Integrating Heterogeneous Stream and Historical Data Sources using SQL

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), Brazil
{jnamara, victor.stroele, mario.dantas}@ice.ufjf.br, regina.braga@ufjf.edu.br
^{2} Department of Computer Science – University of Western Ontario (UWO), Canada
bauer@uwo.ca

Abstract. Applications capable of integrating data from historical and streaming sources can make the most contextualized and enriched decision-making. However, the complexity of data integration over heterogeneous data sources can be a hard task for querying in this context. Approaches that facilitate data integration, abstracting details and formats of the primary sources can meet these needs. This work presents a framework that allows the integration of streaming 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 relational datasets and real data produced by sensors. The results point to the feasibility of the approach.

Categories and Subject Descriptors: H.2 [**Database Management**]: Miscellaneous; H.3 [**Information Storage and Retrieval**]: Miscellaneous; I.7 [**Document and Text Processing**]: Miscellaneous

Keywords: Data Integration, Heterogeneous Data, Historical Data, Streaming Data, Structured Query Language (SQL)

1. INTRODUCTION

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 manipulating efficiently this information and extracting knowledge is now seen as a key factor in gaining a competitive advantage. 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].

These *"new"* sources of data are characterized by being open-ended, flowing at high speed, and generated by non-stationary distributions in dynamic environments [Gama 2012]. They also have become ubiquitous since many applications generate a huge amount of data at a great velocity. This made it difficult, for example, for existing data mining tools, technologies, methods, and techniques to be applied directly to big data streams due to their inherent dynamic characteristics [Kolajo et al. 2019].

[Kolajo et al. 2019] point to the relevance and increasing importance of researches in data stream field and trends of big data stream tools and technologies, as well as methods and techniques employed in analysing big data streams. They also point to the Data Integration as a key issue in big data stream analysis. [Galhotra et al. 2020] point that most of the data fairness literature ignores the fact that data integration is one of the primary steps to generate high quality data, and [Tan 2021] sees querying heterogeneous data with the concept *"data outside the box"* as a challenge and opportunity.

Based on data of such different origins, an evident need is to integrate these sources, whether historical or stream, structured or not [Asano et al. 2019]. 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].

[de Souza Campos et al. 2021] performed a systematic review about methods and techniques used in data integration. That research also revealed the growing use of algorithms in query processing between different data sources and logs from heterogeneous data sources, which points to the relevance of the present work.

In the context of Industrial IoT (IIoT) applications for example, [Costa et al. 2020] present a solution for real-time integration in Big Data, considering data heterogeneity 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 integrating data for monitoring in a unified and simplified way, i.e., 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.

With this context in mind, the objective of this work is to allow queries that integrate stream and historical data to be performed more easily, that is, in a more standardized way, regardless of the origin, format, model or heterogeneity of the data, abstracting syntactic aspects from the user, through the use of SQL as a standard language for querying heterogeneous data sources. For this purpose, this work 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 combining 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 work expands a paper from the Brazilian Symposium on Databases [Amará et al. 2021]. As a new material, it introduces an extension of the framework for heterogeneous data integration, considering both stream and historical data. We updated the paper by adding important definitions about these research topics. Also, we extended the evaluation to enrich the framework analysis. To do that, we used other datasets with different formats to analyze the framework concerning its facility in adding other data types and querying them using SQL. Overall, our proposal and analyses represent a step further in data integration, considering both stream and historical data.

[1]https://calcite.apache.org/ (Accessed at 06/06/2022)

This article is organized as follows: Section 2 provides a background over the topics addressed in this paper; Section 3 overviews some related work; Section 4 describes the approaches and methods, including an overview of the proposal's architecture and infrastructure; in Section 5 we present a feasibility study; finally, Section 6 presents the conclusion and future directions.

2. BACKGROUND

The Internet of Things (IoT) is a well-known paradigm that defines a dynamic environment of interrelated computing devices with different components for seamless connectivity and data transfer [Stoyanova et al. 2020]. IoT is implemented in most of the areas in day-to-day applications such as lifestyle, retail, city, building, transportation, agriculture, healthcare, environment, and energy. Some of the applications are smart homes, smart cities, smart energy, smart industry, etc. [Smys 2020].

According to [Zanella et al. 2014], with so many data sources from such heterogeneous fields of application, it becomes a challenge to find solutions that meet the requirements of all scenarios, from aspects related to the acquisition infrastructure, passing through storage paradigms and the process of acquiring useful knowledge from this volume of data.

The IoT scenario is the era of streaming data that are usually represented in different structures or even semi/non-structures. As such, capturing and/or transferring these heterogeneous data in different formats (e.g., .csv, .txt, .html, .xml and so on) into a unified form, which is suitable for analysis, is a challenging task [Chen et al. 2013]. Another issue revolves around how to integrate IoT streaming data from multiple sources on the fly in real-time, as big data query and indexing [Tu et al. 2020]. In the present work, the context of IoT is justified, since the primary source of the data used in the research is precisely distributed environmental sensors.

