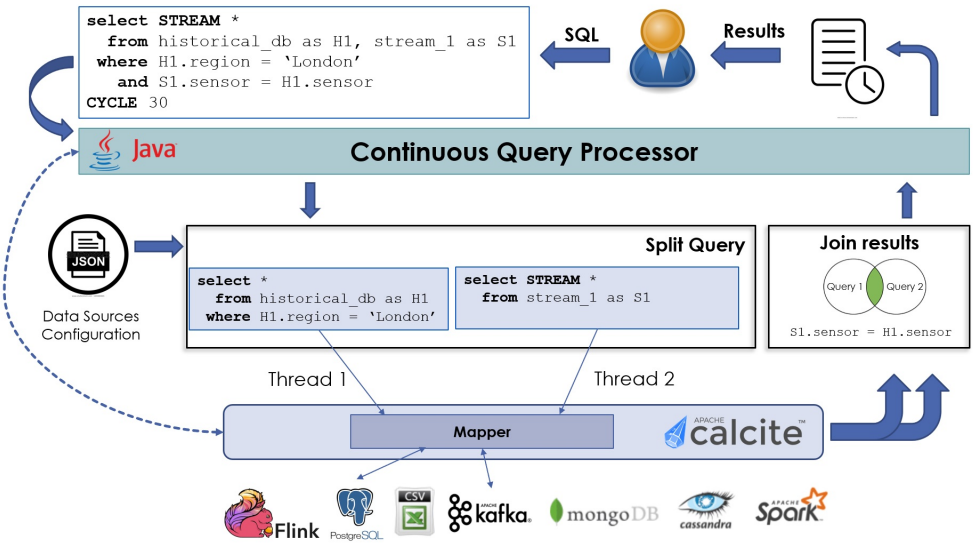

Figure 2. Implementation Solution

We use JSON files to configure the data sources, following the model defined in Apache Calcite, which was used as a query solve in *Mapper*. In JSON, a schema was configured for each of the three repositories used in this study: PostgreSQL, CSV, and Kafka. Thus, it was possible to identify the data sources involved in SQL queries. Part of this file can be seen in Figure 3. The complete files are available on GitHub[2]. As we are using Apache Calcite, our framework is restricted to implemented adapters by it[3].

The SQL query submitted by the user is processed and, based on JSON, the subqueries are created in order to guarantee that all filters and relations refer to the same schema. Filters were characterized between specific filters and intersection filters. Specific filters are those with single-schema relations (eg, *H1.region = 'London'*). Intersection Filters, on the other hand, are those that have relations from more than one schema

[2]https://bitbucket.org/JeffersonAmara/tcc-dcc-2021/src/master/
[3]https://calcite.apache.org/docs/adapter.html

and, therefore, they can only be applied after executing the subqueries. In Figure 2 two subqueries are created, and the intersection filter *S1.sensor = H1.sensor* is not considered in this first step. A Thread is created for each subquery so that execution occurs in parallel in the component process. The mapping of each subquery is the responsibility of the Mapper, which was implemented using the conversion models defined in Apache Calcite.

Based on the same JSON files used to identify the schemas, Apache Calcite identifies the subquery's language, makes the necessary conversions, and executes the query in the appropriate repositories. The STREAM schema subqueries' are executed on the data contained in the stream window.

```
{
  "version": "1.0",
  "schemas": [
    {
      "name": "historical",
      "type": "jdbc",
      "jdbcUser": "x",
      "jdbcPassword": "y",
      "jdbcUrl": "jdbc:postgresql://server:5432/historical?user=x&password=y",
      "jdbcCatalog": "historical",
      "jdbcSchema": "public",
      "jdbcDriver": "org.postgresql.Driver",
      .....
    }

    {
      "name": "KAFKA",
      "tables": [
        {
          "name": "KAFKA",
          "type": "custom",
          "factory": "org.apache.calcite.adapter.kafka.KafkaTableFactory",
          .....
        }
      ]
    }
  ]
}
```

Figure 3. Part from JSON file with configuration of two schemas: Relational database (PostgreSQL) and data streaming (Kafka).

