IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received February 29, 2020, accepted March 11, 2020, date of publication March 23, 2020, date of current version April 2, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.2982472

MusQ: A Multi-Store Query System for IoT Data Using a Datalog-Like Language

HANI RAMADHAN[1], FITRI INDRA INDIKAWATI[2], JOONHO KWON[3], (Member, IEEE), AND BONYONG KOO[4]

[1]Department of Big Data, Pusan National University, Busan 46241, South Korea
[2]Department of Informatics Engineering, Ahmad Dahlan University, Yogyakarta 55166, Indonesia
[3]School of Computer Science and Engineering, Pusan National University, Busan 46241, South Korea
[4]School of Mechanical System Engineering, Kunsan National University, Gunsan 54150, South Korea

Corresponding author: Joonho Kwon (jhkwon@pusan.ac.kr)

This work was supported in part by the Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education under Grant NRF-2017R1D1A1A09000706, and in part by the National Research Foundation of Korea (NRF) grant funded by the Korean Government (Ministry of Science and ICT) under Grant 2018R1A5A7059549.

ABSTRACT The growing number of connected Internet of Things (IoT) devices has increased the necessity for processing IoT data from multiple heterogeneous data stores. IoT data integration is a challenging problem owing to the heterogeneity of data stores in terms of their query language, data models, and schemas. In this paper, we propose a multi-store query system for IoT data called MusQ, where users can formulate join operation queries for heterogeneous data sources. To reconcile the heterogeneity between source schemas of IoT data stores, we extract a global schema from local source schemas semi-automatically by applying schema-matching and schema-mapping steps. In order to minimize the burden on the user to understand the finer details of various query languages, we define a unified query language called the multi-store query language (MQL), which follows a subset of the Datalog grammar. Thus, users can easily retrieve IoT data from multiple heterogeneous sources with MQL queries. As the three MQL query-processing join algorithms are based on a mediator–wrapper approach, MusQ performs efficient data integration over significant volumes of IoT data from multiple stores. We conduct extensive experiments to evaluate the performance of the MusQ system using a synthetic and large real IoT data set for three different types of data stores (RDBMS, NoSQL, and HDFS). The experimental results demonstrate that MusQ is suitable, scalable, and efficient query processing for multiple heterogeneous IoT data stores. Those advantages of MusQ are important in several areas that involve complex IoT systems, such as smart city, healthcare, and energy management.

INDEX TERMS Data management and analytics, Internet of Things, multi-store system, query processing, schema integration.

I. INTRODUCTION

The proliferation of Internet of Things (IoT) technology has led to a rapid deployment of a massive number of IoT devices [1]. The number of interconnected devices is still increasing and is expected to reach 41.6 billion by 2025 [2]. As this number grows, the amount of IoT data will grow accordingly. Thus, much of the effort in research and technology in this area [3], [4] focuses on gaining full value from IoT by trying to efficiently manage the massive amounts of IoT data.

The associate editor coordinating the review of this manuscript and approving it for publication was Luigi de Russis.

However, complex IoT systems in areas such as smart cities [5], healthcare [6], and energy management [7] require processing IoT data from multiple heterogeneous data stores [8], [9]. However, the heterogeneity of data sources is identified as a challenging problem in building IoT Big Data frameworks [4]. In general, integrating data from multiple data sources requires three steps: (1) retrieve data from each individual data store, (2) determine the relationships between the multiple data sources, and (3) merge the intermediate results from each data source into a final result [10]. Thus, if a user wants to access and combine IoT data from multiple sources, the user should have in-depth knowledge about each data store. Specifically, the user is required to write queries in

various languages and understand the mechanism of merging intermediate results. Without sufficient knowledge, the user may have difficulty finding the ad hoc relationship between the data stores [4].

These problems in IoT data integration have motivated many researchers to build *multi-store systems* [11]. Prior work on multi-store systems can be divided into four categories: (1) the *warehousing* approach [12], (2) the *federated* or *virtual integration* approach [13]–[16], (3) the specialized approach [17], and (4) the schema-less approach [18], [19]. The warehousing approach integrates data by materializing combined data into one storage repository called a *warehouse*. This approach provides easy and fast access to the integrated data because there is only one physical data store. However, this approach is not suitable for IoT applications because of the high cost of materializing and synchronizing massive volumes of IoT data. The federated approach requires a global schema, which provides a unified view by concealing the heterogeneity of the data sources. Existing multi-store systems [13] require users to manually define the global schema, which requires a deep understanding of the relationships between local source schemas. The specialized approach includes Cloud IoT [17], which provides integrated access that hides the heterogeneity of the data stores using a Data Access Component (DAC). This removes the burden of defining different connections and query languages for each data store. However, it lacks join operations over different data stores, since it does not create a warehouse nor a global schema. By contrast, the schema-less approach [18], [19] provide access to heterogeneous multi-store systems without schema. However, this approach requires the user to write the details of the data stores in the query.

To build a useful multi-store system for IoT data, we investigate three key features by addressing three critical challenges: (1) constructing a global schema from local source schemas by exploiting relationships, as in the federated approach; (2) performing complex queries on multiple data stores without knowing all the different query languages; and (3) efficiently executing user queries, especially join operations, to retrieve relevant data from local sources and merging them into a final result.

To address these challenges, we propose a multi-store query system called MusQ, which uses the federated approach. First, we devise a process for the semi-automatic construction of global schema from the local sources. The global schema construction consists of two steps: schema matching and schema mapping. The schema-matching step computes all possible combinations of equivalence attributes (semantic matches) by considering both the similarity and the meanings of the attributes of the local schemas. The schema-mapping step extracts a set of mapping definitions by further refining the semantic matches. Second, to provide a unified query language, we propose a Datalog-like query language called multi-store query language (MQL). We choose the Datalog [20] language as the basis of MQL because it is a simple and logical language, which allows for easy translation into other query languages. MusQ can execute complex MQL join queries by utilizing the constructed global schema. Third, for efficient integrated access to multiple heterogeneous data stores, we suggest three query-processing methods that are focused on join operations, based on a mediator–wrapper approach.

MusQ's three main features are designed to assist users whose tasks are related to querying in a multi-store IoT system. A user may only have limited information about each data store's data model, query language, and schema. Since MusQ supports different types of heterogeneous systems such as relational databases (MySQL [21]), NoSQL databases (Cassandra [22] and MongoDb [23]), and the Hadoop Distributed File System (HDFS), it aids the user by removing the burden of manually defining the global schema from each local source. In addition, the user of MusQ can simply use the unified query language, MQL, to perform the query in the multi-store system. This minimizes the necessity of a deep understanding of different query languages when developing IoT data applications.

The main contributions of this study are summarized as follows:

- We propose a semi-automatic global schema construction method to replace the process of manually defining the global schema, which is a tedious task in building IoT data applications when different types of heterogeneous IoT data stores are supported. This semi-automatic construction incorporates correspondence between the local schema relationships using two steps: (1) a schema-matching step and (2) a schema-mapping step.
- For an easier representation under multiple heterogeneous data stores, we provide a formal unified query language (MQL) that is extensible to each IoT data store's query language. The MQL language is also used to define the elements of the global schema for conformity and simplicity.
- To efficiently perform data integration over substantial volumes of IoT data from multiple stores, we implement three MQL query-processing join algorithms based on a mediator–wrapper approach: a nested-loop join approach, a hash join approach, and a sort-merge join approach. We also provide the proof of correctness and the cost analysis of the proposed approaches.
- To evaluate the performance and suitability of the MusQ system, we conduct extensive experiments using both a synthesized and a large real IoT data set [24] with different scenarios. The experimental results demonstrate that MusQ is scalable and effective for heterogeneous IoT data stores.

The remainder of this paper is organized as follows. Section II provides an overview of related work. In Section III, we discuss the overall architecture of the proposed MusQ system. Section IV and Section V explain how to construct a global schema from local schemas and how to execute a user query specified in MQL, respectively. We discuss the results of the performance evaluation of the

MusQ system in Section VI. Finally, Section VII concludes the paper.

II. RELATED WORK

The growing necessity of accessing multiple data stores in various applications has led to extensive research on IoT data integration. The multi-store systems for IoT Big Data require the capabilities of supporting various database systems, such as relational and non-relational databases, and processing queries seamlessly [4], [16]. In this section, we briefly survey existing approaches and highlight how they differ from our approach. We classify these approaches into two broad categories: (1) theoretical schema integration and (2) multi-store query systems.

A. SCHEMA INTEGRATION

Conventional data integration requires both a global schema and several local schemas. The schema integration is performed in two steps: schema matching and schema mapping [25]. In the schema-matching step, equivalent attributes between a local schema and a global schema are obtained by a *matcher*. In the case of different levels of heterogeneity, the schema matching can incorporate several matchers [26]. For example, a schema-level matcher can match attribute names and data types, and an instance-level matcher can perform pair-wise comparisons of instance values or data. The schema-mapping step transforms a set of matches into a mapping definition, which is a query expression that shows the relation between local and global schemas.

The schema matching and mapping reduce the user's effort in understanding the relations between the local schemas. When working with the structured data, such as in a Relational Database Management System (RDBMS), deriving a schema can be a simple process. However, for unstructured data, deriving a schema is more complicated as the unstructured data might not have a defined schema [14]. Manual definition of a global schema requires the user to check each local schema's attribute and match it to another attribute from the other local schemas. To guarantee this match is appropriate, the user needs to check the data store's structure and contents. Thus, a user needs to deeply understand the data store's structure, content, and other features that support data store matching. This problem can become even more difficult if, while working with a multi-store system, a new unstructured data store is introduced to the user. The user is then required to gain more knowledge and build a global schema for this new data store.

Several approaches have been developed for data integration using schema-level matching [27], [28]. For example, Cupid [27] attempts multiple matching methods at the schema level to produce mappings between schema elements using attribute names, data types, constraints, and the schema structure. Based on graph-like structure, Cupid performs schema-mapping generation with various shortcomings, such as missing elements, nested structure, non-matching data types, and different element names. Furthermore, Cupid still needs to perform the mapping of a local schema to a larger (global) schema. In addition, Clio [28] uses an iterative *integration-by-example* method that allows the user to specify and choose potential schema mappings. Clio provides the mapping between two schemas: a source schema and a target schema. However, the target schema in Clio is similar to a global schema. This means that Clio needs a global schema as prerequisite.

Data values can provide useful insights into the meanings of schema elements. Thus, a number of approaches in data integration exploit instance-level matching [16], [29], [30]. Falcon [29] and Coma 3.0 [30] are the latest works to combine schema-level matching with instance-level matching. Falcon's matching exploits the linguistic similarity between two values combined with the system's ontology-based structure. The linguistic similarity in Falcon immensely impacts the correctness of the schema matching. Furthermore, COMA, in terms of ontology, provides an intuitive visual interface for users to examine its generated schema matches. Users can inspect the possible matches and then verify them as a valid global schema. However, both systems focus on ontology matching for web applications and still require a pre-supplied global schema.

Another approach [16] focuses on the integration of a heterogeneous multi-store system of IoT. This approach models the global schema of IoT using the contexts from Wikipedia articles, local schemas, and data from the connected sensors. As a result, it gives several mappings as an information fusion architecture to support the heterogeneity and interoperability of IoT data. However, this procedure requires external semantic knowledge rather than information on the data stores.

All the previous works on schema integration require specifying a global schema manually and calculating matches between local schemas and the global schema. Furthermore, the existing multi-store system [13] does not implement modules for constructing the global schema. However, MusQ avoids these prerequisites because it does not initially require a global schema. To the best of our knowledge, there is still no work on schema integration that recommends a global schema using the definitions of local schemas only. The global schema recommendation avoids the manual definition of global schema in which the user is required to check all possible matches of each local schema. Therefore, we consider the global schema recommendation as the semi-automatic way to build the global schema (which we discuss in Section IV) because it can support heterogeneity in multiple IoT data stores.

B. QUERY PROCESSING IN MULTI-STORE SYSTEMS

Data integration on multiple databases is well documented in the database research literature [25], [31]. Due to increasing demand, especially from enterprises, it has begun to attract increasing attention in recent years [32]. Existing multi-store systems can be broadly categorized into four approaches: (1) the *warehousing* approach [12], [33], (2) the *federated* or *virtual integration* approach [13]–[15], [34]–[37],

TABLE 1. Existing multi-store systems.

Name	Architecture	Query Language	Sup. Data stores
Data Adapter [12]	Warehouse	Hbase query	HBase, RDBMS
P2P-based integration [39]	Federated/P2P	FOL query	-
Polybase [35]	Federated/Mediator–wrapper	SQL-like	HDFS, RDBMS
Big Integrator [13]	Federated/Mediator–wrapper	SQL-like	BigTable, RDBMS
CloudMdsQL [14]	Federated	SQL-like with native subqueries	RDBMS, NoSQL,HDFS
Cloud IoT [17]	Specialized	DAC, native subqueries	RDBMS, NoSQL (separately)
Drill [19]	Schema-less	ANSI SQL	RDBMS, NoSQL, HDFS
Presto [18]	Schema-less	ANSI SQL	RDBMS, NoSQL, HDFS
MusQ	Federated/Mediator–wrapper	Datalog-like	RDBMS, NoSQL, HDFS

(3) the specialized approach [17], and (4) the schema-less approach [18], [19].

In the warehousing approach, data residing in different sources are combined and materialized into a single storage repository, called a warehouse. This materialization process focuses on the extraction-transformation-load (ETL) to the warehouse, which may incur an enormous cost in proportion to the level of data granularity and the summary level. Thus, when performing updates, deletions, or insertions to the data sources, the warehousing approach needs to synchronize the data sources and the warehouse [38].

Liao *et al.* [12] propose three data-synchronization mechanisms to overcome the problem of inconsistent data: *blocking transformation* (BT), *blocking dump* (BD), and *direct access* (DA). Odyssey [33] uses data life cycle management (DLM) to perform data transformations, including type conversions, file formatting, and data distribution on Hadoop. However, these approaches incur a high cost in the materialization and synchronization processes when the warehouse maintains large volumes of data and when data sources are frequently updated.

In the second approach, federated systems provide a unified query interface for data residing in different data stores, and are based on a mediator–wrapper architecture [13], [15], [18], [19], [35]. These systems can access data directly from their original database through a unified view, called a *mediated schema* or a *global schema*. The federated approaches also facilitate a declarative *mapping language* that explicitly specifies the relationship between a global schema and local schemas. The correspondence between local and global schemas can be expressed using two basic approaches: *global-as-view (GAV)* [34]–[36] and *local-as-view (LAV)* [13]. The *GAV* approach describes a global schema as a combination of all local schemas, whereas the *LAV* approach expresses local schemas as a function of the global schema. Both approaches can retrieve results from user queries specified in the global schema. Recently, a new solution, CloudMdsQL [14], that does not follow the mapping approaches of GAV or LAV was proposed. CloudMdsQL uses an ad hoc schema extracted from the table names in subqueries.

In the specialized approach, Cloud IoT [17] provides a heterogeneous integration access for IoT data stores using a Data Access Component (DAC) to hide the complexity of accessing heterogeneous local sources. However, it only demonstrates its feasibility on a filter query and a limited number of data store types (NoSQL or RDBMS) at a time. Cloud IoT does not support join operations on different data stores, and thus it can only can perform the query separately on each data store. Hence, performing queries that include join and/or aggregation operations under a heterogeneous multi-store system is still of interest. Other specific heterogenous data store integration systems using the specialized approach can be found in [11].

The schema-less approaches [18], [19] do not necessarily use schema to perform queries across multiple databases. These approaches use an ANSI SQL-based language for the heterogenous data stores. Drill [19] provides schema-less integration with columnar execution, but does not support Cassandra by default. Presto [18] is a distributed SQL query engine and also supports querying on various data stores with columnar execution. However, Drill and Presto require the user to define the details of the tables in the query, as they perform queries without global schema. Thus, the developers are required to know the specific location, table name, and mapping of each data store.

Table 1 compares our approach to several existing multi-store systems according to the architecture, query language, and supported data stores. MusQ exploits the mediator–wrapper architecture, and supports integrated query processing for RDBMS, NoSQL, and DFS data stores as well as for IoT and non-IoT data. Unlike the other approaches, MusQ uses a Datalog-like language that is easily translatable to another query language. Furthermore, MusQ supports three different data store types, which are traditional RDBMS, NoSQL (MongoDB & Cassandra), and HDFS.