A data stream is a real-time, continuous, ordered (implicitly by arrival time or explicitly by timestamp) sequence of items [Golab and Özsu 2003]. Data stream is an ordered sequence of instances that can be read only once or a small number of times using limited computing and storage capabilities. With these definitions in mind, Data Streaming can be defined as the process of transmitting a continuous flow of data (also known as streams), typically fed into stream processing software to derive valuable insights [Tibco 2019].

Data streams are usually generated by external sources or other applications and are sent to a Data Stream Management System (DSMS). Typically, DSMS do not have direct access or control over the data sources [Jiang and Chakravarthy 2009]. It is a new paradigm useful because of new sources of data generating scenarios which include ubiquity of location services, mobile devices, and sensor pervasiveness. Examples of such applications include financial applications, network monitoring, security, telecommunications data management, web applications, manufacturing, sensor networks, and others. In these applications it is not feasible to load the arriving data into a traditional data base management system (DBMS) and traditional DBMS are not designed to directly support the continuous queries required by these applications. The fundamental assumption of this paradigm is that the potential value of data lies in its freshness and, with stream computing, organisations can analyse and respond in real-time to rapidly changing data [Kolajo et al. 2019].

Also according to [Kolajo et al. 2019], the essence of big data streaming analytics is the need to analyse and respond to real-time streaming data from diverse sources, using continuous queries so that it is possible to continuously perform analysis on the fly within the stream. In this scenario, data are analysed as soon as they arrive in a stream to produce result as opposed to batch computing, where data are first stored before they are analysed. [Jiang and Chakravarthy 2009] also highlight some characteristics about data stream itself and **Data Stream Applications**, which are organized as follow:

—Continuous processing of newly arrived data is necessary.

—Many applications can tolerate approximate results as long as other critical (e.g., response time) requirements are satisfied.

—Many applications have very specific Quality of Service (QoS) requirements.

—Usage of available resources (e.g., CPU cycles, memory) to maximize their impact on QoS metrics is critical for stream-based applications.

—Finally, Complex Event Processing (CEP), rule processing, and notification are other important requirements of many stream-based applications that detect events or conditions and have to fire rules/triggers/actions in a timely manner when abnormal or user-defined events are detected.

[Golab and Özsu 2003] list some **Data Stream characteristics**:

—**The input from the data stream**: It is unpredictable in terms of rate or volume of data.

—**The Data Model**: Each data stream could have its own underlying data model or not (e.g. it could be semi-structured such as an XML document, or not structured like a text file).

—**Data Reliability**: The data inside of the data stream are processed like they are, being not free of errors because they depend on the data source and the transited path.

2.1 Batch vs Stream Data Processing

It is possible to group Big Data Processing into two major groups according to the state of art of this domain: Batch Data Processing and Stream Data Processing [Sakr et al. 2015]. Batch processing is a well known paradigm which involves operating over a large, static dataset and returning the result at a later time when the computation is complete. Thus, the state of data is maintained for the duration of the calculations during the processing [Gurusamy et al. 2017]. This is typically ideal for non-time sensitive work. According to [Gurusamy et al. 2017] the datasets in batch processing are bounded (batch datasets represent a finite collection of data), persistent (data is almost always backed by some type of permanent storage), large (batch operations are often the only option for processing extremely large sets of data).

While batch processing is a good fit for certain types of data and computation, other workloads require more real-time processing [Venkatesh et al. 2019]. Stream Data Processing is required for these needs, most of the data generated in a real-time data stream need real-time data analysis. In addition, the output must be generated with low-latency and any incoming data must be reflected in the newly generated output within seconds [Kolajo et al. 2019]. Also according to [Gurusamy et al. 2017], the datasets in stream processing are considered "unbounded". This has a few important implications:

—The total dataset is only defined as the amount of data that has entered the system so far.

—The working dataset is perhaps more relevant and is limited to a single item at a time.

—Processing is event-based and does not "end" until explicitly stopped. Results are immediately available and will be continually updated as new data arrives.

For dealing with stream and batch processing, one of the most common system architectures is called Lambda Architecture [Kiran et al. 2015]. It combines two different processing layers, namely batch and speed layers, each providing specific views of data while ensuring robustness, fast, and scalable data processing [Yousfi et al. 2021]. Lambda Architecture caters to three layers (1) Batch processing for pre-computing large amounts of data sets, (2) Speed or real-time computing to minimize latency by doing real-time calculations as the data arrives, and (3) an interface layer for the users send queries and get its responses [Kiran et al. 2015]. In the present proposal this architecture is adopted once is designed to process large volume of historical data, as well as rapid incoming data streams.

3. 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. The paper 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, and this aspect it is part of our solution.

[Brown et al. 2019] focus 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. Differently of our proposal, there is no mention about data streaming integration.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. 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), and that proposal does not deal with data streaming integration.