Upon receiving the result of the execution of each subquery, the Continuous Query Processor component joins the results and applies the intersection filters, which were not applied due to the separation of the query in its subqueries. In the current stage of implementation, the framework does not have implementations of all join operators, only the "inner join" operator was implemented.

As it is a Continuous Query Processor, the query's result is delivered to the user/application/monitoring tool and the same query is executed again considering the execution cycle length defined in the submission of the query. In Figure 2, the user has defined an SQL query "**select STREAM * ... CYCLE 30**". This query is executed every 30 minutes. As data stream are limitless, this cycle remains until the user stops the execution.

4. Feasibility Study

In this section, we conduct a feasibility study, presenting real use case scenarios in which the architecture can be used to join streaming data produced by sensors in a Home Environment with historical temperature data.

A feasibility study attempts to characterize a technology to ensure that it actually does what it claims to do and is worth developing [dos Santos 2016]. As this is the first

effort to implement the solution and no interface has been implemented, it has become prohibitive to apply more traditional assessment methods, such as case studies. Instead, we use the Goal-Question-Metric (GQM) methodology [Caldiera and Rombach 1994].

4.1. Datasets description

One of the datasets used in this evaluation is a sensor dataset with data collected from three residences. The layout plan of two of them is presented in Figure 4, with the location of each sensor identified in the floor plan.

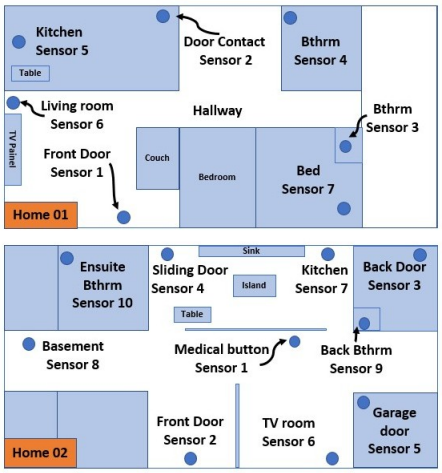

Figure 4. Layouts of home 01 and home 02.

The sensor data has the following features: *resident number*, *sensor number*, *message*, *day*, *month*, *year*, *hour*, *minute* and *mill-seconds*. Data were collected from these three homes over a year. Home 01 has 7 sensors and the data were monitored from 05/15/2018 to 05/31/2019, representing 381 days of data collection. Home 02 has 10 sensors and monitoring occurred from 08/09/2018 to 08/31/2019, a total of 352 days. Home 03 has 8 sensors whose data were received from 10/23/2017 to 9/30/2018, or 342 days. Note that each sensor message has an associated timestamp and that there are time gaps between consecutive sensor messages, sometimes on the order of minutes or even hours. While we could process sensor messages in actual time, the processing would require over a year of actual time. To evaluate our framework, we processed the sensor data as a continuous stream using Kafka.

Table 1 shows the messages sent by sensors during the monitoring period from Home 01; the other houses follow the same message pattern. We can see that it is possible to detect the action taken by the person who activated or deactivated the sensor. In general, the sensors are activated or deactivated and send messages related to these two actions.

The second dataset is about Historical Climate Data. Data made available by the Canadian government were used[4]. In this repository are available weather data for the period 2004-present. Using this service, collecting historical data about weather, climate data, and related information for numerous locations across Canada is possible. Some available data are temperature, precipitation, degree days, relative humidity, wind speed and direction, monthly summaries, averages, extremes, and climate norms.

[4]https://climate.weather.gc.ca/index_e.html

Table 1. Sensors messages of Home 01

Sensor Number	Message
1	Front Door Contact Opened
	Front Door Contact Closed
2	Back Door Contact Opened
	Back Door Contact Closed
3	Bedroom Motion Activated
	Bedroom Motion Idle
4	Bathroom Motion Activated
	Bathroom Motion Idle
5	Kitchen Motion Activated
	Kitchen Motion Idle
6	Living Rm Motion Activated
	Living Rm Motion Idle
7	ABS Bed Sensor Occupied
	ABS Bed Sensor Briefly Vacated
	ABS Bed Sensor Vacated

We collected data for the household monitoring period (2017-2019) to enable integrating these data using their dates as a filter. A total of 3 years of data for the region of London (Ontario) were collected and stored in a PostgreSQL relational database. Based on these data, we used the framework to join the two datasets and evaluate our solution.