MusQ can execute join operations for different data stores at the same time, unlike Cloud IoT, which can only execute queries on one data store at a time. Although Drill and Presto can solve similar tasks to MusQ, MusQ does not use the columnar style execution that is used by Drill and Presto. Also, MusQ does not concentrate on several concepts of Presto such as data partitioning, spill-to-disk operation, and interconnected stage operations. In addition, MusQ also applies the GAV approach by providing the recommendations of a global schema from the source schema.

III. THE DESIGN OF MusQ

In this section, we present the novel design of our multi-store query processing system (MusQ).

A. SYSTEM ARCHITECTURE OVERVIEW

The overall architecture of MusQ is depicted in Figure 1.

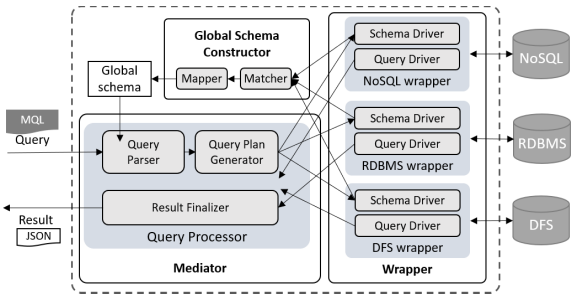


FIGURE 1. Overview of MusQ.

MusQ consists of three key components: (1) the global schema constructor module, (2) the mediator module, and (3) the wrapper module. The global schema constructor semi-automatically generates the global schema to hide the heterogeneity of local source schemas and supports complex join queries over multiple stores. The main goal of the mediator module is to handle user queries written using the Multistore Query Language (MQL). After parsing a user query, the mediator splits the query into subqueries based on the global schema. It also creates a logical query plan to manage the flow of query executions for each subquery. The wrapper module of each data store translates each subquery into the respective query language, executes the translated subquery, and retrieves an intermediate result. The result of each wrapper is sent back to the mediator and then merged into a final result in JSON (JavaScript Object Notation) [40] format.

These features of MusQ effectively reduce the burden on users to manually define the global schema and to remember the details of various query languages.

This is particularly relevant for IoT, for which it is necessary to handle the heterogeneity of multiple data stores with various query languages and attributes.

We explain the detailed steps of the global schema constructor module and the mediator and wrapper modules in Sections IV and V-A, respectively.

B. MULTI-STORE QUERY LANGUAGE

MusQ is designed to deal with user queries and to provide integrated access to multiple heterogeneous stores. Thus, we define a multi-store query language called *MQL* as a unified query language, which conforms to a subset of the grammar of the Datalog language [20]. Datalog is chosen as the basis for MQL because of its logic-based language, which provides easy conversion to each wrapper's query language. In MusQ, the MQL grammar is used to specify user queries as well as global schema. During query processing, MQL queries (user queries) are used by the mediator to communicate with the wrappers. The wrapper module then converts the MQL queries to each data store's query language.

```
queryname (column [, column] [,
aggregate(column)] [,...]) :-
  table (column [, column] [,...])
  [AND table(column [, column] [,...])]
  [AND filters]
```

FIGURE 2. Grammar of MQL queries.

Figure 2 shows the query syntax for MQL. The head of MQL (the syntax before ":-") describes the query definition, and the body (the syntax after ":-") explains the source definition and query filters. The query definition describes the user-specified query name and selected attributes or columns. We can define an aggregate function, such as SUM, AVG, MAX, MIN, and COUNT, in one of the query attributes. The source definition in the query body provides the table and the attributes defined in the global schema. Query filters specify a search condition using a comparison operator (=, <, >, <=, and >=), which returns tuples that meet the terms of the search condition.

The MQL query is expressed using the table and attribute definitions in the global schema. It uses an implicit join, which aids users in focusing on *what* to query, rather than *how* to perform the query. Thus, users do not need to provide detailed information about joins, such as the locations of tables, in an MQL query.

```
Q1(SID,heartrate,serialNumber,empID,name) :-
  wearable(SID,serialNumber,model,
   purchaseDate,type,dt,pos,heartrate,tm)
  AND empSensor(empID,name,jobID,sensorID)
  AND heartrate > 100
  AND type = 'wearable'
```

FIGURE 3. An example MQL query.

Example 1: Figure 3 shows an example of an MQL query. The query $Q1$ combines data from the `wearable` and `empSensor` schemas, which are defined in the global schema shown in Figure 10. The query retrieves only wearable sensor data that reports a heart rate of more than 100 using a comparison operator on the `heartrate` and `type` attributes.

MusQ supports various types of MQL queries, such as (1) filter queries, (2) aggregation queries, and (3) join queries. IoT analytics commonly use the combination of these three query types. The aggregation and filter queries are important for generating the summary of batch IoT data, whereas the join query is necessary for merging results from data stores.

IV. CONSTRUCTING THE GLOBAL SCHEMA

In this section, we explain how to construct a global schema semi-automatically from disparate source schemas in order to

TABLE 2. Data source schemas.

ID	Data Store	Database Name	Schema Definition
DB1	Cassandra	Sensor	`readings(sid, dt, tm, heartrate, pos)`
DB2	MySQL	HR	`employee(empID, name, address, DoB, email, jobID)`
DB3	MySQL	Inventory	`sensor(SID, serialNumber, model, purchaseDate, type)`
DB4	MySQL	Inventory	`wearSensor(employeeID, sensorID, status)`
DB5	HDFS	Batch	`batchwearable(SID, date, avgRate, minRate, maxRate)`

perform schema integration. The construction consists of two steps: (1) a schema-matching step and (2) a schema-mapping step. The schema-matching step finds the correspondences, or semantic matches, between source schemas. A semantic match is accompanied with a similarity score, which measures the likelihood of a match to be included in the global schema. MusQ recommends these semantic matches to the user. Thus, the user needs to check only a small set of matches with a high similarity score as the preferred global schema. The schema-mapping step transforms semantic matches into mapping definitions for constructing the global schema. Hence, we define this process as the semi-automatic global schema construction. This process eliminates the need for the user to examine all possible matches, including the unlikely ones. This is also useful when a new data store is suggested because the user does not need to examine its structure and contents to define the match from the new data store.

A. EXAMPLE SCHEMAS IN AN IoT ENVIRONMENT

Before describing how to construct our global schema, we briefly describe an example of source schemas, which we use to explain the schema integration in this paper.

We assume a scenario that combines sensor data stored in Cassandra and HDFS with static data stored in MySQL. This scenario is one of many possible applications in an IoT environment. Table 2 depicts the local source schemas for multiple data stores in this IoT case study. A `readings` table or column family in Cassandra keeps sensor data from wearable devices and the positions of employees for a given time and date. In this case, a wearable device continuously measures heart rate values. An `employee` table in the HR database stores information about all employees, including employee numbers, names, addresses, dates of birth, e-mail addresses, and job levels. An inventory department stores information from all sensor devices in tables such as `sensor` and `wearSensor`. The `sensor` table maintains data on all sensor devices, such as the sensor identification number, serial number, model, date of purchase, and the type of sensor. The `wearSensor` table keeps track of the owners of wearable devices listed in the `sensor` table. A `batchwearable` file in HDFS stores the daily batch-processed data from the `readings` table, which consists of the average, minimum, and maximum heart rate values from each sensor for each day.

B. GLOBAL SCHEMA CONSTRUCTOR

As explained in Section III, the global schema constructor of MusQ integrates disparate local source schemas to obtain a global schema. Figure 4 illustrates the two main components of the global schema constructor: (1) schema matching and (2) schema mapping.

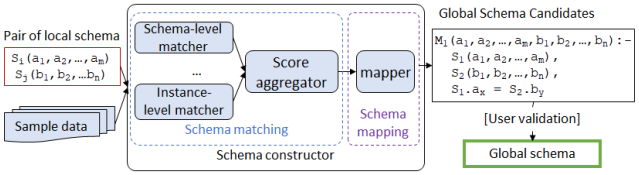


FIGURE 4. Global schema constructor.

1) SCHEMA MATCHING

Before describing the schema integration in greater detail, we provide formal definitions of the crucial concepts used in our global schema constructor module.

The formal definition of a source schema is as follows:

Definition 1: (Source Schema) A source schema S is a collection of tables and is denoted by $S = \{T_1, T_2, \cdots, T_n\}$. Each table T can have several attributes, represented by $T(a_1, a_2, \cdots, a_m)$.

Example 2: Figure 5 shows an example of source schema definitions using the scenario from Table 2. The data stores of S_1, S_2, S_3, and S_4 are Cassandra, MySQL, MySQL, and HDFS, respectively.

```
T1 = readings(sid, dt, tm, heartrate,
pos)
T2 = employee(empID, name, address, DoB,
email, jobID)
T3 = sensor(SID, serialNumber, model,
purchaseDate, type)
T4 = wearSensor(employeeID, sensorID,
status)
T5 = batchwearable(SID, date, avgRate,
minRate, maxrate)
S1 = {T1}
S2 = {T2}
S3 = {T3, T4}
S4 = {T5}
```

FIGURE 5. Example of source schemas.

The goal of schema matching is to find a correspondence between source schemas, called a *semantic match*. A correspondence specifies how some elements of two source schemas are related to each other. This reduces the effort in defining the global schema and understanding the relationship between each element of two source schemas from the

available local schemas. For example, the correspondence between the `empID` column in `employee` from the local schema S_1 and the `employeeID` of the `wearSensor` table from the local schema S_2 could be found from multiple data sources.

Definition 2: (Similarity Score) A similarity score is denoted by $sc_{ij} = (S_1.a_i, S_2.b_j, simScore)$, where a_i and b_j are matching candidates from the source schemas S_1 and S_2, respectively, and *simScore* is the similarity score between attributes a_i and b_j ($0 <= simScore <= 1$).

Example 3: Given the pair of schemas in Figure 5, the schema matcher calculates a similarity score for each combination of attributes from schemas S_1 and S_2. Figure 6 shows a sample of the similarity scores for the attributes of S_1 and S_2.

```
sc00 = (employee.empID,
wearSensor.employeeID,0.54)
sc01 = (employee.empID,
wearSensor.sensorID,0.22)
sc10 = (employee.jobID,
wearSensor.employeeID,0.23)
...
sc51 = (employee.email,
wearSensor.sensorID,0.18)
```

FIGURE 6. Similarity scores for the attributes of S_1 and S_2.

The next step is to define a semantic match from the aggregated similarity score. A semantic match is defined as follows:

Definition 3: (Semantic Match) A semantic match is a *one-to-one* match between an attribute of S_i and an attribute of S_j, so that $S_i.a_x \approx S_j.b_y$. A semantic match has the highest similarity score, which means that attribute a_x from S_i is very likely to be related to attribute b_y from S_j.

We have the highest similarity score for `empID` and `employeeID` using the previous example. Thus, *employee.empID* $\approx$ *wearSensor.employeeID* is a *semantic match* for S_1 and S_2. In other words, the `empID` and `employeeID` attributes are likely to be related, and we can perform a join operation between the two schemas using these attributes as the join condition. We can identify other schema semantic matches using the same schema-matching process. The result of the semantic matching is shown in Figure 7.

```
employee.empID ≈ wearSensor.employeeID
readings.sid ≈ sensor.SID
sensor.SID ≈ batchwearable.SID
sensor.SID ≈ wearSensor.sensorID
```

FIGURE 7. Examples of semantic matches.

Algorithm 1 outlines the steps in the schema-matching process. The schema-level matcher checks for similarities in the attribute names and data types in line 5. Attribute-name similarity is calculated using a string similarity function (such as Jaccard similarity) and semantic similarity by utilizing the

Algorithm 1 Schema Matching

Input: Schema List S
Output: Similarity Score sc
procedure `SchemaMatching`(S)
1: Initialize a set of SimilarityScore SC;
2: **for** i *from* 0 *to* $|S| - 1$ **do**
3: **for** j *from* $i + 1$ *to* $|S|$ **do**
`// Calculate schema-level similarity score`
4: Initialize new SimilarityScore ss as a tuple of (s_i, s_j, *simScore*);
5: $ss \leftarrow$ `SchemaLevelMatch`(s_i, s_j);
`// Calculate instance-level similarity score`
6: Initialize new SimilarityScore is as a tuple of (s_i, s_j, *simScore*);
7: $is \leftarrow$ `InstanceLevelMatch`(s_i, s_j);
`// Calculate weighted average of schema level and instance level score`
8: $avgscore \leftarrow$ `GetWeightedAverage`(ss, is);
9: Initialize new SimilarityScore *score* as a tuple of (s_i, s_j, *avgscore*);
10: SC.`Add`(*score*);
end for
end for
11: SC.`SortDesc`();

WordNet lexical database [41]. In line 7, the instance-level matcher checks for similarities in the sample data from schemas S_1 and S_2 using the schema definition in Table 2. Subsequently, in line 8, the similarity scores from both matchers are aggregated using a weighted average calculated by the *score aggregator*.

To prove the correctness of Algorithm 1, we use the loop invariant technique [42]. This approach examines the correctness of the algorithm in three loop stages: (1) initialization, (2) maintenance, and (3) termination.

Theorem 1: With the schema list S as input, algorithm Schema Matching (Algorithm 1) is correct with these loop invariants: for any step in the outer loop, we apply this step: $0 \leq i < |S| - 1$, otherwise $i = |S| - 1$; for any step in the inner loop, we apply this step: $i < j < |S|$, otherwise $j = |S|$.

Proof 1 (Proof of Schema Matching) The algorithm generates a set SC of tuples $(s_i, s_j)|s_i, s_j \in S \times S, i < j$ from schema S with their corresponding weighted average similarity score. The weighted average similarity score is a function that computes the similarity score from the schema-level and instance-level matching as described by line 5 and line 7. Both matching processes also take the same input pair of schema (s_i, s_j), and the weighted average similarity computation takes the output of both matching processes as input. Hence, we can simply denote the weighted average similarity computation as $f_{avgscore} : (s_i, s_j) \rightarrow (s_i, s_j, avgscore)$. Then the algorithm

sorts the tuples in descending order according to the weighted average similarity score.

(Outer Loop) Initialization: The similarity score result set is initialized as $SC = \varnothing$ and $i = 0$. No value for j is declared yet. Thus, the result set should be empty.

(Outer Loop) Maintenance: Suppose at the beginning of the iteration, the invariant holds at a particular value $i < |S| - 1$ and the SC contains the result set from the beginning until $i - 1$, as described in Equation 1. Let SC' and i' denote the content of SC and i at the end of the iteration, as described in Equation 2 below:

$$\begin{aligned} SC &= \{(s_k, s_j, avgscore) | 0 \leq k < i \wedge k < j < |S| \\ &\quad \wedge f_{avgscore}(s_k, s_j) = (s_k, s_j, avgscore)\} \quad (1) \\ SC' &= SC \cup \{(s_i, s_j, avgscore) | i < j < |S| \\ &\quad \wedge f_{avgscore}(s_i, s_j) = (s_i, s_j, avgscore)\}; \\ i' &= i + 1. \quad (2) \end{aligned}$$

At the end of the iteration, SC' must contain the set of tuples computed by $f_{avgscore}$ from S, from the beginning until i, and it must hold the invariant. We prove this invariant in Equation 3:

$$\begin{aligned} SC' &= SC \cup \{(s_i, s_j, avgscore) | i < j < |S| \\ &\quad \wedge f_{avgscore}(s_i, s_j) = (s_i, s_j, avgscore)\} \\ SC' &= \{(s_k, s_j, avgscore) | 0 \leq k < i \wedge k < j < |S| \\ &\quad \wedge f_{avgscore}((s_k, s_j)) = (s_k, s_j, avgscore)\} \cup \\ &\quad \{(s_i, s_j, avgscore) | i < j < |S| \wedge f_{avgscore}(s_i, s_j) \\ &\quad = (s_i, s_j, avgscore)\} \\ SC' &= \{(s_k, s_j, avgscore) | 0 \leq k < i + 1 \wedge k < j < |S| \\ &\quad \wedge f_{avgscore}((s_k, s_j)) = (s_k, s_j, avgscore)\} \\ SC' &= \{(s_k, s_j, avgscore) | 0 \leq k < i' \wedge k < j < |S| \\ &\quad \wedge f_{avgscore}((s_k, s_j)) = (s_k, s_j, avgscore)\} \\ &\quad (proved). \quad (3) \end{aligned}$$

Next, we prove the invariant in the inner loop of Algorithm 1 to find the match between s_i with the elements of S after i until the end.