[Dividino et al. 2018] present a semantic data stream integration approach. Stream data sources are pre-mapped in JSON-LD (Json for Linking Data), which is similar to our solution. However, in that work, the authors make use of the C-SPARQL language for querying and for each data source, they create custom mapping rules based on ontology, which can harm the extensibility of the solution.

[Grand et al. 2019] present an approach for scientists execute complex queries without any knowledge of query languages or database structures, and easily integrate heterogeneous data stored in multiple databases through the use of an graphical interface, however that solution does not involve streaming data, only historical databases.

In their systematic review, [de Souza Campos et al. 2021] summarize methods and techniques used in data integration in Big Data, considering data heterogeneity, reviewing techniques that use the concepts of Semantic Web, Cloud Computing, Data Analysis, Big Data, Data Warehouse and other technologies to solve the problem of data heterogeneity. Most of the solutions cited in that systematic review lies on ontology approach for data integration, and problems such as incompatible data and support to real-time data flow are some of the challenges. Two of the reviewed works [Fathy et al. 2019; Ostrowski et al. 2016] use SPARQL as language to submit queries, which differs from our solution which uses SQL. And there are no references to simultaneous integration solutions between rational and non-relational, or historical and stream data sources.

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.

4. 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 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.

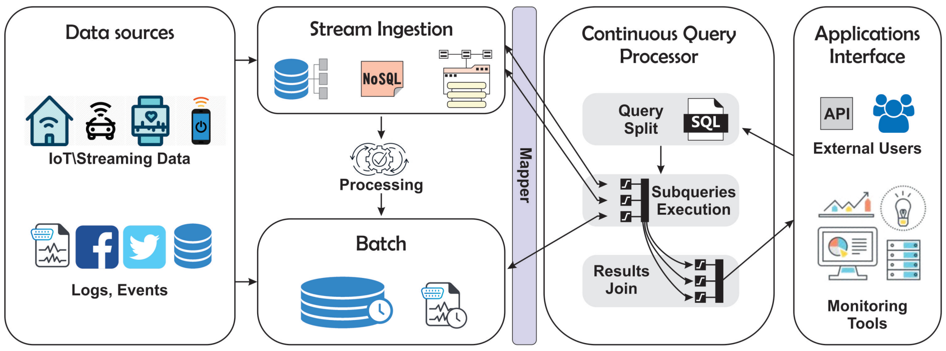


Fig. 1. Framework Architecture

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, classified as Tumbling windows (no overlap) and Sliding windows (with overlap). At this moment, our framework was set up only with Tumbling windows options. 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 is a data processing architecture designed to handle large volumes of data, taking advantage of both 'batch' and 'stream-processing' methods with a hybrid approach. In general, it 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.

We assume that a Stream is a bag (multiset), possibly infinite, of elements. Moreover, these elements have a timestamp associated with them, corresponding to the moment the data source produced it. On the other hand, Historical Data is a standard relational model, i.e., a set of tuples, with no notion of time as far as the semantics of relational query languages are concerned. So, we are assuming that it is a not time-varying relation. Once the Stream and Historical data are joined, the result of this operation will always be a standard relation. In this sense, the framework was not designed to allow a Continuous Query to be executed over another Continuous Query result. Instead, this resulting relation is periodically updated according to the window defined by the user in its continuous query. Thus, this relation is updated at each period (window) based on the Continuous Query re-execution.

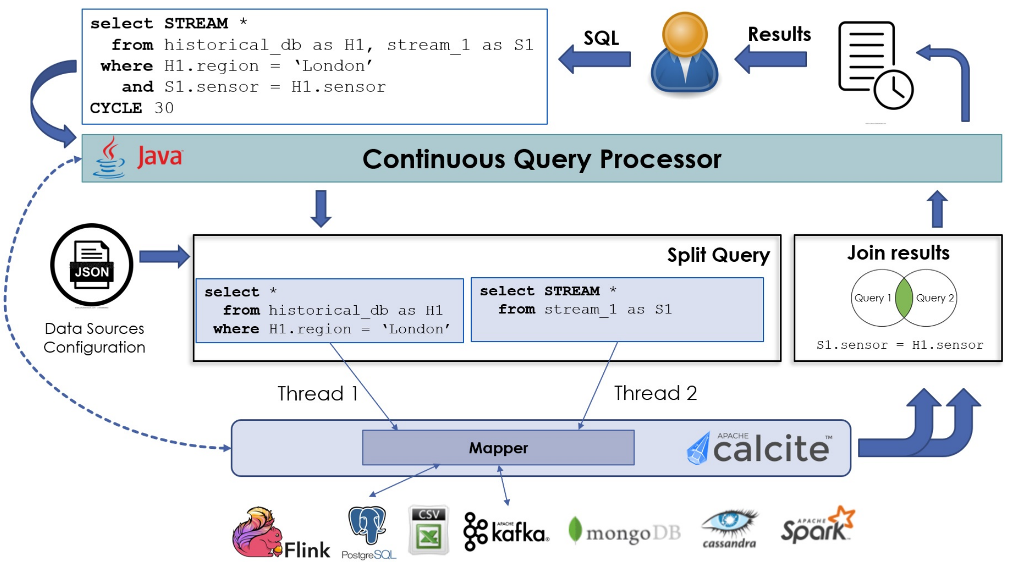
In the **Applications Interface** component, a set of applications can be used to consume both historical and stream data. These applications submit SQL queries and receive the query result in a tabular format, standard SQL.