4.2. Joining Stream and Historical Data

Based on the data described above, we want to make queries capable of enriching the data produced by the sensors with historical data. The purpose of this scenario is: **analyze** the framework **with the purpose of** evaluating **with respect to** its usefulness and extensibility **from the point of view of** a researcher **in the context of** joining stream and historical data. We thereby derive the following questions:

Q1) Can the framework be used as a tool for joining stream and historical data described by heterogeneous data formats and models?

Q2) Is the solution extensible, that is, is it possible to add other data sources (stream or historical) in a simplified way?

In many cases, users need to correlate persistent historical data and reference data with a real-time data stream to make smarter system decisions. This type of join requires an input source for the reference/historical data to be defined. Some tools (Flink, Spark) allow the user to implement the join between Stream and Tables. However, each tool works on a language, making the integrated use of the data produced by them difficult.

Following the architectural proposal defined in Section 3, the user configures the JSON file with the schemas referring to each data source, one for the sensor stream and another for the relational database with the weather data. With this configuration, the Continuous Query Processor component is able to identify the data sources involved in the submitted query, create the sub-queries, execute them using Apache Calcite as a mapper, and present the consolidated results, providing users with a SQL-based solution.

For example, let's assume the user is interested in monitoring sensor readings for the Back Door Contact. Also, he wants to check weather information when the sensor detects that the door is open. In other words, he wants to execute an SQL query that brings up information that is distributed in two repositories with different data models. The following query can be submitted to the Continuous Query Processor component to consolidate this information.

```
SELECT STREAM Station_Name, time_lst, temp_c,
       Wind Spd_km_h, weather, dates,
       res, sensor, message
  FROM historical.historical_climate AS h,
       kafka.ambient_sensor AS s
 WHERE h.dates = s.dates
   AND s."sensor" = `Sensor 2`
   AND s."message" LIKE '%Opened%';
```

The subqueries 01 and 02 are generated and submitted to Apache Calcite. The specific filters are applied in Subquery 01 to reduce the volume of data capture in the stream. On the other hand, the intersection filter (h.date = s.date) and the projections defined in the select clause are applied just when the component joins the historical and the stream results.

Subquery 01:

```
SELECT STREAM *
  FROM kafka.ambient_sensor AS s
 WHERE s."sensor" = `Sensor 2`
   AND s."message" LIKE '%Opened%';
```

Subquery 02:

```
SELECT *
  FROM historical.historical_climate AS h;
```

Figure 5 presents the result of the query returned by the component. In this figure only part of the results is presented. As it is a stream query, the component presents the results separately, considering the data being ingested by Kafka.

Station Name	time_lst	temp_c	Wind Spd_km_h	weather	date	res	sensor	message
LONDON A	09:00	19.5	5	NA	2018-05-15	Res3053	Sensor 2	Back Door Contact Opened
LONDON A	13:00	22.7	4	Mainly Clear	2018-05-15	Res3053	Sensor 2	Back Door Contact Opened
LONDON A	15:00	22.1	17	NA	2018-05-15	Res3053	Sensor 2	Back Door Contact Opened
LONDON A	16:00	21.1	27	Mostly Cloudy	2018-05-15	Res3053	Sensor 2	Back Door Contact Opened

Figure 5. One of the results returned by the Continuous Query Processor component.

In addition, other schemas can be configured, allowing users to monitor data (history and stream) from heterogeneous repositories without the need for knowledge of specific languages. This reinforces the utility appeal of the solution since users can consume the data without directly interfacing with storage solutions. Finally, although not all SQL operations are implemented in this version, it can be used in scenarios where it is necessary to join data by equality criteria, that is, using an inner join.

With this project, both location and access transparency are provided [Coulouris et al. 2005], freeing users from the need to understand the underlying storage solution and how data is organized in this distributed environment. In addition, this conceptual design also offers the option of using the SQL language for queries, enabling its use by most database users and by most of the tools used for data monitoring (e.g., dashboards).