(Inner Loop) Initialization: At the beginning of the inner loop iteration, s_i and SC from the outer loop are given. The similarity score set of the loop is denoted as SC^i. Then the value of j is initialized as $j = i + 1$. This invariant still holds because the similarity score $f_{avgscore}(s_i, s_j)$ is not computed yet.

(Inner Loop) Maintenance: Suppose at the beginning of the iteration, the invariant holds a particular value of $j < |S|$, and SC^i contains the union of SC and the result of $f_{avgscore}$ from $i + 1$ to $j - 1$, as defined in Equation 4. Let $SC^{i'}$ and j' denote the content of SC^i and j at the end of the iteration. Equation 5 describes $SC^{i'}$ and j', and this invariant is proven in Equation 6:

$$\begin{aligned} SC^i &= SC \cup \{(s_i, s_l, avgscore) | i < l < j \\ &\quad \wedge f_{avgscore}(s_i, s_l) = (s_i, s_l, avgscore)\} \quad (4) \end{aligned}$$

$$\begin{aligned} SC^{i'} &= SC^i \cup \{(s_i, s_j, avgscore) | f_{avgscore}(s_i, s_j) \\ &\quad = (s_i, s_j, avgscore)\}; \\ j' &= j + 1 \quad (5) \end{aligned}$$

$$\begin{aligned} SC^{i'} &= SC^i \cup \{(s_i, s_j, avgscore) | f_{avgscore}(s_i, s_j) \\ &\quad = (s_i, s_j, avgscore)\} \\ SC^{i'} &= \{(s_i, s_l, avgscore) | i < l < j \wedge f_{avgscore}(s_i, s_l) \\ &\quad = (s_i, s_l, avgscore) \cup \{(s_i, s_j, avgscore) | \\ &\quad sf_{avgscore}(s_i, s_j) = (s_i, s_j, avgscore)\} \\ SC^{i'} &= \{(s_i, s_l, avgscore) | i < l < j + 1 \\ &\quad \wedge f_{avgscore}((s_i, s_l)) = (s_i, s_l, avgscore) \\ SC^{i'} &= \{(s_i, s_l, avgscore) | i < l < j' \\ &\quad \wedge f_{avgscore}((s_i, s_l)) = (s_i, s_l, avgscore) \\ &\quad (proved). \quad (6) \end{aligned}$$

(Inner Loop) Termination: Because the loop is a **for** loop, it clearly terminates.

(Inner Loop) Correctness: The loop exits when $j = |S|$. The invariant shows that SC^i contains the $f_{avgscore}$ result with the inputs from s_i, s_{i+1} until $s_i, s_{|S|-1}$ when the loop terminates.

(Outer Loop) Termination: Because the loop is a **for** loop, it clearly terminates.

(Outer Loop) Correctness: The loop exits when $i = |S|$. The invariant shows that SC contains the join between the $f_{avgscore}$ result with the inputs from the pairs $(s_i, s_j) | s_i, s_j \in S \times S, i < j$ when the loop terminates.

The schema-matching process, as described before, may benefit from using one or more types of matchers to generate semantic scores, which are then aggregated as a similarity score of a single combination of corresponding attributes. Multiple matchers may be used because elements of a source schema are limited and cannot always represent real data exactly. Thus, they may provide more accurate schema integration results. Therefore, we implement two different matchers to calculate several similarity scores: a *schema-level* matcher and an *instance-level* matcher. The score aggregator module aggregates these two similarity scores as a value in the range [0,1]. The schema matcher calculates a similarity score for each combination of attributes. Therefore, it is possible to have more than one match. The match with the highest similarity score is called a semantic match. The higher the similarity score, the more likely a match is to be selected as the preferred global schema. The semantic match is used in the schema-mapping process for the mapping definition.

Example 4: Figure 8 shows an example of schema matching. First, we calculate the attribute similarities of two source schemas, `employee` and `wearSensor` from the Cassandra and MySQL data stores, respectively, using a schema-level matcher. Second, we calculate the data-instance similarity score using the instance-level matcher. Because `employee` has six attributes and `wearSensor` has two attributes, both matchers produce 12 similarity scores from the attribute combinations. Thus, we need to examine only a small subset

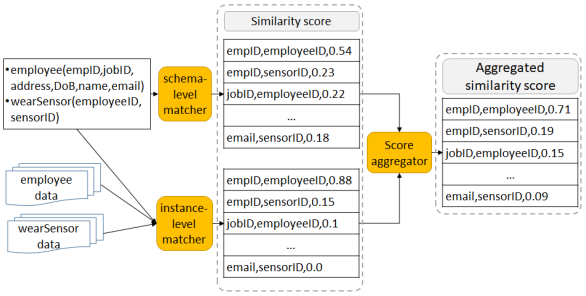


FIGURE 8. Example of schema matching.

of a set of similarity scores. Next, we aggregate and sort the similarity scores from the matchers into an aggregated similarity score.

This example demonstrates that our approach simplifies the process of schema matching from heterogeneous data stores. The similarity scores provide the likelihood of a match to be a global schema. Therefore, the user needs only to focus on the matches with a high similarity score rather than on all matches.

2) SCHEMA MAPPING

The schema-mapping process produces a mapping definition or global schema candidates. The schema mapper transforms the *semantic matches* from the schema matching and inputs a pair of local schemas into the *mapping definition*. By definition, the *semantic match* is very likely to be related, meaning that we can perform a join operation on the two schemas. The join concept is similar to that of a relational database, where we can join two tables using a join condition, such as a foreign key.

Definition 4 (Mapping Definition): A mapping definition is schema mapping expressed using a Datalog-like language that shows the relation between S_i and S_j by the semantic match $S_i.a_x \approx S_j.b_y$.

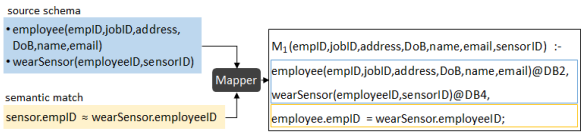


FIGURE 9. Mapping definition.

We can define a mapping definition from a pair of source schemas and a semantic match, as shown in Figure 9. The mapping definition consists of three sections: the head section is the schema definition for the global schema, and the body section describes the source schemas used to construct the global schema and the semantic matches between the two schemas. The global schema constructor module generates a default name for the global schema, such as `M1`, `M2`, and `M3`. We can edit the name of the global schema as part of the user validation step, (e.g., rename `M1` to `empSensor`).

The attributes of the global schema are constructed from the attributes in the source schemas. In the `employee` schema, the attributes consist of `empID`, `jobID`, `address`, `DoB`, `name`, and `email`, and the `wearSensor` schema has attributes `employeeID` and `sensorID`. We have the semantic match *employee.empID* $\approx$ *wearSensor.employeeID*, which describes how `empID` in the `employee` schema is similar to `employeeID` in the `wearSensor` schema. Thus, we can choose either `empID` or `employeeID` as an attribute in the global schema. We follow a rule that chooses the first attribute from the first schema as the attribute for the global schema, which is `empID` from the `employee` schema. Thus, we have the following global schema: `empSensor(empID,jobID,address,DoB, name, email,sensorID)`.

We can define several mapping definitions with the semantic matches shown in Figure 7 using the same mapping process. We use these mapping definitions to define our global schema. Figure 10 shows the global schema definition, which is constructed from several mapping definitions. The final global schema consists of four schemas: `empSensor`, `wearable`, `batch`, and `sensorWear`. The global schema also stores information on which data sources are used and the relations between them. Therefore, if a query is posed to MusQ, it is performed using the global schema, and the query processor module splits the query based on the data source definition in the global schema.

```
empSensor(empId,jobID,address,DOB,name,email,sensorID):-
   employee(empID,jobID,address,DoB,name,email)@DB2,
   wearSensor(employeeID,sensorID)@DB4,
   employee.empID=wearSensor.employeeID;
```

```
wearable(SID,serialNumber,model,purchaseDate,type,dt,pos,
heartrate,tm) :-
   sensor(SID,serialNumber,model,purchaseDate,type)@DB3,
   readings(sid,dt,pos,heartrate,tm)@DB1,
   sensor.SID = readings.sid;
```

```
batch(SID,date,avgRate,minRate,maxRate,purchaseDate,
serialNumber,model,type) :-
   sensor(purchaseDate,serialNumber,model,type,SID)@DB3,
   batchwearable(SID,date,avgRate,minRate,maxRate)@DB5,
   batchreading.SID=sensor.SID;
```

```
sensorWear(SID,serialNumber,model,purchaseDate,type,
employeeID) :-
   sensor(SID,serialNumber,model,purchaseDate,type)@DB3,
   wearSensor(employeID,sensorID)@DB4,
   sensor.SID = wearSensor.sensorID;
```

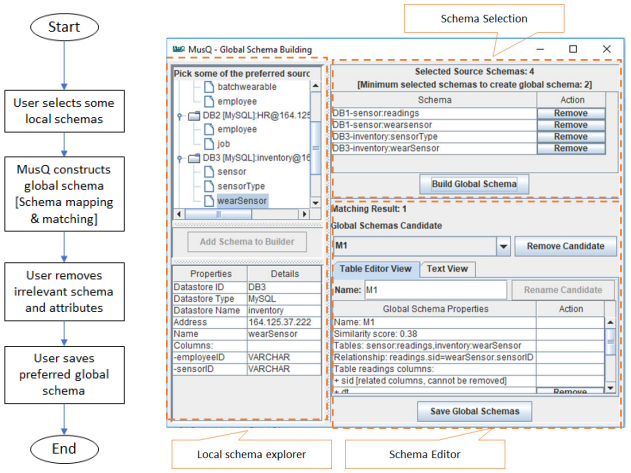
FIGURE 10. Global schema using several mapping definitions.

Because the global schema is built for data integration, it does not necessarily have to contain all the information from the local source schemas. We can omit several attributes that are not relevant to our system. In the above example, `email`, `address`, and `DoB` might be sensitive information from HR and considered irrelevant for our IoT application. Here, the user can omit these attributes from the `empSensor` schema. This attribute removal process is also considered a user validation step. Thus, the `empSensor` schema is given as follows:

```
empSensor(empID,name,jobID,sensorID).
```

The `email`, `address`, and `DoB` attributes have been removed from the `empSensor` schema, so that only attributes that are relevant to the IoT application, such as *empID, name, jobID*, and `sensorID` are used in the global schema.

Example 5: We demonstrate the construction of the global schema from local schemas and the sample data of multiple data stores using a GUI, as depicted in Figure 11.



(a) Flowchart (b) User interface

FIGURE 11. Global schema construction process using MusQ.

1) In the Local Schema Explorer area, we choose several local schemas (a minimum of two) that exist in one or more sources.
2) We execute global schema construction in the Schema Selection area, without manually choosing attributes to match. MusQ performs the schema matching and mapping at this step.
3) In the Schema Editor area, the similarity score of the constructed global schema is visible in Table Editor View and the raw constructed schema is visible in Text View (the global schema properties are similar to those in Figure 10). Thus, we can remove irrelevant global schemas and attributes.
4) Finally, we save the global schema in MQL for later usage.

From 2) and 3) above, the construction is clearly semi-automatic. MusQ shows all constructed schemas. However, the users can remove the irrelevant ones, including the attribute.

V. QUERY PROCESSING

In this section, we present our multi-store query processing system using the mediator–wrapper approach.

Figure 12 illustrates the query processing flow of MusQ, which consists of query decomposition, subquery processing, and the result finalizer. The mediator is responsible for the query decomposition and the result finalizer, whereas the wrapper performs the subquery processing.

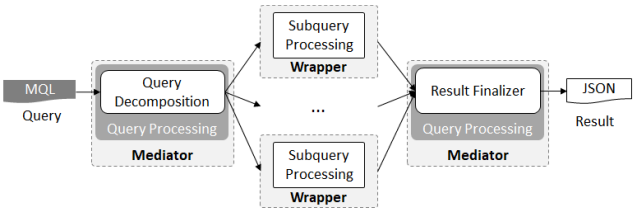


FIGURE 12. The flow of query processing in MusQ.

A. QUERY DECOMPOSITION IN A MEDIATOR

In the mediator, MusQ decomposes an input query into subqueries using the help of the global schema. To perform the decomposition, the mediator uses two modules: the *Query Parser* and the *Query Plan Generator*, as shown in Figure 13. The *Query Parser* is responsible for parsing the input MQL query into a grammar tree and performing schema validation using the global schema definition.

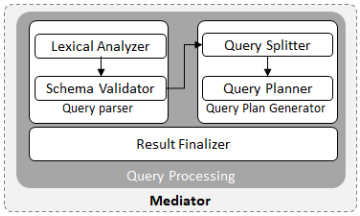


FIGURE 13. Mediator module.

Then the *Query Plan Generator* decomposes the validated input query into multiple subqueries for each local data store in the wrapper. Next, after the wrapper finishes the local query processing, the mediator receives each wrapper's intermediate results. The *result finalizer* merges all intermediate results from the wrappers into a final result.

We explain the *result finalizer* in Section V-C.

1) QUERY PARSER

A user specifies an input MQL query in the mediator and the *Lexical Analyzer* parses the input query into a *grammar tree*. We implement the *Lexical Analyzer* using ANTLR [43] as the parser generator for the MQL grammar. The *Schema Validator* checks the query using the *grammar tree* and validates the query using the global schema definition.

2) QUERY PLAN GENERATOR

After validation, the mediator sends the validated input queries to the *Query Plan Generator Module*. The *Query Splitter* decomposes the query into several subqueries based on the global schema. It looks for the schema mapping in the global schema that corresponds to the query. Then it splits the query into subqueries for each data store according to the schema mapping. The query planner also generates the query execution order for each subquery. The query execution order is important to optimize the merging of the intermediate result in the mediator.

```
SQ1(SID,serialNumber) :-
sensor(SID,serialNumber,model,purchaseDate,type)
AND type = 'wearable'
```

```
SQ2(sid,heartrate) :-
readings(sid,dt,pos,heartrate,tm)
AND heartrate > 0
```

```
SQ3(empID,name) :-
employee (empID,jobID,address,DoB,name,email)
```

```
SQ4(employeeID,sensorID) :-
wearSensor(employeeID,sensorID)
```

FIGURE 14. Subqueries for Q1.

Example 6: Consider the example in Figure 3 again. The *Query Splitter* decomposes an input MQL query into four subqueries, as illustrated in Figure 14. Since the `wearable` schema consists of two local source schemas, such as `sensor` and `readings`, the input query is split into SQ_1 and SQ_2. In the same way, since the `empSensor` schema consists of `employee` schema and `wearSensor` schema, the second part of the query is decomposed into SQ_3 and SQ_4.

After this, the *Query Planner* generates the query plan for every subquery from the query splitter. We describe the query plan generation algorithm in Algorithm 2. First, we initialize an empty query plan set *QPs*. The query plan generator examines every source s defined in the query Q in line 2. Then we find a global schema gs that is associated with the source s. Next, we assign the parent of the query plan qp as source s (line 6). First, we assign the parent of the query plan qp as source s (line 8). The parent defines the source of the subquery in the query plan. Then we assign the subquery qs, where qs has the same source as the local schema ls (line 9). We initially set the query plan as unmerged (line 11) to indicate that it is necessary to merge the subresults from the subqueries based on the matches. The matches later determine the merging pairs of the subquery results in the result finalizer. Then we add the query plan qp to the set *QPs*. Thus, at the end of the algorithm, we acquire the query plan for all subqueries.

After the query planner generates the query plan for each subquery, the mediator of MusQ pushes each subquery to the corresponding data store's wrapper.

Example 7: Figure 15 depicts a query plan corresponding to the input query in Figure 3. Based on the local schema definition in Table 2, SQ_1 and SQ_4 are sent to the inventory data store, SQ_2 is sent to the sensor data store, and SQ_3 is sent to the HR data store. We can perform a *selection* or filter operation on `sensor` and `readings` using the `type` and `heartrate` attributes, respectively. In the subqueries SQ_3 and SQ_4, we do not perform any filter operations, but directly perform a join operation using the semantic match $employee.empID \approx wearSensor.employeeID$. The order of the join operation is based on the global schema in Figure 10. The results of subqueries SQ_3 and SQ_4 are joined first using the `empSensor` schema, then the results of subqueries of SQ_1 and SQ_2 are joined using the `wearable` schema. Next, both join results are joined together, using `sensorWear` schema, as the final result.