4.1 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. We use Apache Calcite in this work because it has an adapter designed for extensibility and support for heterogeneous models and data stores (relational, semi-structured, streaming, and geospatial). Calcite supports complex column data types, allowing a hybrid of relational and semi-structured data to be stored in tables. Specifically, columns can be of type ARRAY, MAP, or MULTISET. Furthermore, these complex types can be nested so that it is possible, for example, to have a MAP where the values are of type ARRAY. So when we want to expose, for example, MongoDB data to Calcite, a table is created for each document collection with a single column called _MAP: a document identifier for your data. In Calcite, we can expose this data as a relational table, creating a view after extracting the desired values and converting them to the appropriate type [Begoli et al. 2018].

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.

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



```
select STREAM *
  from historical_db as H1, stream_1 as S1
 where H1.region = 'London'
   and S1.sensor = H1.sensor
CYCLE 30
```

```
select *
  from historical_db as H1
 where H1.region = 'London'
```

```
select STREAM *
  from stream_1 as S1
```

Fig. 2. Implementation Solution

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 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.

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.

[2]https://bitbucket.org/JeffersonAmara/tcc-dcc-2021/src/master/ (Accessed at 06/06/2022)
[3]https://calcite.apache.org/docs/adapter.html (Accessed at 06/06/2022)

```
{
  "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",
          .....
        }
      ]
    }
  ]
}
```

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

5. 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.

5.1 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.

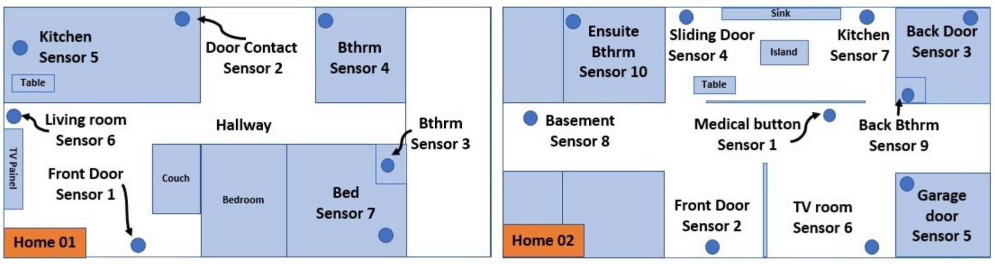


Fig. 4. Layouts of home 01 and home 02.

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 I 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.

Table I. 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

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.

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.

5.2 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?

[4]https://climate.weather.gc.ca/index_e.html (Accessed at 06/06/2022)

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.

Query 1:

```
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%’;
```

Subqueries 1.1 and 1.2 are generated and submitted to Apache Calcite. The specific filters are applied in Subquery 1.1 to reduce the volume of data capture in the stream. On the other hand, the intersection filter (h.dates = s.dates) and the projections defined in the select clause are applied just when the component joins the historical and the stream results.

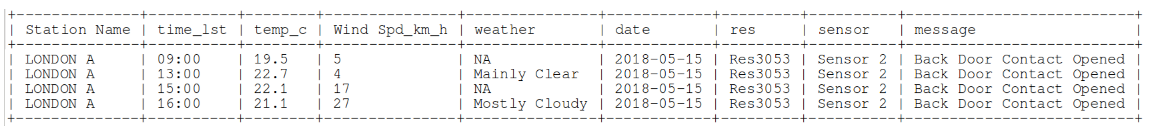
Subquery 1.1:

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

Subquery 1.2:

```
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.

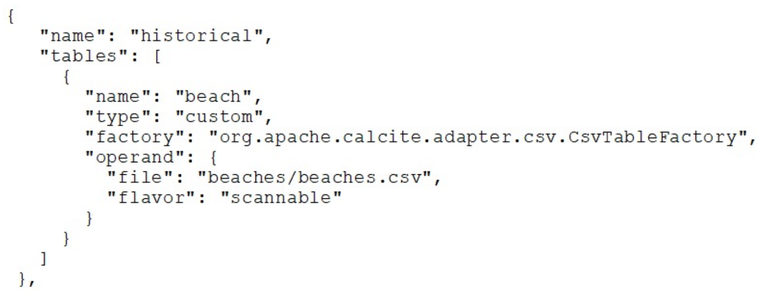


```
+--------------+----------+--------+--------------+---------------+------------+---------+----------+--------------------------+
| Station Name | time_lst | temp_c | Wind Spd_km_h | weather       | date       | res     | sensor   | message                  |
+--------------+----------+--------+--------------+---------------+------------+---------+----------+--------------------------+
| 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 |
+--------------+----------+--------+--------------+---------------+------------+---------+----------+--------------------------+
```

Fig. 5. Results returned by the Continuous Query Processor component (Query 1).

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.

To evaluate the flexibility of using Historical data stored in other formats, we used another dataset in this evaluation, containing data from beaches sensors in Canada. The available data are temperature, beach name, turbidity, and wave height. These data are available in CSV format. So, we needed to configure the JSON to specify it (Figure 6).