This way, given the scenario at hand the elements that compose it, we can answer the questions previously outlined. **(Q1) Can the framework be used as a tool for joining stream and historical data described by heterogeneous data formats and models?** *Partly*. We demonstrate in this scenario that we can query stream and historical data. By automating the process of split the main query and execute the subqueries in parallel using Apache Calcite as mapper. Via the query endpoints, users can interface with the solution in a transparent manner, leaving to the Continuous Query Processor component the interpretation and proper redirection of incoming queries. However, the current implementation does not support all SQL operations, because of that we are answering this question *partly*. **(Q2) Is the solution extensible, that is, is it possible to add other data sources (stream or historical) in a simplified way?** *Yes*. In the analyzed scenario, we show that by using the Apache Calcite and JSON configuration files, new storage solutions (schemas) can be added, exempting the need to restructure the solution. This way, we show the framework's viability for this scenario, since it can contemplate the goals previously outlined.

5. Final Remarks and Future Works

This article proposes a framework that enables the monitoring of data produced by IoT devices and sensors and its integration with historical data. The proposed solution is based on the SQL language and seeks to facilitate access to and the use of distributed data repositories with different data models. Furthermore, users can use the framework to enrich data produced by IoT devices and sensors by integrating them with historical databases.

The framework was developed as an Apache Calcite extension, using JSON files to configure the data sources. With this, the framework can detect which data sources are involved in the query, create the subqueries and execute them in parallel, with Apache Calcite working as a mapper. Finally, the results of the subqueries are joined, respecting the intersection filters (inter-queries filters).

Real data produced by sensors in assisted home environments and time data were used in the feasibility study conducted to evaluate the proposed solution. The framework integrated this heterogeneous data in a non-intrusive way and allowed the user to access the data by submitting a single query, thus enabling a comprehensive analysis of historical and stream data in a unified system. The results obtained with the evaluation and the answers to the proposed research questions allow us to conclude that the developed framework achieved the objectives of this work.

So far, the framework does not allow all join operations to be applied on the integrated results from different repositories. This is future work and further we plan to continue with the development of other SQL operations and also incorporate elements for manipulating data stream windows. Although Apache Calcite adapters limit our solution,

many initiatives are underway to expand the range of data models supported by Apache Calcite[5]. Given the complexity of streaming processing, we intend to evaluate real-time requirements (i.e., low latency, high throughput, scalability, and fault tolerance). Finally, we plan to measure the overhead generated by the framework for executing the queries since the monitoring tools are almost real-time.

References

Abu-Salih, B., Wongthongtham, P., Zhu, D., Chan, K. Y., Rudra, A., Abu-Salih, B., Wongthongtham, P., Zhu, D., Chan, K. Y., and Rudra, A. (2021). Social big data: An overview and applications. *Social Big Data Analytics: Practices, Techniques, and Applications*, pages 1–14.

Akanbi, A. and Masinde, M. (2020). A distributed stream processing middleware framework for real-time analysis of heterogeneous data on big data platform: Case of environmental monitoring. *Sensors*, 20(11):3166.

Alkhamisi, A. O. and Saleh, M. (2020). Ontology opportunities and challenges: Discussions from semantic data integration perspectives. In *2020 6th Conference on Data Science and Machine Learning Applications (CDMA)*, pages 134–140. IEEE.

Analytics, M. (2016). The age of analytics: competing in a data-driven world. *McKinsey Global Institute Research*.

Asano, Y., Herr, D.-F., Ishihara, Y., Kato, H., Nakano, K., Onizuka, M., and Sasaki, Y. (2019). Flexible framework for data integration and update propagation: System aspect. In *2019 IEEE International Conference on Big Data and Smart Computing (BigComp)*, pages 1–5.

Barros, M. (2020). Book review: Digital objects, digital subjects: Interdisciplinary perspectives on capitalism, labour and politics in the age of big data.

Brown, K. S., Spivak, D. I., and Wisnesky, R. (2019). Categorical data integration for computational science. *Computational Materials Science*, 164:127–132.

Caldiera, V. R. B.-G. and Rombach, H. D. (1994). Goal question metric paradigm. *Encyclopedia of software engineering*, 1:528–532.