Algorithm 2 Query Plan Generation

Input: MQL query Q, subqueries SQ
Output: Query plan set *QPs*

```
1:  Initialize QueryPlan set QPs;
2:  foreach Source s ∈ Q do
3:      GlobalSchema gs ← NULL;
4:      Set gs as the global schema associated with s.table;
5:      Matches M ← gs.getMatches();
6:      foreach LocalSchema ls ∈ gs.getLocalSchema() do
7:          Initialize QueryPlan qp;
8:          Set qp.parent as s.table;
9:          Set qp.subquery as qs ∈ QS|qs.source = ls;
10:         Set qp.match as M;
11:         Set qp.merged as FALSE;
12:         QPs.Add(qp);
        end foreach
    end foreach
```

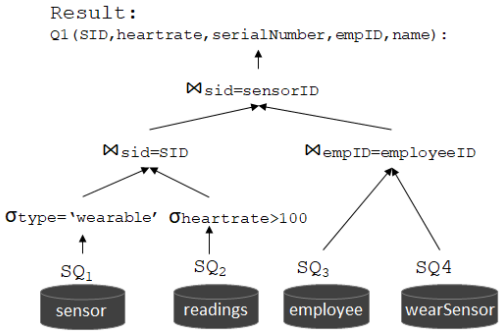


FIGURE 15. Query plan for Q1's subqueries.

B. SUBQUERY PROCESSING IN THE WRAPPER

As described in the previous section, the wrapper receives an MQL subquery from the mediator during the query processing. The role of the wrapper is to perform schema validation of the subquery over the local schema, translate it into the local data store's query language, retrieve the intermediate results by executing it, and send those results back to the mediator module. Thus, as depicted in Figure 16, the wrapper has two main components: (1) a schema driver and (2) a query driver. The schema driver extracts the schema definition from each local data store in the global schema construction process, whereas the query driver handles the subquery processing phase.

Because each data store has its own wrapper, the implementation of the *Query Converter* function is different for each data store. For example, the MySQL wrapper has an MQL to SQL *query converter* function, Cassandra has an MQL to CQL (Cassandra Query Language) *query converter* function, and MongoDB has an MQL to MongoDB *query converter* function.

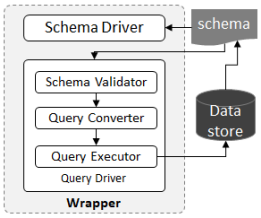


FIGURE 16. Wrapper module.

C. RESULT FINALIZER IN MEDIATOR

The result finalizer merges the intermediate results from each wrapper and stores them in a JSON object.

Algorithm 4 Merging the Subquery Result Using a Nested-Loop Join

Input: JSON subresults SR_1, JSON subresults SR_2, Matches M
Output: Merged Result R
procedure `LoopJoin`(S)
1: Initialize JSON result R;
2: **for** i *from* 0 *to* $|SR_1|$ **do**
  $sr_i = SR_1$.`Get`(i);
3: **for** j *from* 0 *to* $|SR_2|$ **do**
    $sr_j = SR_2$.`Get`(j);
4: **if** sr_i.`Get`($M.col_1$).`Equals`(sr_j.`Get`($M.col_2$)) **then**
      Initialize JSONObject o;
5: o.`AddAll`(sr_i);
6: o.`AddAll`(sr_j);
7: R.`Add`(o);
    end if
  end for
end for

As a merging option, we implemented three join algorithm approaches: nested-loop join, hash join, and sort-merge join. The detailed mechanisms of these techniques are described in Algorithm 4, Algorithm 5, and Algorithm 6, respectively. Due to the matches provided by the global schema, these join algorithms are able to merge the subresults from heterogeneous data stores.

The three algorithms use the same join approach by iterating the result set of two corresponding data stores SR_1 and SR_2 using the semantic match M defined in the global schema, which are used as input. According to semantic match M, if there is a record from the first data store subresult SR_1 that has a matching attribute to a record from the second data store subresult SR_2, both records are joined as a single record in JSON format. Thus, after all records that have the same matching attribute values are joined, each algorithm will return the merged result R as a list of JSON objects. In the case of the aggregation function, we can consider the merge results as a single table after executing partial queries, as mentioned in Section III.

Algorithm 5 Hash Join on Subquery Results

Input: JSON subresults SR_1, JSON subresults SR_2, Matches M
Output: Merged Result R
procedure `HashJoin`(S)
1: Initialize JSON result R;
  $H \leftarrow$ new Hash(key:string, value:< **JSONObject** *List* >);
2: **for** i *from* 0 *to* $|SR_1|$ **do**
  $sr_i = SR_1$.`Get`(i);
  H.`Add`(sr_i.`Get`($M.col_1$),sr_i);
  end for
3: **for** j *from* 0 *to* $|SR_2|$ **do**
  $sr_j = SR_2$.`Get`(j);
  HashList $\leftarrow H$.`Get`(sr_j.`Get`($M.col_2$));
  foreach h *in* **HashList** **do**
4: **if** h.`Get`($M.col_1$).`Equals`(sr_j.`Get`($M.col_2$)) **then**
      Initialize JSONObject o;
5: o.`AddAll`(h);
6: o.`AddAll`(sr_j);
7: R.`Add`(o);
    end if
  end foreach
end for

The nested-loop join approach in Algorithm 4 merges the subresult sets SR_1 and SR_2 according to the schema matches M in a naive way. It iterates through all the records in SR_1 (line 2), checking whether each match satisfies $sr_i \in SR_1$ and $sr_j \in SR_2$ (line 3). A match occurs when the attribute value of $M.col_1$ of sr_i equals the attribute value of $M.col_2$ of sr_j (line 4). If so, it stores the matching record of SR_1 and SR_2 as a JSON object in the result set R (lines 5-7). Thus, R contains the joined records of SR_1 and SR_2.

The hash join approach in Algorithm 5 merges the subresult sets SR_1 and SR_2 according to the schema matches M with the help of the hash table H. First, the algorithm creates a hash table H that maps all the records of SR_1 as values, where each value is paired with a key (line 2). The key for the hash table comes from the hash function result $h(x_1)$, where x_1 is the value of the $M.col_1$ attribute of a record in SR_1. After the hash table H creation finishes, the hash join attempts to find the records of SR_2 that match the records of SR_1 in H (line 3). The hash join looks for the matching record of a single record $sr_j \in SR_2$ with the subset $HashList \subseteq SR_1$. The $HashList$ contains all the records of SR_1, where their keys are equal to the hash function result $h(x_2)$, where x_2 is the value of $M.col_2$ attribute of sr_j. Next, the hash join finds the matching records of sr_j with h, where $h \in HashList$ given the condition that the attribute value of $M.col_1$ of h should equal the attribute value of $M.col_2$ of sr_j (line 4). Then it stores the matching records of SR_1 and SR_2 as a list of JSON objects in the

Algorithm 6 Sort-Merge Join on Subquery Results

Input: JSON subresults SR_1, JSON subresults SR_2, Matches M
Output: Merged Result R
procedure `SortMergeJoin`(S)
1: Initialize JSON result R;
$i \leftarrow 0, j \leftarrow 0$;
while $i < |SR_1|$ *and* $j < |SR_2|$ **do**
2: **if** SR_1.Get(*i*).Get($M.col_1$) < SR_2.Get(*j*).($M.col_2$) **then**
$i \leftarrow i + 1$;
3: **else if** SR_1.Get(*i*).Get($M.col_1$) > SR_2.Get(*j*).($M.col_2$) **then**
$j \leftarrow j + 1$;
4: **else**
5: **while** $i < |SR_1|$ *AND* SR_1.Get(*i*).Get($M.col_1$).Equals(SR_2.Get(*j*).($M.col_2$)) **do**
$k \leftarrow j$;
$sr_i = SR_1$.Get(i);
$sr_k = SR_2$.Get(k);
6: **while** $k < |SR_2|$ *AND* sr_i.Get($M.col_1$).Equals(sr_k.Get($M.col_2$)) **do**
Initialize JSONObject o;
7: o.AddAll(sr_i);
8: o.AddAll(sr_k);
9: R.Add(o);
$k \leftarrow k + 1$;
$sr_k = SR_2$.Get(k);
end while
$i \leftarrow i + 1$;
end while
end if
end while

result set R (lines 5-7). Thus, R contains the joined records of SR_1 and SR_2.

The sort-merge join approach in Algorithm 6 merges the initially sorted subresult sets SR_1 and SR_2 according to the schema matches M. The subresult sets SR_1 and SR_2 have their records sorted in non-descending order with respect to the attribute values of $M.col_1$ and $M.col_2$, respectively. The sort-merge join iterates the records of SR_1 and SR_2 by comparing the matching attributes ($M.col_1$ and $M.col_2$). If the algorithm finds that a record's $sr_i \in SR_1$ has a matching attribute value that is smaller than the matching attribute of a record $sr_j \in SR_2$, the algorithm skips the records of SR_1 until sr_i has a value larger than or equal to sr_j (line 2). The mechanism also works for SR_2 (line 3). Then, if sr_i and sr_j have an equal matching attribute value (4), this means the sort-merge join has found the start of the matching records of SR_1 and SR_2. Because the subresults are sorted, this guarantees that the i-th through the $i+a$-th records of SR_1, where $a \geq 0$, hold the same value of attribute $M.col_1$. This guarantee also holds for SR_2. Thus, in lines 5-6, the algorithm joins the matching records while also checking for their equality and the termination of the loop. Then it stores the joined records as a list of JSON objects in the result set R (lines 7-9). Therefore, R contains the joined records of SR_1 and SR_2.

Next, we provide the proof of correctness for those algorithms.

TABLE 3. The notations and corresponding descriptions for the proof of the nested-loop join.

Notation	Description
M	the match from the global schema that contains two attributes col_1 and col_2
col_1, col_2	the matching attributes of the subresults SR_1 and SR_2, respectively
$sr_{1,a}, sr_{2,b}$	a-th and b-th record of SR_1 and SR_2, respectively
$sr_{*,a}[col]$	the value of the attribute col of $sr_{*,a}$
$SR_*[a..b]$	the subset of SR_* from a-th to b-th record (inclusive).

1) CORRECTNESS OF THE NESTED-LOOP JOIN APPROACH

In this subsection, we prove the correctness of the proposed merging algorithms of the nested-loop join by applying the loop invariant technique [42] again. The loop invariant technique consists of three parts: (1) initialization, (2) maintenance, and (3) termination. Table 3 displays the notations used in this proof. The proofs for the hash join and sort-merge join algorithms are provided in the Appendix.

Theorem 2: With the two subresults SR_1 and SR_2 from the subqueries and the match M from the matching schema as input, the algorithm for the nested-loop join is correct with these loop invariants: for any step in the outer loop, this step is applied: $0 \leq i < |SR_1|$, otherwise $i = |SR_1|$; for any step in the inner loop, this step is applied: $0 \leq j < |SR_2|$, otherwise $j = |SR_2|$.

Proof 2 (Proof. Nested-Loop Join) The algorithm finds the result set R of the join between SR_1 and SR_2 with the matching attributes provided by the match M, denoted as $SR_1 \bowtie_{M.col1=M.col2} SR_2$.

(Outer Loop) Initialization: The join result set is initialized as $R = \varnothing$ and $i = 0$. No value for j is declared yet. Thus, the result set should be empty.

(Outer Loop) Maintenance: Suppose at the beginning of the iteration, the invariant holds at a particular value $i < |SR_1|$ and R contains the join result set from the beginning until $i-1$, as described in Equation 7. Let R' and i' denote the content of R and i at the end of the iteration, as described in Equation 8:

$$R = \{sr_{1,k} \cup sr_{2,j} | 0 \leq k < i \wedge 0 \leq j < |SR_2| \wedge sr_{1,k}[M.col_1] = sr_{2,j}[M.col_2]\} \quad (7)$$

$$R' = R \cup \{sr_{1,i} \cup sr_{2,j} | 0 \leq j < |SR_2| \wedge sr_{1,i}[M.col_1] = sr_{2,j}[M.col_2]\}$$

$$i' = i + 1. \quad (8)$$

At the end of the iteration, R' must contain the set of join results of the records of SR_1 from the beginning until i and

the entire SR_2, and must hold the invariant. We prove this invariant in Equation 9:

$$\begin{aligned}
R' &= R \cup \{sr_{1,i} \cup sr_{2,j} | 0 \leq j < |SR_2| \wedge sr_{1,i}[M.col_1] \\
&\quad = sr_{2,j}[M.col_2]\} \\
R' &= \{sr_{1,k} \cup sr_{2,j} | 0 \leq k < i \wedge 0 \leq j < |SR_2| \\
&\quad \wedge sr_{1,k}[M.col_1] = sr_{2,j}[M.col_2]\} \cup \\
&\quad \{sr_{1,i} \cup sr_{2,j} | 0 \leq j < |SR_2| \wedge sr_{1,i}[M.col_1] \\
&\quad = sr_{2,j}[M.col_2]\} \\
R' &= \{sr_{1,k} \cup sr_{2,j} | 0 \leq k \leq i \wedge 0 \leq j < |SR_2| \\
&\quad \wedge sr_{1,k}[M.col_1] = sr_{2,j}[M.col_2]\} \\
R' &= \{sr_{1,k} \cup sr_{2,j} | 0 \leq k < i' \wedge 0 \leq j < |SR_2| \\
&\quad \wedge sr_{1,k}[M.col_1] = sr_{2,j}[M.col_2]\} \\
&\quad (proved). \qquad (9)
\end{aligned}$$

Next, we prove the invariant in the inner loop of Algorithm 4 to find the match between $sr_{1,i}$ with the whole SR_2.

(Inner Loop) Initialization: At the beginning of the inner loop iteration, $sr_{1,i}$ and R from the outer loop are given. The result set of the loop is denoted as R^i. Then the value of j is initialized as $j = 0$. This invariant still holds because $sr_{1,i}$ is not joined with anything from SR_2 yet.

(Inner Loop) Maintenance: Suppose at the beginning of the iteration, the invariant holds a particular value of $j < |SR_2|$, and R^i contains the union of R and the joint records of $sr_{1,i}$ with the records of $SR_2[0 \ldots j-1]$, as defined in Equation 10. Let $R^{i'}$ and j' denote the content of R^i and j at the end of the iteration, as described in Equation 11. We prove this invariant in Equation 12:

$$\begin{aligned}
R^i &= R \cup \{sr_{1,i} \cup sr_{2,l} | 0 \leq l < j \wedge sr_{1,k}[M.col_1] \\
&\quad = sr_{2,l}[M.col_2]\} \qquad (10) \\
R^{i'} &= R^i \cup \{sr_{1,i} \cup sr_{2,j} | sr_{1,i}[M.col_1] = sr_{2,j}[M.col_2]\}; \\
j' &= j + 1 \qquad (11) \\
R^{i'} &= R^i \cup \{sr_{1,i} \cup sr_{2,j} | sr_{1,i}[M.col_1] = sr_{2,j}[M.col_2]\} \\
R^{i'} &= R \cup \{sr_{1,i} \cup sr_{2,l} | 0 \leq l < j \wedge sr_{1,i}[M.col_1] \\
&\quad sr_{2,l}[M.col_2]\} \cup \{sr_{1,i} \cup sr_{2,j} | sr_{1,i}[M.col_1] \\
&\quad = sr_{2,j}[M.col_2]\} \\
R^{i'} &= R \cup \{sr_{1,i} \cup sr_{2,l} | 0 \leq l \leq j \wedge sr_{1,i}[M.col_1] \\
&\quad = sr_{2,l}[M.col_2]\} \\
R^{i'} &= R \cup \{sr_{1,i} \cup sr_{2,l} | 0 \leq l < j' \wedge sr_{1,i}[M.col_1] \\
&\quad = sr_{2,l}[M.col_2]\} \\
&\quad (proved). \qquad (12)
\end{aligned}$$

(Inner Loop) Termination: As the loop is a **for** loop, it clearly terminates.

(Inner Loop) Correctness: The loop exits when $j = |SR_2|$. The invariant shows that R^i contains the join between $sr_{1,i}$ and $SR_2[0 \ldots |SR_2| - 1]$ with the match M when the loop terminates.