```
{
   "name": "historical",
   "tables": [
     {
       "name": "beach",
       "type": "custom",
       "factory": "org.apache.calcite.adapter.csv.CsvTableFactory",
       "operand": {
         "file": "beaches/beaches.csv",
         "flavor": "scannable"
       }
     }
   ]
 },
```

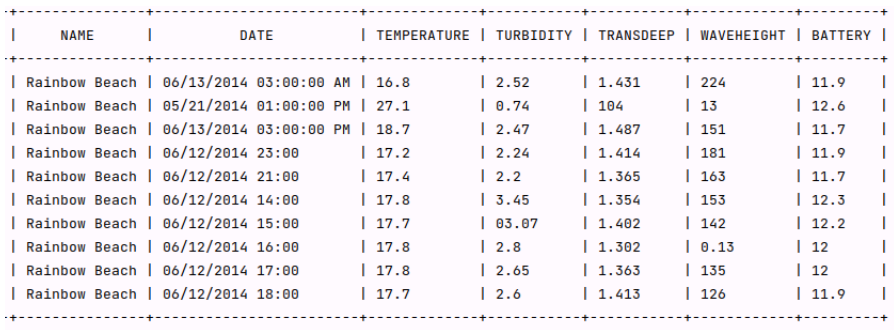
Fig. 6. New schema added to JSON configuration file

We also used another streaming dataset containing data from sensors with information about air quality. We used Kafka to ingest these stream data, so we added dataset information in the JSON following the structure shown in Figure 3.

Query 2 presents a simple query on the historical dataset only where the datasource is a CSV file. Once the main query only involves a historical dataset, there is no subqueries generated. Figure 7 shows part of the results. Although data is stored in a CSV file, it was only necessary to add this information to the JSON (Figure 6), and the framework started to allow queries in SQL format on the dataset.

Query 2:

```
SELECT name, date, temperature,
         turbidity, transdeep, waveheight, battery
  FROM historical.beach AS hb
 WHERE hb.beach = 'Rainbow␣Beach';
```



```
+---------------+----------------------+-------------+-----------+-----------+------------+---------+
|     NAME      |         DATE         | TEMPERATURE | TURBIDITY | TRANSDEEP | WAVEHEIGHT | BATTERY |
+---------------+----------------------+-------------+-----------+-----------+------------+---------+
| Rainbow Beach | 06/13/2014 03:00:00 AM | 16.8      | 2.52      | 1.431     | 224        | 11.9    |
| Rainbow Beach | 05/21/2014 01:00:00 PM | 27.1      | 0.74      | 104       | 13         | 12.6    |
| Rainbow Beach | 06/13/2014 03:00:00 PM | 18.7      | 2.47      | 1.487     | 151        | 11.7    |
| Rainbow Beach | 06/12/2014 23:00     | 17.2        | 2.24      | 1.414     | 181        | 11.9    |
| Rainbow Beach | 06/12/2014 21:00     | 17.4        | 2.2       | 1.365     | 163        | 11.7    |
| Rainbow Beach | 06/12/2014 14:00     | 17.8        | 3.45      | 1.354     | 153        | 12.3    |
| Rainbow Beach | 06/12/2014 15:00     | 17.7        | 03.07     | 1.402     | 142        | 12.2    |
| Rainbow Beach | 06/12/2014 16:00     | 17.8        | 2.8       | 1.302     | 0.13       | 12      |
| Rainbow Beach | 06/12/2014 17:00     | 17.8        | 2.65      | 1.363     | 135        | 12      |
| Rainbow Beach | 06/12/2014 18:00     | 17.7        | 2.6       | 1.413     | 126        | 11.9    |
+---------------+----------------------+-------------+-----------+-----------+------------+---------+
```

Fig. 7. Results returned by the Continuous Query Processor component on historical dataset (Query 2).

Query 3 presents a join between historical (historical_beach) and stream (stream_air) data. Also, it includes a filter by date in the historical data. This query aimed to verify the feasibility of joining a data stream (Kafka) and the data stored in the CSV file.

Query 3:

```
SELECT STREAM *
  FROM historical.historical_beach AS hb,
       kafka.stream_air AS sa
 WHERE hb.DATE = sa.DATE
   AND hb.DATE = '05-28-2014';
```

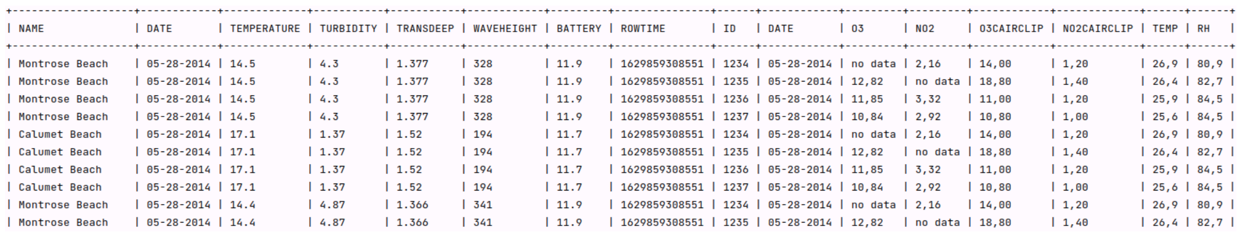
As it involves two different datasets, the following subqueries are generated, one for querying the Historical data and the other for the stream. Figure 8 shows part of the results.

Subquery 3.1:

```
SELECT *
  FROM historical.historical_beach AS hb
 WHERE hb.DATE = '05-28-2014';
```

Subquery 3.2:

```
SELECT STREAM*
  FROM kafka.stream_air AS sa;
```



```
+-------------------+------------+-------------+-----------+-----------+------------+---------+---------------+------+------------+---------+---------+------------+-------------+------+------+
| NAME              | DATE       | TEMPERATURE | TURBIDITY | TRANSDEEP | WAVEHEIGHT | BATTERY | ROWTIME       | ID   | DATE       | O3      | NO2     | O3CAIRCLIP | NO2CAIRCLIP | TEMP | RH   |
+-------------------+------------+-------------+-----------+-----------+------------+---------+---------------+------+------------+---------+---------+------------+-------------+------+------+
| Montrose Beach    | 05-28-2014 | 14.5        | 4.3       | 1.377     | 328        | 11.9    | 1629859308551 | 1234 | 05-28-2014 | no data | 2,16    | 14,00      | 1,20        | 26,9 | 80,9 |
| Montrose Beach    | 05-28-2014 | 14.5        | 4.3       | 1.377     | 328        | 11.9    | 1629859308551 | 1235 | 05-28-2014 | 12,82   | no data | 18,80      | 1,40        | 26,4 | 82,7 |
| Montrose Beach    | 05-28-2014 | 14.5        | 4.3       | 1.377     | 328        | 11.9    | 1629859308551 | 1236 | 05-28-2014 | 11,85   | 3,32    | 11,00      | 1,20        | 25,9 | 84,5 |
| Montrose Beach    | 05-28-2014 | 14.5        | 4.3       | 1.377     | 328        | 11.9    | 1629859308551 | 1237 | 05-28-2014 | 10,84   | 2,92    | 10,80      | 1,00        | 25,6 | 84,5 |
| Calumet Beach     | 05-28-2014 | 17.1        | 1.37      | 1.52      | 194        | 11.7    | 1629859308551 | 1234 | 05-28-2014 | no data | 2,16    | 14,00      | 1,20        | 26,9 | 80,9 |
| Calumet Beach     | 05-28-2014 | 17.1        | 1.37      | 1.52      | 194        | 11.7    | 1629859308551 | 1235 | 05-28-2014 | 12,82   | no data | 18,80      | 1,40        | 26,4 | 82,7 |
| Calumet Beach     | 05-28-2014 | 17.1        | 1.37      | 1.52      | 194        | 11.7    | 1629859308551 | 1236 | 05-28-2014 | 11,85   | 3,32    | 11,00      | 1,20        | 25,9 | 84,5 |
| Calumet Beach     | 05-28-2014 | 17.1        | 1.37      | 1.52      | 194        | 11.7    | 1629859308551 | 1237 | 05-28-2014 | 10,84   | 2,92    | 10,80      | 1,00        | 25,6 | 84,5 |
| Montrose Beach    | 05-28-2014 | 14.4        | 4.87      | 1.366     | 341        | 11.9    | 1629859308551 | 1234 | 05-28-2014 | no data | 2,16    | 14,00      | 1,20        | 26,9 | 80,9 |
| Montrose Beach    | 05-28-2014 | 14.4        | 4.87      | 1.366     | 341        | 11.9    | 1629859308551 | 1235 | 05-28-2014 | 12,82   | no data | 18,80      | 1,40        | 26,4 | 82,7 |
```

Fig. 8. Results returned by the Continuous Query Processor on simple join on date (Query 3).

The previous queries were performed considering a maximum of two data sources. To test a more complex situation, Query 4 involves the join between two historical datasets (historical_beach and historical_climate), one stream dataset (stream_air), and a filter in the streaming dataset.

Query 4:

```
SELECT STREAM *
  FROM historical.historical_beach AS hb,
       historical.historical_climate AS hc
       kafka.stream_air AS sa,
 WHERE hb.DATE = sa.DATE
   AND hc.DATE = sa.DATE
   AND sa."DATE" = '05-28-2014';
```

As Query 4 is executed over three datasets, the following subqueries are generated, and Figure 9 shows part of its results.

Subquery 4.1:

```
SELECT *
  FROM historical.historical_beach AS hb;
```

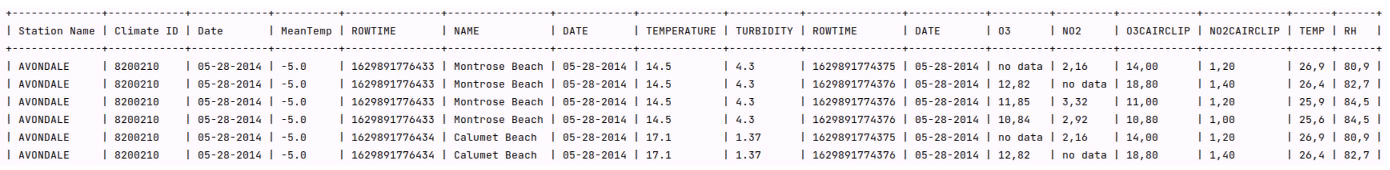
Subquery 4.2:

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

Subquery 4.3:

```
SELECT STREAM *
  FROM kafka.stream_air AS sa
 WHERE sa."DATE" = '05-28-2014';
```

Query processing time varies depending on the data sources, the volume of data returned, and the optimizations implemented in the data sources. In this evaluation, the Continuous Query Processor