Cappuzzo, R., Papotti, P., and Thirumuruganathan, S. (2020). Creating embeddings of heterogeneous relational datasets for data integration tasks. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 1335–1349.

Cavallo, G., Di Mauro, F., Pasteris, P., Sapino, M. L., and Candan, K. S. (2018). Contextually-enriched querying of integrated data sources. In *2018 IEEE 34th International Conference on Data Engineering Workshops (ICDEW)*, pages 9–16. IEEE.

Costa, F. S., Nassar, S. M., Gusmeroli, S., Schultz, R., Conceição, A. G., Xavier, M., Hessel, F., and Dantas, M. A. (2020). Fasten iiot: An open real-time platform for vertical, horizontal and end-to-end integration. *Sensors*, 20(19):5499.

Coulouris, G., Dollimore, J., Kindberg, T., and Blair, G. (2005). *Distributed Systems: Concepts and Design*. Pearson Education, 5th edition.

[5]https://calcite.apache.org/docs/powered_by.html

dos Santos, R. P. (2016). *Managing and monitoring software ecosystem to support demand and solution analysis*. PhD thesis, Universidade Federal do Rio de Janeiro.

Freitas, A. and Curry, E. (2014). Natural language queries over heterogeneous linked data graphs: A distributional-compositional semantics approach. In *Proceedings of the 19th international conference on Intelligent User Interfaces*, pages 279–288.

Garofalakis, M., Gehrke, J., and Rastogi, R., editors (2016). *Data Stream Management*. Springer Berlin Heidelberg.

Ghasemaghaei, M. and Calic, G. (2020). Assessing the impact of big data on firm innovation performance: Big data is not always better data. *Journal of Business Research*, 108:147–162.

Kiran, M., Murphy, P., Monga, I., Dugan, J., and Baveja, S. S. (2015). Lambda architecture for cost-effective batch and speed big data processing. In *2015 IEEE International Conference on Big Data (Big Data)*, pages 2785–2792. IEEE.

María Cavanillas, J., Curry, E., and Wahlster, W. (2016). *New horizons for a data-driven economy: a roadmap for usage and exploitation of big data in Europe*. Springer Nature.

Mikalef, P., Pappas, I., Krogstie, J., and Pavlou, P. A. (2020). *Big data and business analytics: A research agenda for realizing business value*. Elsevier.

Miller, R. J. (2018). Open data integration. *Proceedings of the VLDB Endowment*, 11(12):2130–2139.

Shan, S., Luo, Y., Zhou, Y., and Wei, Y. (2019). Big data analysis adaptation and enterprises' competitive advantages: the perspective of dynamic capability and resource-based theories. *Technology Analysis & Strategic Management*, 31(4):406–420.

Shein, A. and Chrysanthis, P. K. (2020). Multi-query optimization of incrementally evaluated sliding-window aggregations. *IEEE Transactions on Knowledge and Data Engineering*.

Stonebraker, M. and Ilyas, I. F. (2018). Data integration: The current status and the way forward. *IEEE Data Eng. Bull.*, 41(2):3–9.

Tatbul, N. (2010). Streaming data integration: Challenges and opportunities. In *2010 IEEE 26th International Conference on Data Engineering Workshops (ICDEW 2010)*, pages 155–158. IEEE.

Tian, A., Sequeda, J. F., and Miranker, D. P. (2013). Qodi: Query as context in automatic data integration. In *International Semantic Web Conference*, pages 624–639. Springer.

Toman, S. H. (2017). The design of a templating language to embed database queries into documents. *Journal of Education College Wasit University*, 1(29):512–534.

Tu, D. Q., Kayes, A., Rahayu, W., and Nguyen, K. (2020). Iot streaming data integration from multiple sources. *Computing*, 102(10):2299–2329.

Wang, J., Yang, Y., Wang, T., Sherratt, R. S., and Zhang, J. (2020). Big data service architecture: a survey. *Journal of Internet Technology*, 21(2):393–405.

Wang, X., Haas, L., and Meliou, A. (2018). Explaining data integration. *Data Engineering Bulletin*, 41(2).