(Outer Loop) Termination: As the loop is a **for** loop, it clearly terminates.

(Outer Loop) Correctness: The loop exits when $i = |SR_1|$. The invariant shows that R contains the join between $SR_1[0 \ldots |SR_1| - 1]$ and SR_2 with the match M when the loop terminates.

Thus, we can perform the aggregation after the join operation finishes according to the previously produced query plan.

D. COMPLEXITY ANALYSIS OF QUERY PROCESSING

In this subsection, we discuss the cost model of MusQ's query processing to show the complexity analysis of our approach. We modified the cost model in [44] to suit our mediator–wrapper-based heterogeneous multi-store system. Our cost model C is defined as the sum of the query processing costs at the mediator $C_{mediator}$ and at the wrapper $C_{wrapper}$ and the network transfer costs $C_{transfer}$, as described in the equation below:

$$C = C_{mediator} + C_{wrapper} + C_{transfer}. \qquad (13)$$

1) MEDIATOR COST

First, we describe the cost model of the mediator of a query, $C_{mediator}$, according to Sections V-A and V-C. In Section V-A, we decomposed a single MQL query into sub-queries by two processes: parsing and query plan generation. Suppose, for a single query, the time to perform the parsing and query plan generation are t_{parse} and t_{plan}, respectively.

After the wrappers finish their work, the mediator merges the intermediate results and convert them into JSON objects. MusQ performs these two steps a number of times equal to the number of intermediate result join operations as decided in the generated query plan. We define the set of those join operations as J. In the j-th join operation, suppose we have two different sets of intermediate results SR_1^j and SR_2^j. The time to merge the intermediate results depends on the sizes of the result sets ($|SR_1^j|$, $|SR_2^j|$) and the choice of merging algorithm m. Thus, the merging time of the j-th join operation is $t_{merge-m,j}(|SR_1^j|, |SR_2^j|)$, where $m \in \{nested-loop-join, hash-join, merge-join\}$. Therefore, we can define the cost of the mediator $C_{mediator}$ as

$$C_{mediator} = t_{parse} + t_{plan} + \sum_{j \in J} t_{merge-m,j}(|SR_1^j|, |SR_2^j|). \qquad (14)$$

We now describe the merging time of each algorithm. Suppose MusQ is processing the j-th join operation to merge $|SR_1^j|$ and $|SR_2^j|$. We describe the merging time of the nested-loop join $t_{merge-nested-loop-join}(|SR_1^j|, |SR_2^j|)$, hash join $t_{merge-hash-join}(|SR_1^j|, |SR_2^j|)$, and sort-merge join $t_{merge-sort-merge-join}(|SR_1^j|, |SR_2^j|)$ in Equations 15, 16, and 17, respectively. The nested-loop join (Algorithm 4) naively joins the intermediate result set SR_1^j and SR_2^j directly. We denote the time of this merging style as $t_{nested-loop-join}$. Next, the hash join (Algorithm 5) maps the SR_1^j first as H,

then joins H and SR_2^j. We denote the time of the map and the merging of hash join as $t_{hash-map}$ and $t_{hash-join}$, respectively. Finally, the sort-merge join (Algorithm 6) first sorts the intermediate result sets SR_1^j and SR_2^j as SRs_1^j and SRs_2^j, respectively, and then joins SRs_1^j and SRs_2^j. We denote the time of the sorting and the merging of sort merge join as t_{sort} and $t_{sort-merge-join}$, respectively:

$$t_{merge-nested-loop-join} = t_{nested-loop-join}(|SR_1^j|, |SR_2^j|) \quad (15)$$

$$t_{merge-hash-join} = t_{hash-map}(|SR_1^j|) + t_{hash-join}(|H|, |SR_2^j|) \quad (16)$$

$$t_{merge-sort-merge-join} = t_{sort}(|SR_1^j|) + t_{sort}(|SR_2^j|) + t_{sort-merge-join}(|SRs_1^j|, |SRs_2^j|). \quad (17)$$

Due to the merging process (t_{merge}), the mediator cost is highly dependent on the size of the intermediate results. By contrast, the other components, which are t_{parse} and t_{plan}, do not depend on the size of the intermediate results. Next, we discuss the wrapper cost.

2) WRAPPER COST

We define the cost model of the wrapper's query processing using the explanation in Section V-B. First, we suppose there are a set of wrappers W that process the subqueries from the mediator. Then, a single wrapper $w \in W$ validates a subquery input. The wrapper w translates this subquery into the wrapper's query language. We denote the required times for the validation and the translation as $t_{validate,w}$ and $t_{translate,w}$, respectively. The wrapper w then executes the subquery on a data store D. The execution time is highly dependent on the data set size $|D|$ and the wrapper's data store type. We denote the execution time as $t_{exec,w}$. Therefore, we can write the cost model of the wrapper as

$$C_{wrapper} = \sum_{w \in W} t_{validate,w} + t_{translate,w} + t_{exec,w}(|D|)). \quad (18)$$

The wrapper cost consists of three components: $t_{validate,w}$, $t_{translate,w}$, and $t_{exec,w}$. The original data size in each corresponding data store wrapper in $t_{exec,w}$ heavily affects the wrapper cost. The wrapper cost also significantly depends on the performance of the wrappers that process the subqueries W. This applies to all components of the wrapper cost ($t_{validate,w}$, $t_{translate,w}$, and $t_{exec,w}$). We now discuss the transfer cost that covers the communication-related problems.

3) TRANSFER COST

The transfer cost $C_{transfer}$ consists of the communication time required to send and receive data between the system and the user, and between the mediator and the wrapper. We define the communication time of the transfers from system to user as t_{q-s} and from user to system as t_{s-q}. Suppose the query processing involves a set of wrappers W. We define the data transfer time from mediator to wrapper as t_{med-w} and from wrapper to mediator as t_{w-med}, where $w \in W$. The transfer time from the wrapper to the mediator t_{w-med} also depends on the result set of the subqueries RS_w in each wrapper. However, the mediator only sends the query plan to the wrapper. Therefore, the transfer time from the mediator to the wrapper t_{med-w} is independent of the size of the data. Thus, we can write the transfer cost model as

$$C_{transfer} = t_{s-q} + \sum_{w \in W} t_{med-w} + t_{w-med}(|RS_w|) + t_{q-s}. \quad (19)$$

The transfer cost $C_{transfer}$ consists of four different costs, including the interaction between the system and the user (t_{s-q}, t_{s-q}) and between the mediator and the wrapper (t_{med-w}, t_{w-med}). All of these transfer costs are independent of the intermediate result size $|RS_w|$, except for the transfer from the wrapper to the mediator t_{w-med}.

VI. EXPERIMENT AND EVALUATION

In this section, we describe a comprehensive performance evaluation of MusQ's query processing using a synthetic and a real IoT data set, where each data set is more than 1 GB in size. We also provide queries that closely resemble IoT analytics queries to show that our system is suitable for IoT applications.

A. EXPERIMENT SETUP

1) HARDWARE AND ENVIRONMENT SETUP

We implemented MusQ[1] in Java using the Java Development Kit version 1.8, which sets the maximum JVM memory to 1024 MB. All experiments were conducted on commodity machines equipped with an Intel Core i3-6100 3.7 GHz CPU and 8 GB of memory on the 64-bit Ubuntu 16.04 operating system. Four data stores were used to store the data set: Cassandra version 3.10, MySQL version 5.7, MongoDB version 3.2.13, and HDFS from Hadoop 2.6.0.

2) EXPERIMENT SCENARIOS

The experiments were developed within several categories: global schema construction, pre-experiment for query processing, and query processing. We performed the global schema construction experiment to show the capability of MusQ to build a global schema from local schemas. We conducted the preliminary experiment using a small data set to provide perspective on data store combination configurations before using the large data set. Then we executed the query-processing experiment for several narrower purposes, including the effects of the global schema in the multi-store system, performance of different data store combinations, and performance of different merging algorithms. To obtain sound and reliable experimental results, we repeated every test 10 times and averaged the results over all repetitions.

We also examined the performance of Apache Drill on some of the experiments as a comparison to the performance of our system

[1] Source code available at: https://gitlab.com/dslab/MusQ.

3) DATA SET

We used two different IoT data sets: a synthetic and a real data set. The synthetic data can be classified into two categories: static data and sensor data. Static data are non-device data that do not contain streamed sensor data, such as HR data or inventory data. The sensor data are device data that contain streamed sensor data, such as heartbeat readings of the device wearer. For the real data set, we used the CityPulse road traffic data set [24], which stores data on vehicle traffic.

As mentioned before, several combinations of static and sensor data stores were required to perform the query-processing experiments. These combinations consisted of two and three data stores. The combination of two data stores is composed of static and sensor data stores, whereas the combination of three data stores is composed of either two static and one sensor data stores or one static and two sensor data stores. The static data were stored in both a relational database (MySQL) and in a non-relational database (MongoDB, Cassandra, and HDFS), whereas the sensor data were stored in a non-relational database only. This option was chosen because a relational database is not built to store massive amounts of sensor data.

B. EXPERIMENTAL RESULT ON SYNTHESIZED DATA SET

1) GLOBAL SCHEMA CONSTRUCTION

We performed a global schema construction based on two different data stores, DB1 (unstructured) and DB3 (structured). We redisplay the data store information in Table 4.

TABLE 4. Local schemas used in global schema construction demonstration.

ID	Data Store	Database Name	Schema Definition
DB1	Cassandra	Sensor	readings(sid, dt, tm, heartrate, pos)
DB3	MySQL	Inventory	sensor(SID, serialNumber, model, purchaseDate, type)

```
wearable(SID,serialNumber,model,purchaseDate,dt,pos,
heartrate,tm):-
    sensor(SID,serialNumber,model,purchaseDate,type)@DB3,
    readings(sid,dt,pos,heartrate,tm)@DB1,
    sensor.SID=readings.sid;
```

FIGURE 17. Result of semi-automatic global schema construction experiment.

The global schema in Figure 17 incorporates the properties of both local schemas. The structure of the global schema is defined as follows. The first term (‘‘wearable’’) describes the name of the global schema and the remainder of the term contains the extracted and user-selected attributes from the local schemas. After the colon and hyphen (‘‘:-’’), the definitions of the local schemas are similar to the < name:(set of attributes)> style, but followed by ‘‘@< data store ID>.’’ The global schema process concluded when we obtained the join conditions of both local schemas.

2) EXPERIMENT USING A SMALL DATA SET

We conducted this experiment as a preliminary to identify data store technical problems that may be related to the size of the data set when performing a query, such as a memory limit or a long execution time. We addressed these problems from the query processing experiment, as they may hinder the execution of the experiment. In addition to identifying the limitations specific to the data stores, we also examined the performance of the mediator–wrapper query processing approach on a small data set.

```
Q-EX(sid,dt,tm,heartrate,model,serialnumber,empid,name):-
  wearable(sid,serialnumber,model,purchasedate, type,
  dt,tm,heartrate,pos) AND
  empSensor(empid,jobid,address,dob,name,email,sensorid)
  AND sid = 'S0001' dt='2017-04-11'
  AND heartrate > 95
```

(a) Query of MusQ

```
SELECT readings.sid, readings.dt, readings.tm,
readings.heartrate, sensor.model, sensor.serialnumber,
employee.empid, employee.name FROM DB1.sensor.readings,
DB3.inventory.wearSensor, DB2.hr.employee,
DB3.inventory.sensor WHERE readings.sid = sensor.SID
AND sensor.SID = wearSensor.sensorID AND employee.empid
= wearSensor.employeeID AND readings.sid = 'S0001' AND
readings.dt='2017-04-11' AND readings.heartrate > 95
```

(b) Query of Apache Drill

FIGURE 18. Query example for the small synthetic data set.

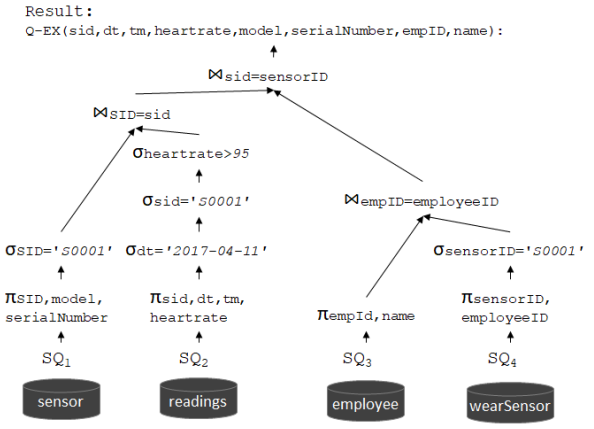


FIGURE 19. Query plan for the small synthetic data set.

We detail the small data set's query and query plan in Figures 18 and 19, respectively. The query shows filter operations and the query plan displays join operations for merging the results from each data store. Thus, this query resembles an IoT analytics application because it deals with heterogeneous data stores simultaneously. In further experiments, we used this data set and varied the size according to our needs. We controlled the size of the data set by removing or changing the selection criteria of some attributes, such as `sid`, `sensorID` `dt`, or `heartrate`.

We provide the query of MusQ and that of Apache Drill in Figure 18(a) and (b), respectively. With our global schema definition of MusQ, we only need to describe our desired query with selected columns and filters. By contrast, in the

query of Apache Drill, it is obvious that the table names are necessary in the filter and join definition. The global schema of MusQ already provides the match, thus the users do not need to rewrite them in the query.

For the small data set of 2,000 tuples, we performed the MQL queries using a combination of two data stores (static + sensor). We considered only MySQL as static data storage because it represents RDBMS, whereas the others represent non-relational databases (NoSQL and HDFS). This consideration demonstrates the ability of MusQ to work on both relational and non-relational databases when processing a single query. We used three different combinations of two data stores (static + sensor) for the experiment: MySQL+Cassandra, MySQL+MongoDB, and MySQL+HDFS.

TABLE 5. Query performance of static + sensor data stores using 2,000 tuples.

Stage	Execution time (s)		
	MySQL+Cassandra	MySQL+MongoDB	MySQL+HDFS
Mediator	**0.152**	0.161	0.235
Wrapper	0.801	**0.786**	1.258
Total	0.953	**0.947**	1.493

Table 5 shows the results of this preliminary task. The combinations of MySQL+Cassandra and MySQL+MongoDB showed similar performance, whereas the combination of MySQL+HDFS exhibited the worst performance.

This experimental result is consistent with our wrapper cost model, as described in Equation 18. The query processing time of the wrapper was highly dependent on the data store type of each involved wrapper. Thus, each data store combination had a different query processing time.

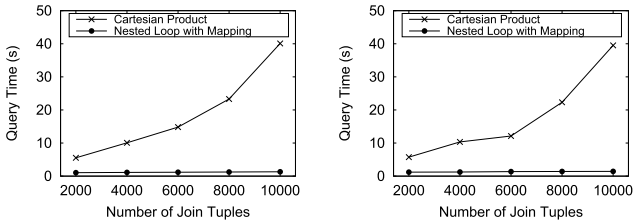
The inferior performance of MySQL+HDFS occurred because HDFS does not support indexes. The absence of indexes required MusQ to access all the files, which greatly reduced the performance of query processing. The performance of MySQL+HDFS in terms of query time was also consistent in both the mediator and the wrapper of each combination.

The execution times of the wrappers were much slower than those of the mediators. This indicates that processes in a wrapper, such as subquery translation and local query processing in each data store, are more exhaustive than the processes in the mediator (e.g., query parsing and the merging of intermediate results in a small data set).

Based on these results, we do not further report any combination of data stores using HDFS because, even for the small data set, it took up to 1.5 times longer to process a query than the other combinations did.

3) QUERY-PROCESSING EXPERIMENT

In this section, we show the performance of MusQ when processing a larger synthesized data set. We conducted this experiment to investigate the following questions: (1) *Does a global schema affect the performance of a query in a multi-store system?* (2) *How well does the mediator–wrapper approach of MusQ perform?* (3) *How do different merging algorithms influence the query processing for different data store combinations?* (4) *Is MusQ robust enough to process a large data set?*



(a) MySQL + MongoDB (b) MongoDB + Cassandra

FIGURE 20. Comparison between the Cartesian product and nested-loop (NL) joins with global schema mapping when processing a small number of join tuples.

Thus, the experiments were performed on a large data set using approaches that provide a global schema, different data store configurations, and different intermediate result merging algorithms. In all the experiments, we used the execution times to compare performances.