```
+--------------+------------+------------+----------+---------------+----------------+------------+-------------+-----------+---------------+------------+---------+---------+------------+-------------+------+------+
| Station Name | Climate ID | Date       | MeanTemp | ROWTIME       | NAME           | DATE       | TEMPERATURE | TURBIDITY | ROWTIME       | DATE       | O3      | NO2     | O3CAIRCLIP | NO2CAIRCLIP | TEMP | RH   |
+--------------+------------+------------+----------+---------------+----------------+------------+-------------+-----------+---------------+------------+---------+---------+------------+-------------+------+------+
| AVONDALE     | 8200210    | 05-28-2014 | -5.0     | 1629891776433 | Montrose Beach | 05-28-2014 | 14.5        | 4.3       | 1629891774375 | 05-28-2014 | no data | 2,16    | 14,00      | 1,20        | 26,9 | 80,9 |
| AVONDALE     | 8200210    | 05-28-2014 | -5.0     | 1629891776433 | Montrose Beach | 05-28-2014 | 14.5        | 4.3       | 1629891774376 | 05-28-2014 | 12,82   | no data | 18,80      | 1,40        | 26,4 | 82,7 |
| AVONDALE     | 8200210    | 05-28-2014 | -5.0     | 1629891776433 | Montrose Beach | 05-28-2014 | 14.5        | 4.3       | 1629891774376 | 05-28-2014 | 11,85   | 3,32    | 11,00      | 1,20        | 25,9 | 84,5 |
| AVONDALE     | 8200210    | 05-28-2014 | -5.0     | 1629891776433 | Montrose Beach | 05-28-2014 | 14.5        | 4.3       | 1629891774376 | 05-28-2014 | 10,84   | 2,92    | 10,80      | 1,00        | 25,6 | 84,5 |
| AVONDALE     | 8200210    | 05-28-2014 | -5.0     | 1629891776434 | Calumet Beach  | 05-28-2014 | 17.1        | 1.37      | 1629891774375 | 05-28-2014 | no data | 2,16    | 14,00      | 1,20        | 26,9 | 80,9 |
| AVONDALE     | 8200210    | 05-28-2014 | -5.0     | 1629891776434 | Calumet Beach  | 05-28-2014 | 17.1        | 1.37      | 1629891774376 | 05-28-2014 | 12,82   | no data | 18,80      | 1,40        | 26,4 | 82,7 |
```

Fig. 9. Results returned by the Continuous Query Processor on multiple join on date (Query 4).

component took a maximum of 0.16 seconds from receiving the SQL query, splitting it, and sending the subqueries to Calcite. Also, the component took a maximum of 0.8 seconds to join the results. Considering the data used in this evaluation, this time is more than sufficient since the time between sensor messages is on the order of seconds to minutes. However, it is important to evaluate these aspects considering other data stream contexts.

Although not all SQL operations are implemented in this framework version, it can be used in scenarios where it is necessary to join data by equality criteria using an inner join. Thus, considering the previous queries, we can verify the feasibility of the solution for its main objective, which is to allow the joining of a stream and historical data using the SQL language.

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 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.* While this is a limitation of the current implementation, we can overcome it by implementing the other join operators. Furthermore, there are no technological restrictions for such an implementation. Therefore, we believe that this question can be answered completely in future versions.

(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.

6. FINAL REMARKS AND FUTURE WORK

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

Akanbi, A. and Masinde, M. 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, 2020.

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

Amará, J., Ströele, V., Braga, R., Dantas, M., and Bauer, M. Stream and historical data integration using SQL as standard language. In *Anais do XXXVI Simpósio Brasileiro de Banco de Dados (SBBD 2021)*. Sociedade Brasileira de Computação - SBC, 2021.

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

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

Begoli, E., Camacho-Rodríguez, J., Hyde, J., Mior, M. J., and Lemire, D. Apache calcite. In *Proceedings of the 2018 International Conference on Management of Data*. ACM, 2018.

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

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

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

Chen, J., Chen, Y., Du, X., Li, C., Lu, J., Zhao, S., and Zhou, X. Big data challenge: a data management perspective. *Frontiers of computer Science* 7 (2): 157–164, 2013.

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

Coulouris, G., Dollimore, J., Kindberg, T., and Blair, G. *Distributed Systems: Concepts and Design*. Pearson Education, 2005.

de Souza Campos, V. V., Brancher, J. D., Farias, F. P., Mioni, J. L. V. M., and Brahim, P. L. G. Review and comparison of works on heterogeneous data and semantic analysis in big data. *Semina: Ciências Exatas e Tecnológicas* 42 (1): 113–128, 2021.

Dividino, R., Soares, A., Matwin, S., Isenor, A. W., Webb, S., and Brousseau, M. *Semantic integration of real-time heterogeneous data streams for ocean-related decision making*. Defence Research and Development Canada= Recherche et développement pour la . . . , 2018.

[5]https://calcite.apache.org/docs/powered_by.html (Accessed at 06/06/2022)

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

Fathy, N., Gad, W., and Badr, N. A unified access to heterogeneous big data through ontology-based semantic integration. In *2019 Ninth International Conference on Intelligent Computing and Information Systems (ICICIS)*. IEEE, pp. 387–392, 2019.

Galhotra, S., Shanmugam, K., Sattigeri, P., and Varshney, K. R. Fair data integration. *arXiv preprint arXiv:2006.06053*, 2020.

Gama, J. A survey on learning from data streams: current and future trends. *Progress in Artificial Intelligence* 1 (1): 45–55, 2012.

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

Golab, L. and Özsu, M. T. Issues in data stream management. *ACM Sigmod Record* 32 (2): 5–14, 2003.

Grand, A., Geda, E., Mignone, A., Bertotti, A., and Fiori, A. One tool to find them all: a case of data integration and querying in a distributed lims platform. *Database* vol. 2019, 2019.

Gurusamy, V., Kannan, S., and Nandhini, K. The real time big data processing framework: Advantages and limitations. *International Journal of Computer Sciences and Engineering* 5 (12): 305–312, 2017.

Jiang, Q. and Chakravarthy, S. *Stream data processing: a quality of service perspective*, 2009.

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

Kolajo, T., Daramola, O., and Adebiyi, A. Big data stream analysis: a systematic literature review. *Journal of Big Data* 6 (1): 1–30, 2019.

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

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

Ostrowski, D., Rychtyckyj, N., MacNeille, P., and Kim, M. Integration of big data using semantic web technologies. In *2016 IEEE Tenth International Conference on Semantic Computing (ICSC)*. IEEE, pp. 382–385, 2016.

Sakr, S., Bajaber, F., Barnawi, A., Altalhi, A., Elshawi, R., and Batarfi, O. Big data processing systems: state-of-the-art and open challenges. In *2015 International Conference on Cloud Computing (ICCC)*. IEEE, pp. 1–8, 2015.

Shan, S., Luo, Y., Zhou, Y., and Wei, Y. 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, 2019.

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

Smys, S. A survey on internet of things (iot) based smart systems. *Journal of ISMAC* 2 (04): 181–189, 2020.

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

Stoyanova, M., Nikoloudakis, Y., Panagiotakis, S., Pallis, E., and Markakis, E. K. A survey on the internet of things (iot) forensics: challenges, approaches, and open issues. *IEEE Communications Surveys & Tutorials* 22 (2): 1191–1221, 2020.

Tan, W.-C. Deep data integration. In *Proceedings of the 2021 International Conference on Management of Data*. pp. 2–2, 2021.

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

Tibco. What is data streaming?, 2019. Acesso em 19 de agosto de 2021.

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

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

Venkatesh, K., Ali, M. J. S., Nithiyanandam, N., and Rajesh, M. Challenges and research disputes and tools in big data analytics. *International Journal of Engineering and Advanced Technology* vol. 6, pp. 1949–1952, 2019.

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

Yousfi, S., Rhanoui, M., and Chiadmi, D. Towards a generic multimodal architecture for batch and streaming big data integration. *arXiv preprint arXiv:2108.04343*, 2021.

Zanella, A., Bui, N., Castellani, A., Vangelista, L., and Zorzi, M. Internet of things for smart cities. *IEEE Internet of Things journal* 1 (1): 22–32, 2014.