In addition, we examined the performance of Apache Drill starting with the large data set query as a comparison to the performance of MusQ.

Effects of a Global Schema:

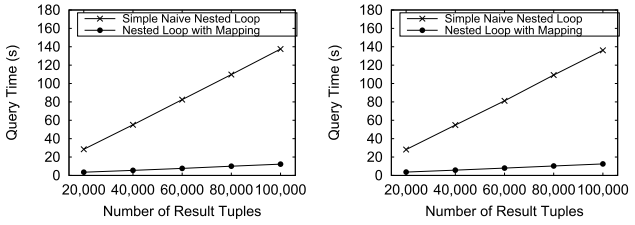
To see the effects of a global schema, we compare the query execution times with and without a global schema using a small data set. The execution without a global schema uses two distinct approaches: the Cartesian product and the naive simple nested-loop join. This experiment used four static + sensor data store combinations: MySQL+Cassandra, MySQL+MongoDB, MongoDB+Cassandra, and Cassandra+MongoDB.

As explained in Section V, MusQ utilizes the global schema to decompose an MQL query into subqueries, and performs join-style algorithms to merge all intermediate results from the subqueries. Even though the global schema does not exist, a query in a multi-store system can still be executed, especially the merging of the intermediate results. We can combine the intermediate results using a Cartesian product operation, which is the most naive approach (this also called the baseline approach). Similarly to the Cartesian product, the naive simple nested loop (simple NL) can be used to merge these results without mapping. Both approaches can be explained more clearly when comparing their performance to that of the MusQ system. We select one merging algorithm as MusQ's merging algorithm (nested loop or NL) to make the comparison easier.

Figure 20 shows the results from two combinations of data stores with a small set of join tuples between two different joins. We omit the results for the MySQL+Cassandra and Cassandra+MongoDB combinations because they show the same trend as the combinations of MySQL+MongoDB and of MongoDB+Cassandra, respectively.

The execution time of the Cartesian product approach increased significantly as the number of tuples increased, whereas the execution time for our query processing system (NL with mapping) increased only slightly. The Cartesian product approach's worst performance occurred for MySQL+MongoDB when processing 10,000 tuples. It needed approximately 40 s, which is 30 times longer than our nested-loop approach using the global schema. This is problematic because 10,000 is a relatively small number of tuples.

The Cartesian product approach is inefficient and wastes memory because of its naive approach. For example, a sensor relation with 2,000 tuples and a static relation with 1,000 tuples create 2 million tuples as a result of the Cartesian product. Therefore, due to the main memory limitation of the Cartesian product, we varied the number of tuples of the sensor relation from 2,000 to 10,000 in steps of 2,000 tuples and fixed the number of tuples in the static relation to 1,000 in this experiment.



(a) MySQL + Cassandra (b) MongoDB + Cassandra

FIGURE 21. Comparison between the NL join without and with the global schema.

We show the difference between the join performance of merging the intermediate subquery results with and without global schema in Figure 21. For this experiment, we varied the number of tuples in a sensor relation from 20,000 to 100,000 in steps of 20,000 and joined them with the same number of tuples in a static relation. Again, we omit the result from the combinations of MySQL+MongoDB and Cassandra+MongoDB for the same reason as the previous result. The execution time of the simple naive NL join approach increased significantly as the number of tuples increased, whereas the execution time of the NL join with the global schema showed only a slight increase. This is mainly because the simple NL join still tries to find join conditions from two tuples by checking values of all attributes, whereas the join with the global schema exploits join conditions provided by the global schema.

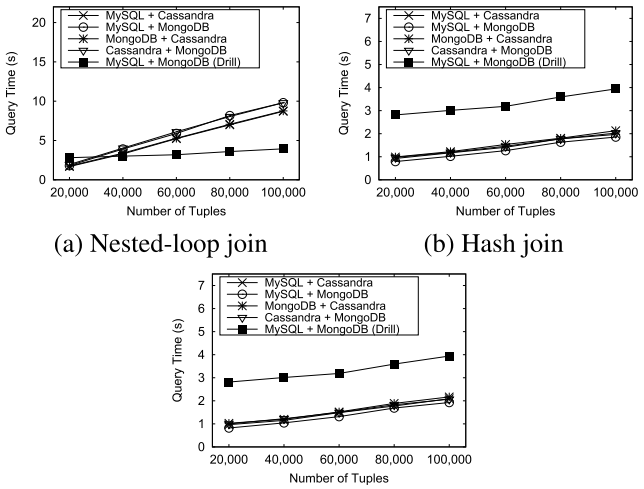
Because the performance of our system was always superior to the baseline approach, henceforth we only report the experimental performance of our MusQ system.

4) EFFECTS OF MERGE ALGORITHMS WITH A LARGE DATA SET

As explained in Section V-B, we devise three join approaches to merge intermediate results from wrappers: (1) nested-loop join, (2) sort-merge join, and (3) hash join.

In this set of experiments, we evaluated the query-processing performance of these three approaches when MusQ handled a large data set using two different data stores. We investigated how each merging algorithm affects the query processing of MusQ. Similar to the previous experiment, we used four data store combinations: MySQL+Cassandra, MySQL+MongoDB, MongoDB+Cassandra, and Cassandra+MongoDB.

We only used the MySQL+MongoDB combination for querying in Apache Drill because it does not support Cassandra by default. In addition, Drill uses hash join by default to merge intermediate results.



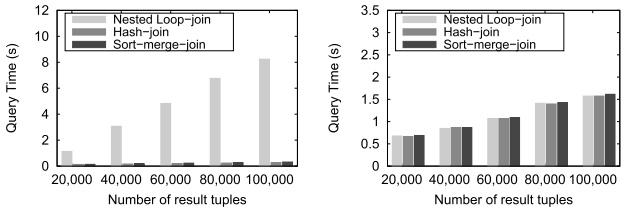
(a) Nested-loop join (b) Hash join

(c) Sort-merge join

FIGURE 22. Query processing with two data stores using different merging algorithms.

Figure 22 depicts the results of this experiment for different combinations of data stores. We placed the default performance of Drill using hash join in each result. The query execution times for all four data store combinations were similar, with only slight differences proportional to the provided number of tuples. These slight differences were due to the wrapper performance of each data store. The combination of MySQL+MongoDB showed the best performance because it had the lowest network communication time and query execution. On the other hand, the four data store combinations were similar because of the uniformity of the subquery results in the form of JSON objects from each data store wrapper. The query execution time of the NL join with the global schema was approximately 10 s when 100,000 tuples were utilized during the merge process, whereas the other two merging algorithms showed lower execution times.

Furthermore, MusQ using hash join and sort-merge join outperformed Drill with the MySQL+MongoDB data stores. The performance of Drill was approximately linear with the number of tuples, but it exhibited an additional 2 s compared to the hash join and sort-merge join of MusQ for each number of tuples.



(a) Mediator (b) Wrapper

FIGURE 23. Performance of MusQ query processing in the MySQL+MongoDB combination, separated into mediator and wrapper processes.

Figure 23 shows the same experimental result, but divided into the mediator and the wrapper to better grasp the comparison using three different join algorithms in MusQ's mediator–wrapper architecture. The combination of MySQL and MongoDB as static and sensor data stores were used because they performed best in this experiment (see Figure 22). The wrapper performance for all results was similar for the same number of tuples and any data store combination. However, significant differences in mediator performance are visible in the join algorithms.

The mediator in the nested-loop join performed poorly compared with the hash join and sort-merge join approaches. This finding is consistent with our mediator cost model in Equation 14, as the choice of the merging algorithm affects the mediator cost. The wrapper cost model in Equation 18 also holds true because the processing time of a wrapper depends on the size of the data set. Thus, in further direction, it is necessary to determine the join style to merge the intermediate results based on the cost models estimation. A precise estimation of the cost model, especially the mediator cost model, provides an efficient join strategy for MusQ.

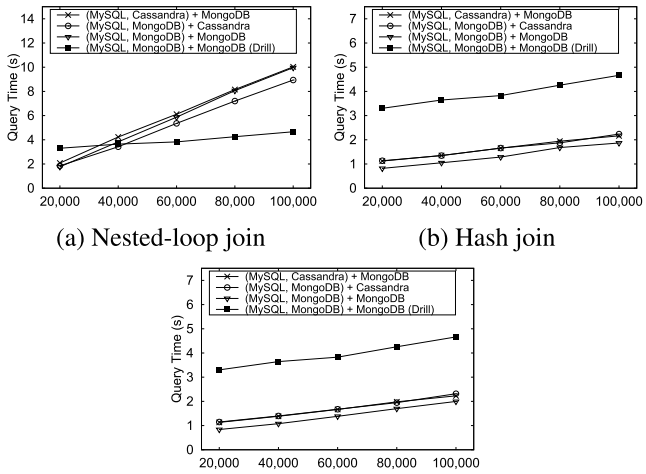
Similar to the previous result, the nested-loop join took the longest among the other joins. When performing a join operation on 100,000 tuples, the nested-loop join method required more than 8 s, whereas the hash join and sort-merge join approaches took less than 1 s. Thus, the choice of join algorithm has a significant effect on the mediator performance. The nested-loop join looks for all possible matches from a single record by iterating over all records in a pairing relation. Thus, the mediator takes significantly longer than the hash join and the sort-merge join, which minimize the matching process using a hashing function and sorting, respectively.

Moreover, the wrapper execution time for each join algorithm increased with the number of result tuples. In contrast to mediator, the choice of algorithm does not affect the performance of the wrapper. The wrapper performed similarly for each number of tuples, with a difference of less than 0.2 s.

5) COMBINATIONS OF THREE DATA STORES

In this subsection, we evaluate the query processing performance using combinations of three data stores. In this case, we use MySQL plus an additional NoSQL data store type as static data storage, and the NoSQL data store for the sensor data store. Thus, we used three data store combinations (which we denote as (static, static) + sensor) for this experiment: (1) (MySQL, Cassandra) + MongoDB, (2) (MySQL, MongoDB) + Cassandra, and (3) (MySQL, MongoDB) + MongoDB. The third combination serves as a comparison to Apache Drill, which does not support Cassandra by default.

Similarly to the previous experiment, we varied the number of sensor data from 20,000 to 100,000 in steps of 20,000 tuples and used all three merging algorithms to combine the intermediate results from the wrappers. We compare the results of the experiment with the performance of Apache Drill using hash join as its default merging algorithm.



(a) Nested-loop join (b) Hash join

(c) Sort-merge join

FIGURE 24. Query processing of three data store combinations using different merging algorithms.

Figure 24 depicts the results of query processing in three data stores. Similar to the experiment that used two data store combinations, the execution time was proportional to the number of result tuples. However, we also found that (MySQL, MongoDB) + MongoDB, as a combination of two static data stores and one sensor data store, performed best. Similar to the previous experiment, the performance of Apache Drill using the (MySQL, MongoDB) + MongoDB data store combination also was outperformed by the hash join and sort-merge join of MusQ. This result is consistent with the MySQL+MongoDB combination as a static and sensor data store in the previous experiment, which also performed slightly better than the other combinations. Both results are caused by less network communication than the other combinations.

Table 6 summarizes the same experiment on 100,000 tuples of intermediate results, divided into mediator and wrapper execution times for each algorithm and data store combination. Table 6 also includes the breakdown of the Apache Drill performance. The performance of Drill consists of two stages: planning and execution. These two terms have different meanings compared to MusQ's mediator and wrapper. The execution time of Drill includes the merging process of

TABLE 6. Query performance of three data stores combination using 100,000 tuples.

Data store	Stage	Merging Algorithm		
		Nested-loop	Hash	Sort-merge
(MySQL,MongoDB)+Cassandra	Mediator	7.11	0.27	0.28
	Wrapper	1.82	1.95	2.02
(MySQL,Cassandra)+MongoDB	Mediator	8.16	0.30	0.34
	Wrapper	1.86	1.85	1.88
(MySQL,MongoDB)+MongoDB	Mediator	8.38	0.29	0.32
	Wrapper	1.57	1.60	1.66
(MySQL,MongoDB)+MongoDB (Drill)	Planning	-	2.16	-
	Execution	-	2.50	-

the mediator of MusQ and the execution time of the wrapper, whereas the planning time only focuses on the query planning stage.

Consistent with the previous experiment using two data store combinations, the nested loop performed the worst in the mediator stage, whereas in the wrapper stage, all three algorithms worked similarly. We also found that the hash join worked better than the sort-merge join in the mediator stage, because the sort-merge join sorts all tuple values first rather than matching records first. On average, the (MySQL, MongoDB) + Cassandra data store combination outperformed the (MySQL, Cassandra) + MongoDB combination. This is supported by the lower communication cost of MySQL as a static data store and MongoDB as a sensor data store. It is clear that, compared to the hash join and sort-merge join of MusQ, Drill is not as efficient as MusQ in performing the queries. The execution time of Drill is longer than the sum of the mediator time, which includes the merging time, and wrapper time of MusQ.

C. EXPERIMENTAL RESULT USING A REAL IoT DATA SET

In this experiment, we evaluated the query processing of MusQ using the CityPulse data set [24], which stores static and sensor data. The static data consists of traffic report metadata, whereas the sensor data consists of vehicle road traffic and pollution reports. The total size of the CityPulse data set is 1.3 GB, which is spread into heterogeneous data stores. We compared the performance results of MusQ to Apache Drill using MySQL+MongoDB on two data store combinations and MySQL + (MongoDB, MongoDB) on three data store combinations. We also performed a query that represents an IoT analytics query in a smart city application.

We used a slightly different data store combination setup from the previous experiment and only used the hash join merging algorithm, as it is the best of the three merging algorithms. We tested various use cases, such as monitoring the frequency of passing vehicles in slightly polluted areas and checking the speed of passing vehicles in several polluted streets. The example query in Figure 25 reports carbon monoxide rates, the average speed of passing vehicles, and the street IDs. We only include reports where the average speed of the passing vehicles is higher than 70 km/h and report IDs in a certain interval (between 158,324 and 158,600). We also use our construction of global schema from

```
Q1(REPORT_ID,TIMESTAMP,timestamp,avgSpeed,point_1_street,
point_2_street,carbon_monoxide) :-
 traffics_cm(REPORT_ID,TIMESTAMP,avgSpeed,vehicleCount,
 DISTANCE_IN_METERS,ROAD_TYPE,point_1_street,
 point_2_street)
 AND pollutions_cm(REPORT_ID,timestamp,ozone,
 particullate_matter,carbon_monoxide,sulfure_dioxide,
 nitrogen_dioxide,point_1_street,point_2_street)
 AND REPORT_ID >= 158324 AND REPORT_ID < 158600
 AND avgSpeed >= 70
```

FIGURE 25. Query for CityPulse data set experiment with 100,000 tuples.

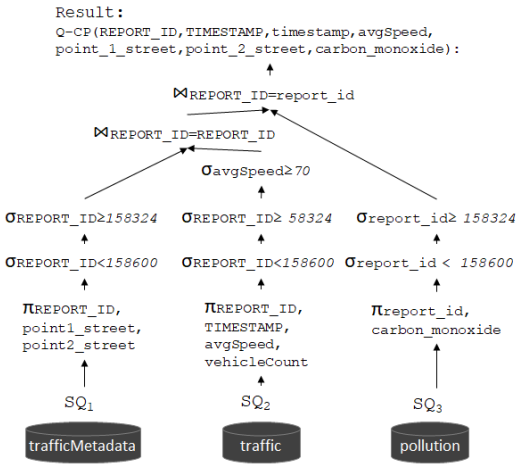


FIGURE 26. The query plan for CityPulse dataset experiment.

the data stores. Similar to the synthetic data set experiment, we can change the result size of the CityPulse data set by altering the selection criteria for some attributes, such as `REPORT_ID` or `avgSpeed`. We performed the filter and join operation to represent the IoT analytics query. A query plan example for the CityPulse data set is shown in Figure 26. The result shows which data store combination is suitable for storing the CityPulse data set.

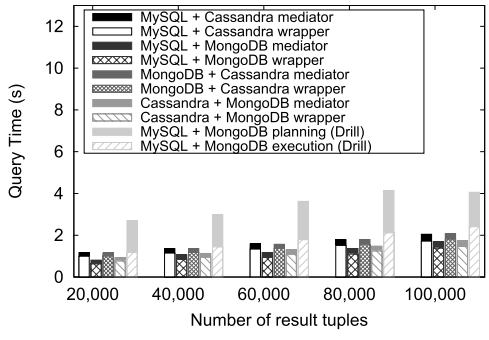


FIGURE 27. Two data stores performance of the CityPulse dataset.

1) TWO DATA STORE COMBINATION

Figure 27 shows that the performance trend of each data store combination can be identified more clearly, especially when choosing MongoDB or Cassandra as the sensor data store. The wrapper times of both MySQL+MongoDB

and Cassandra+MongoDB, which are similar in terms of query time, are significantly different to those of MySQL+Cassandra and MongoDB+Cassandra in the wrapper. Thus, we can conclude that MongoDB, as a sensor data store, plays a significant role in the MusQ query performance due to its minimum communication cost. The performance of Apache Drill is consistent with the previous experiments. The query time of Apache Drill is approximately 2 s longer than that of MusQ. In turn, the query time of MusQ is slightly longer than the synthetic data store, especially in the wrapper. This significant change occurred because the CityPulse data set has more rows and attributes than the synthetic data set.

2) THREE DATA STORE COMBINATION

The three data store combination experiment using the CityPulse data set was slightly different to the synthetic data set experiment. For this experiment, we denote the data set combination as static + [sensor, sensor]. Figure 28 shows that the MySQL + [MongoDB, MongoDB] combination using MusQ had the best performance among all the combinations. This finding is supported by the fact that in the two data store experiment with the CityPulse data set, the usage of MongoDB as the sensor data set gave the best performance. Consistent with the previous experiment, the execution times of Apache Drill were worse than those of MusQ for the same data store combination.

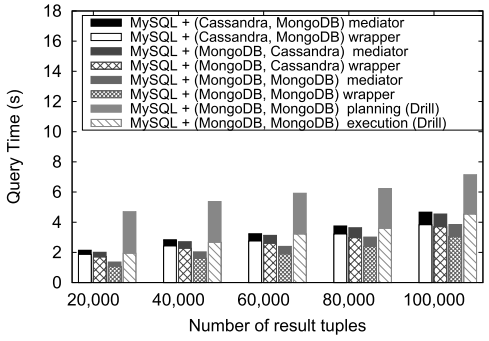


FIGURE 28. Three data stores performance of the CityPulse data set.

The superior performance of MusQ compared to Apache Drill is the result of several factors. Despite the provision of relevant matches to Drill's queries, it still requires a long time to plan the queries. Drill optimizes for columnar storage as well as columnar execution through an in-memory hierarchical columnar data model. Therefore, this style of execution appears to be insufficient for general queries.

VII. CONCLUSION

In this paper, we have presented the design of MusQ for building a multi-store query processing system. MusQ semi-automatically constructs the global schema from local source schemas using schema-matching and schema-mapping steps to provide integrated access to multiple data sources.

The users of MusQ utilize a Datalog-like language for retrieving IoT data. The semi-automatic schema construction process reduces the burden on users to manually define the global schema, while the easily translatable MQL query removes the necessity of remembering the details of various query languages. Three join operations-focused query processing methods based on a mediator–wrapper approach enable efficient integrated access to multiple heterogeneous data stores. We conducted extensive experiments with a synthetic IoT data set and a real IoT data set. The performance evaluations indicate that MusQ provides scalable and efficient query processing across multiple combinations of MySQL, Cassandra, MongoDB, and HDFS. This performance is comparable to a state-of-the-art multi-store system. Our experimental results confirm the suitability of MusQ for the IoT environment. Thus, based on these results, MusQ offers a more efficient data integration in several areas that involve complex IoT systems, such as smart cities, healthcare, and energy management.

In future work, we will extend the capability of MusQ to account for broader support for other data stores and IoT security problems. The current MusQ data store support is limited to MySQL, Cassandra, MongoDB, and HDFS. On the next development, MusQ will cover more data store types, e.g. Hive, HBase. In addition, the cost model estimations of MusQ mediators will be provided to improve the efficiency of the query processing. On the other hand, accessing IoT data store requires a secure protocol because an IoT data store may contain private and sensitive data. As an IoT multi-store query system, MusQ should protect users' private IoT data. As an example, to protect the private IoT data, such as body temperature and heart rate, the query processing of MusQ can be augmented with better authentication and authorization mechanisms.

APPENDIX A
CORRECTNESS PROOF

A. HASH JOIN

Theorem 3: With two subresults SR_1 and SR_2 from the subqueries and the match M from the matching schema as input, and the hash table H as intermediate property, the hash join algorithm is correct with these loop invariants:

1) for any step in the first loop, this step is applied: $0 \leq i < |SR_1|$, otherwise $i = |SR_1|$;
2) for any step in the second loop (outer), this step is applied: $0 \leq j < |SR_2|$, otherwise $j = |SR_2|$; and
3) for any step in the second loop (inner), we iterate through all members of a bucket in H with corresponding key k: $h \in H(k)$, otherwise h is $NULL$.

Proof 3 (Proof of Hash Join Approach)

The algorithm finds the result set R of the join between SR_1 and SR_2 with the matching attributes provided by the match M, which is denoted as $SR_1 \bowtie_{M.col1=M.col2} SR_2$. The algorithm supplies a hash table H as a support to yield the final result set R. The hash table maps SR_1 with each record's corresponding key of $h(sr_{1,i}[M.col1]), 0 \leq i < |SR_1|$ and hash function $h(x) \rightarrow key$. Then the algorithm joins each record of SR_2 with the members of H

that are paired with the key that satisfies $h(sr_{2,j}[M.col2])$, $0 \leq j < |SR_2|$. Thus, the algorithm finishes with R that contains $SR_1 \bowtie_{M.col1=M.col2} SR_2$.

(First Loop) Initialization: A supporting hash table is initialized as $H = \varnothing$ with $< key, value : (record) >$ structure and $i = 0$. Thus, the hash table should not contain any key-value pair yet.

(First Loop) Maintenance: Suppose at the beginning of the iteration, the invariant holds at a particular value $i < |SR_1|$ and H contains the key-value pairs from the beginning until $i-1$, as described in Equation 20. Let H' and i' denote the content of H and i at the end of the iteration, as described in Equation 21:

$$H = \{< key, sr_{1,k} > |0 \leq k < i \wedge key = h(sr_{1,k}[M.col_1]\} \tag{20}$$

$$H' = H \cup \{< key, sr_{1,i} > |key = h(sr_{1,i}[M.col_1]\}; \quad i' = i + 1. \tag{21}$$

At the end of iteration, H' must map the records of SR_1 from the beginning until i and hold the invariant. We prove this invariant in Equation 22. **(First Loop) Termination:** Because the loop is a **for** loop, it clearly terminates.

(First Loop) Correctness: The loop exits when $i = |SR_1|$. The invariant shows that H maps the records of $SR_1[0..|SR_1|-1]$ when the loop terminates.

(Second Loop (Outer)) Initialization: A result set is initialized as $R = \varnothing$ and $j = 0$. The hash table H is given from the first loop. Thus, the result set should be empty.

(Second Loop (Outer)) Maintenance: Suppose at the beginning of the iteration, the invariant holds at a particular value $j < |SR_2|$ and R contains the joint records from the beginning until $j-1$, as described in Equation 23. Let R' and j' denote the content of R and j at the end of the iteration, as described in Equation 24.

At the end of iteration, R' must contain the joint records of SR_2 with corresponding records in H from the beginning until j, and R' must hold the invariant. We prove this invariant in Equation 25:

$$\begin{aligned}
H' &= H \cup \{< key, sr_{1,i} > |key = h(sr_{1,i}[M.col_1]\} \\
H' &= \{< key, sr_{1,k} > |0 \leq k < i \wedge key = h(sr_{1,k}[M.col_1]\} \\
&\quad \cup \{< key, sr_{1,i} > |key = h(sr_{1,i}[M.col_1]\} \\
H' &= \{< key, sr_{1,k} > |0 \leq k < i+1 \wedge key \\
&= h(sr_{1,k}[M.col_1]\} \\
H' &= \{< key, sr_{1,k} > |0 \leq k < i' \wedge key = h(sr_{1,k}[M.col_1]\} \\
&\quad (proved)
\end{aligned} \tag{22}$$

$$\begin{aligned}
R &= \{sr_{2,k} \cup sr_{h,l} | 0 \leq k < j \wedge sr_{h,l} \in SR_H \\
&\quad \wedge SR_H = H(key) \wedge key = h(sr_{2,k}[M.col_2])\}
\end{aligned} \tag{23}$$

$$\begin{aligned}
R' &= R \cup \{sr_{h,l} \cup sr_{2,j} | sr_{h,l} \in SR_H \wedge SR_H = H(key) \\
&\quad \wedge key = h(sr_{2,j}[M.col_2])\}; j' = j + 1
\end{aligned} \tag{24}$$

$$\begin{aligned}
R' &= R \cup \{sr_{2,j} \cup sr_{h,l} | sr_h \in SR_H \wedge SR_H = H(key) \\
&\quad \wedge key = h(sr_{2,j}[M.col_2])\} \\
R' &= \{sr_{2,k} \cup sr_{h,l} | 0 \leq k < j \wedge sr_h \in SR_H \wedge SR_H \\
&= H(key) \wedge key = h(sr_{2,k}[M.col_2])\} \\
&\quad \cup \{sr_{2,j} \cup sr_{h,l} | sr_h \in SR_H \wedge SR_H = H(key) \\
&\quad \wedge key = h(sr_{2,j}[M.col_2])\} \\
R' &= \{sr_{2,k} \cup sr_{h,l} | 0 \leq k \leq j \wedge sr_h \in SR_H \wedge SR_H \\
&= H(key) \wedge key = h(sr_{2,k}[M.col_2])\} \\
R' &= \{sr_{2,k} \cup sr_{h,l} | 0 \leq k < j' \wedge sr_h \in SR_H \wedge SR_H \\
&= H(key) \wedge key = h(sr_{2,k}[M.col_2])\} \\
&\quad (proved).
\end{aligned} \tag{25}$$

Now we prove the invariant in the inner part of the second loop of the algorithm to find the match between $sr_{2,j}$ and $SR_H, SR_H = H(key) \wedge key = h(sr_{2,j}[M.col_2])$, where $SR_H \subseteq SR_1$.

(Second Loop(Inner)) Initialization: At the beginning of the inner loop iteration, $sr_{2,j}$, R, and SR_h are given from the outer loop. The result set of the loop is denoted as R^i. The value of h is initialized as $h = NULL$. This invariant still holds as we do not join sr_j with any record from SR_h yet.

(Second Loop(Inner)) Maintenance: Suppose at the beginning of the iteration, the invariant holds a particular value of $h \in SR_H$, and R^i contains the union of R and the joint records of $sr_{2,j}$ with the records in SR_H from the beginning until just before h, as defined in Equation 26. Let $R^{i'}$ and h' denote the content of R^i and h at the end of the iteration, as described in Equation 27. We prove this invariant in Equation 28:

$$\begin{aligned}
R^i &= R \cup \{sr_{H,l} \cup sr_{2,j} | 0 \leq l < k \wedge sr_{H,k} = h \wedge \\
&\quad sr_{H,l}[M.col_1] = sr_{2,j}[M.col_2]\}
\end{aligned} \tag{26}$$

$$R^{i'} = R^i \cup \{h \cup sr_{2,j} | h[M.col_1] = sr_{2,j}[M.col_2]\}; \quad h' = h.next \tag{27}$$

$$\begin{aligned}
R^{i'} &= R^i \cup \{h \cup sr_{2,j} | h[M.col_1] = sr_{2,j}[M.col_2]\} \\
R^{i'} &= R \cup \{sr_{H,l} \cup sr_{2,j} | 0 \leq l < k \wedge sr_{H,k} = h \wedge \\
&\quad sr_{H,l}[M.col_1] = sr_{2,j}[M.col_2]\} \cup \\
&\quad \{h \cup sr_{2,j} | h[M.col_1] = sr_{2,j}[M.col_2]\} \\
R^{i'} &= R \cup \{sr_{H,l} \cup sr_{2,j} | 0 \leq l < k \wedge sr_{H,k} = h.next \\
&\quad \wedge sr_{H,l}[M.col_1] = sr_{2,j}[M.col_2]\} \\
R^{i'} &= R \cup \{sr_{H,l} \cup sr_{2,j} | 0 \leq l < k \wedge sr_{H,k} = h' \\
&\quad \wedge sr_{H,l}[M.col_1] = sr_{2,j}[M.col_2]\} \\
&\quad (proved).
\end{aligned} \tag{28}$$

(Second Loop (Inner)) Termination: Because the loop is a **for each** loop, it clearly terminates.

(Second Loop (Inner)) Correctness: The loop exits when $h = NULL$, where $sr_{H,l}.next = h | sr_{H,l} \neq NULL$ or $SR_H = \varnothing$. The invariant shows that R^i contains the join between $sr_{2,j}$ and $SR_H[0 \ldots |SR_H|-1] \subseteq SR_1$ with the match M when the loop terminates.

(Second Loop (Outer)) Termination: Because the loop is a **for** loop, it clearly terminates.

(Second Loop (Outer)) Correctness: The loop exits when $j = |SR_2|$. The invariant shows that R contains the join

between SR_1 and $SR_2[0..|SR_2|-1]$ in the match M when the loop terminates, with the support of H as the hash table that maps SR_1 entirely (first loop's proof).

B. SORT MERGE JOIN

Theorem 4: With two subresults SR_1 and SR_2 from the subqueries and the match M from the matching schema as input, the sort-merge join algorithm is correct with these loop invariants:

1) for any step in the outer loop, this step is applied: $0 \leq i < |SR_1|$ and $0 \leq j < |SR_2|$, otherwise $i = |SR_1|$ or $j = |SR_2|$;
2) for any step in the middle loop, this step is applied: $i < |SR_1|$ and $sr_{1,i}[M.col_1] = sr_{2,j}[M.col_2]$, otherwise $i = |SR_1|$ or $sr_{1,i}[M.col_1] > sr_{2,j}[M.col_2]$; and
3) for any step in the inner loop, this step is applied: $j \leq k < |SR_2|$ and $sr_{1,i}[M.col_1] = sr_{2,k}[M.col_2]$, otherwise $k = |SR_2|$ or $sr_{1,i}[M.col_1] < sr_{2,k}[M.col_2]$.

Proof 4 (Proof of Sort Merge Join Approach)

The algorithm finds the result set R of the join between the sorted SR_1 and SR_2 with the matching attributes provided by the match M, which is denoted as $SR_1 \bowtie_{M.col1=M.col2} SR_2$. The algorithm presorts SR_1 and SR_2 according to the matching attributes $M.col_1$ and $M.col_2$, respectively, in non-descending order. Then, the algorithm joins the SR_1 and SR_2 in a way that it joins the matching subsets $SR_1[a_1..b_1] \bowtie_{M.col1=M.col2} SR_2[a_2..b_2]$, where there exists a_*, b_* such that $sr_{*,a_*-1}[M.col_*] > sr_{*,a_*}[M.col_*]$, $sr_{*,b_*}[M.col_*] < sr_{*,b_*+1}[M.col_*]$, $sr_{*,a_*}[M.col_*] = sr_{*,b_*}[M.col_*]$, $sr_{1,c_1}[M.col_1] = sr_{2,c_2}[M.col_2]$, and $a_* \leq c_* \leq b_*$. Thus, the algorithm finishes with R containing $SR_1 \bowtie_{M.col1=M.col2} SR_2$.

(Outer Loop) Initialization: A result set is initialized as $R = \varnothing$, $i = 0$ and $j = 0$. Thus, the result set should be empty.

(Outer Loop) Maintenance: The algorithm has three cases to execute the loop.

1) $i' = i+1, SR_{1,i}[M.col_1] < SR_{2,j}[M.col_2]$
2) $j' = j+1, SR_{1,i}[M.col_1] > SR_{2,j}[M.col_2]$
3) $i' = i+a, j' = j+b, a > 0, b > 0 | SR_{1,i}[M.col_1] = SR_{2,j}[M.col_2]$

We now briefly prove the invariant of the first and second cases. The invariant **maintenance** of the first and second cases, $i' = i+1$ and $j' = j+1$, respectively, are true as they are the only changes happening in the loop. The purpose of the first and second cases is to find the pair of earliest records sr_{1,i^-} and sr_{2,j^-} that have the matching value of attributes defined in M, such that ($i = 0$ OR $sr_{1,i^-}[M.col_1] > sr_{1,i^--1}[M.col_1]$) AND ($j = 0$ OR $sr_{2,j^-}[M.col_2] > sr_{2,j^--1}[M.col_2]$) AND $SR_{1,i^-}[M.col_1] = SR_{2,j^-}[M.col_2]$. We now move onto the third case's invariant.

Suppose at the beginning of the iteration, the invariant holds at a particular value $i < |SR_1|, j < |SR_2|$ and R contains the result set of the joint records from the beginning until $i-1$ and $j-1$, as described in Equation 29. Let R', i', and j' denote the contents of R, i, and j at the end of the iteration, respectively. We describe R', i', and j' in Equation 30. At the end of iteration, R' must contain the joint result set from the beginning until just before i' and j', and R' must hold the invariant. We prove this invariant in Equation 31.

$$R = SR_1[0..i-1] \bowtie_{M.col1=M.col2} SR_2[0..j-1] \tag{29}$$

$$\begin{aligned} R' = {} & R \\ & \cup SR_1[i..i+a-1] \bowtie_{M.col1=M.col2} SR_2[j..j+b-1]; \\ & i' = i+a; \\ & j' = j+b; \\ & \text{such that } sr_{1,i+a-1}[M.col_1] < sr_{1,i+a}[M.col_1] \\ & \wedge sr_{2,j+b-1}[M.col_2] < sr_{2,j+b}[M.col_2] \\ & \wedge sr_{1,i+a}[M.col_1] = sr_{2,j+b}[M.col_2] \end{aligned} \tag{30}$$

$$R' = R \cup SR_1[i..i+a-1] \bowtie_{M.col1=M.col2} SR_2[j..j+b-1]$$

$$\begin{aligned} R' = {} & SR_1[0..i-1] \bowtie_{M.col1=M.col2} SR_2[0..j-1] \\ & \cup SR_1[i..i+a-1] \bowtie_{M.col1=M.col2} SR_2[j..j+b-1] \end{aligned}$$

$$R' = SR_1[0..i+a-1] \bowtie_{M.col1=M.col2} SR_2[0..j+b-1]$$

$$R' = SR_1[0..i'-1] \bowtie_{M.col1=M.col2} SR_2[0..j'-1]$$

$$(proved). \tag{31}$$

We now prove the invariant in the middle loop of Algorithm 6 to find the match between $SR_1[i..i+a-1]$ and $SR_2[j..j+b-1]$.

(Middle Loop) Initialization: At the beginning of the middle loop iteration, $SR_1[i..i+a-1]$, $SR_2[j..j+b-1]$, i, j, $j+b$, and R are given from the outer loop. The result set of the middle loop is denoted as R^m. As the i in this loop is changed, i^- is set to store the initial value of i in this proof. This invariant still holds as none of $SR_1[i..i+a-1]$ is joined with $SR_2[j..j+b-1]$ yet.

(Middle Loop) Maintenance: Suppose at the beginning of the iteration, the invariant holds at a particular value $i < |SR_1|$ and $sr_{1,i}[M.col_1] = sr_{2,j}[M.col_2]$, and R^m contains the joint records until before i, as described in Equation 32. Let $R^{m'}$ and i' denote the content of R^m and i at the end of the iteration, as described in Equation 33:

$$\begin{aligned} R^m = {} & R \cup \{sr_{1,p} \cup sr_{2,q} | i^- \leq p < i \wedge j \leq q \leq j+b-1 \\ & \wedge sr_{1,p}[M.col_1] = sr_{2,q}[M.col_2]\} \end{aligned} \tag{32}$$

$$\begin{aligned} R^{m'} = {} & R^m \cup \{sr_{1,i} \cup sr_{2,q} | j \leq q \leq j+b-1 \\ & \wedge sr_{1,i}[M.col_1] = sr_{2,q}[M.col_2]\}; \\ i' = {} & i+1. \end{aligned} \tag{33}$$

At the end of iteration, $R^{m'}$ must contain the joint records of SR_1 from i^- until i with $SR_2[j \ldots j+b-1]$, and $R^{m'}$ must hold the invariant. We prove this invariant in Equation 34:

$$\begin{aligned} R^{m'} = {} & R^m \cup \{sr_{1,i} \cup sr_{2,q} | j \leq q \leq j+b-1 \wedge \\ & sr_{1,i}[M.col_1] \\ & = sr_{2,q}[M.col_2]\} \\ R^{m'} = {} & R \cup \{sr_{1,p} \cup sr_{2,q} | i^- \leq p < i \wedge j \leq q \leq j+b-1 \\ & \wedge sr_{1,p}[M.col_1] \end{aligned}$$

$$
\begin{aligned}
&= sr_{2,q}[M.col_2]\} \\
&\quad \cup \{sr_{1,i} \cup sr_{2,q} | j \leq q \leq j+b-1 \wedge sr_{1,i}[M.col_1] \\
&= sr_{2,q}[M.col_2]\} \\
R^{m'} &= R \cup \{sr_{1,k} \cup sr_{2,j} | 0 \leq k \leq i \wedge j \leq q \leq j+b-1 \\
&\quad \wedge sr_{1,p}[M.col_1] = sr_{2,q}[M.col_2]\} \\
R^{m'} &= R \cup \{sr_{1,k} \cup sr_{2,j} | 0 \leq k < i' \wedge j \leq q \leq j+b-1 \\
&\quad \wedge sr_{1,p}[M.col_1] = sr_{2,q}[M.col_2]\} \\
&\quad (proved). \qquad (34)
\end{aligned}
$$

Now we prove the invariant in the inner loop of Algorithm 6 to find the match between $sr_{1,i}$ with SR_2.

(Inner Loop) Initialization: At the beginning of the inner loop iteration, $sr_{1,i}$ and R^m from the middle loop and $SR_2[j..j+b-1]$, j, and $j+b$ from the outer loop are given. The result set of the loop is denoted as R^i. Then the value of k is initialized as $k = j$. This invariant still holds $sr_{1,i}$ is not joined with anything from $SR_2[j..j+b-1]$ yet.

(Inner Loop) Maintenance: Suppose at the beginning of the iteration, the invariant holds a particular value of $k < |SR_2|$, and R^i contains the union of R^m and the joint records of $sr_{1,i}$ with the records of $SR_2[j \ldots k-1]$, as defined in Equation 35. Let $R^{i'}$ and k' denote the content of R^i and k at the end of the iteration. We describe $R^{i'}$ and k' in Equation 36, and we prove this invariant in Equation 37:

$$
\begin{aligned}
R^i &= R^m \cup \{sr_{1,i} \cup sr_{2,l} | j \leq l < k-1 \wedge sr_{1,i}[M.col_1] \\
&\quad = sr_{2,l}[M.col_2]\} \qquad (35) \\
R^{i'} &= R^i \cup \{sr_{1,i} \cup sr_{2,k} | sr_{1,i}[M.col_1] = sr_{2,k}[M.col_2]\}; \\
k' &= k+1 \qquad (36) \\
R^{i'} &= R^i \cup \{sr_{1,i} \cup sr_{2,k} | sr_{1,i}[M.col_1] = sr_{2,k}[M.col_2]\} \\
R^{i'} &= R \cup \{sr_{1,i} \cup sr_{2,l} | j \leq l < k \wedge sr_{1,i}[M.col_1] \\
&\quad = sr_{2,l}[M.col_2]\} \\
&\quad \cup \{sr_{1,i} \cup sr_{2,k} | sr_{1,i}[M.col_1] = \wedge sr_{2,k}[M.col_2]\} \\
R^{i'} &= R \cup \{sr_{1,i} \cup sr_{2,l} | j \leq l \leq k \wedge sr_{1,i}[M.col_1] \\
&\quad = sr_{2,l}[M.col_2]\} \\
R^{i'} &= R \cup \{sr_{1,i} \cup sr_{2,l} | j \leq l < k' \wedge sr_{1,i}[M.col_1] \\
&\quad = sr_{2,l}[M.col_2]\} \\
&\quad (proved). \qquad (37)
\end{aligned}
$$

(Inner Loop) Termination: The **while** loop terminates when $k = |SR_2|$ or $sr_{1,i}[M.col_1] < sr_{2,k}[M.col_2]$.

(Inner Loop) Correctness: The loop exits when $k = |SR_2|$ or $sr_{1,i}[M.col_1] < sr_{2,k}[M.col_2]$ (or when $k = j+b$). The invariant shows that R^i contains the join between $sr_{1,i}$ and $SR_2[0..j+b-1]$ in the match M when the loop terminates.

(Middle Loop) Termination: The **while** loop terminates when $i = |SR_1|$ or $sr_{1,i}[M.col_1] > sr_{2,j}[M.col_2]$.

(Middle Loop) Correctness: The loop exits when $i = |SR_1|$ or $sr_{1,i}[M.col_1] > sr_{2,j}[M.col_j]$ (or when $i = i^- + a$). The invariant shows that R^m contains the join between $SR_1[0..i+a-1]$ and $SR_2[0..j+b-1]$ in the match M when the loop terminates.

(Outer Loop) Termination: The **while** loop terminates when $i = |SR_1|$ or $j = |SR_2|$.

(Outer Loop) Correctness: The invariant shows that R contains the join between SR_1 and SR_2 in the match M when the loop terminates.

REFERENCES

[1] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami, "Internet of Things (IoT): A vision, architectural elements, and future directions," *Future Gener. Comput. Syst.*, vol. 29, no. 7, pp. 1645–1660, Sep. 2013.

[2] C. MacGillivray and D. Reinsel, "Worldwide global datasphere IoT device and data forecast, 2019–2023," Int. Data Corp. (IDC), Framingham, MA, USA, Tech. Rep. US45066919, May 2019.

[3] Y. Qin, Q. Z. Sheng, N. J. G. Falkner, S. Dustdar, H. Wang, and A. V. Vasilakos, "When things matter: A survey on data-centric Internet of Things," *J. Netw. Comput. Appl.*, vol. 64, pp. 137–153, Apr. 2016.

[4] GSM Association. (2018). *IoT Big Data Framework Architecture Version 2.0*. [Online]. Available: https://www.gsma.com/iot/wp-content/uploads/2018/11/CLP.25-v2.0.pdf

[5] S. Dey, A. Chakraborty, S. Naskar, and P. Misra, "Smart city surveillance: Leveraging benefits of cloud data stores," in *Proc. 37th Annu. IEEE Conf. Local Comput. Netw. Workshops (LCN Workshops)*, Oct. 2012, pp. 868–876.

[6] B. Xu, L. Da Xu, H. Cai, C. Xie, J. Hu, and F. Bu, "Ubiquitous data accessing method in IoT-based information system for emergency medical services," *IEEE Trans Ind. Informat.*, vol. 10, no. 2, pp. 1578–1586, May 2014.

[7] F. Shrouf and G. Miragliotta, "Energy management based on Internet of Things: Practices and framework for adoption in production management," *J. Cleaner Prod.*, vol. 100, pp. 235–246, Aug. 2015.

[8] L. Atzori, A. Iera, and G. Morabito, "The Internet of Things: A survey," *Comput. Netw.*, vol. 54, no. 15, pp. 2787–2805, Oct. 2010.

[9] P. Widya, Y. Yustiawan, and J. Kwon, "A oneM2M-based query engine for Internet of Things (IoT) data streams," *Sensors*, vol. 18, no. 10, p. 3253, 2018.

[10] Y. Katsis and Y. Papakonstantinou, "View-based data integration," in *Encyclopedia of Database Systems*. New York, NY, USA: Springer, 2009, pp. 3332–3339.

[11] H. Cai, B. Xu, L. Jiang, and A. V. Vasilakos, "IoT-based big data storage systems in cloud computing: Perspectives and challenges," *IEEE Internet Things J.*, vol. 4, no. 1, pp. 75–87, Feb. 2017, doi: 10.1109/JIOT.2016.2619369.

[12] Y.-T. Liao, J. Zhou, C.-H. Lu, S.-C. Chen, C.-H. Hsu, W. Chen, M.-F. Jiang, and Y.-C. Chung, "Data adapter for querying and transformation between SQL and NoSQL database," *Future Gener. Comput. Syst.*, vol. 65, pp. 111–121, Dec. 2016.

[13] M. Zhu and T. Risch, "Querying combined cloud-based and relational databases," in *Proc. Int. Conf. Cloud Service Comput.*, Dec. 2011, pp. 330–335.

[14] B. Kolev, C. Bondiombouy, O. Levchenko, P. Valduriez, R. Jimenez-Peris, R. Pau, and J. Pereira, "Design and implementation of the CloudMdsQL multistore system," in *Proc. 6th Int. Conf. Cloud Comput. Services Sci.*, vol. 1, 2016, pp. 352–359.

[15] Z. Liu, F. Cretton, A. Le Calvé, N. Glassey, A. Cotting, and F. Chapuis, "Musyop: Towards a query optimization for heterogeneous distributed database system in energy data management," in *Proc. Int. Conf. Comput. Technol. Inf. Manage. (ICCTIM), Soc. Digit. Inf. Wireless Commun.*, 2014, pp. 1–9.

[16] F. Wang, L. Hu, J. Zhou, J. Hu, and K. Zhao, "A semantics-based approach to multi-source heterogeneous information fusion in the Internet of Things," *Soft Comput.*, vol. 21, no. 8, pp. 2005–2013, Apr. 2017, doi: 10.1007/s00500-015-1899-7.

[17] S. Watanabe and A. Nakamura, "Integrated data access to heterogeneous data stores for IoT cloud," in *Proc. 10th Asian Conf. Mod. Approaches Intell. Inf. Database Syst. (ACIIDS)*, ĐDồng Hói, Vietnam, Mar. 2018, pp. 423–433.

[18] R. Sethi, M. Traverso, D. Sundstrom, D. Phillips, W. Xie, Y. Sun, N. Yegitbasi, H. Jin, E. Hwang, N. Shingte, and C. Berner, "Presto: SQL on everything," in *Proc. IEEE 35th Int. Conf. Data Eng. (ICDE)*, Macao, China, Apr. 2019, pp. 1802–1813, doi: 10.1109/ICDE.2019.00196.

[19] M. Hausenblas and J. Nadeau, "Apache drill: Interactive ad-hoc analysis at scale," *Big Data*, vol. 1, no. 2, pp. 100–104, Jun. 2013, doi: 10.1089/big.2013.0011.
[20] S. Ceri, G. Gottlob, and L. Tanca, "What you always wanted to know about datalog (and never dared to ask)," *IEEE Trans. Knowl. Data Eng.*, vol. 1, no. 1, pp. 146–166, Mar. 1989.
[21] MySQL. *MySQL*. Accessed: Sep. 9, 2017. [Online]. Available: https://www.mysql.com/
[22] Cassandra. *Cassandra*. Accessed: Sep. 9, 2017. [Online]. Available: http://cassandra.apache.org/
[23] MongoDB. *MongoDB*. Accessed: Sep. 9, 2017. [Online]. Available: https://www.mongodb.com/
[24] M. I. Ali, F. Gao, and A. Mileo, "CityBench: A configurable benchmark to evaluate RSP engines using smart city datasets," in *Proc. 14th Int. Semantic Web Conf. (ISWC)*. Bethlehem, PA, USA: W3C, 2015, pp. 374–389.
[25] A. Doan, A. Halevy, and Z. Ives, *Principles of data integration*. Amsterdam, The Netherlands: Elsevier, 2012.
[26] E. Rahm and P. A. Bernstein, "A survey of approaches to automatic schema matching," *VLDB J.*, vol. 10, no. 4, pp. 334–350, Dec. 2001.
[27] J. Madhavan, P. A. Bernstein, and E. Rahm, "Generic schema matching with cupid," *VLDB*, vol. 1, pp. 49–58, Sep. 2001.
[28] L. Chiticariu, M. A. Hernández, P. G. Kolaitis, and L. Popa, "Semi-automatic schema integration in Clio," in *Proc. 33rd Int. Conf. Very Large Data Bases (VLDB Endowment)*, 2007, pp. 1326–1329.
[29] N. Jian, W. Hu, G. Cheng, and Y. Qu, "Falcon-ao: Aligning ontologies with falcon," in *Proc. Workshop Integrating Ontologies (K-CAP)*, 2005, pp. 85–91.
[30] E. Rahm, "Towards large-scale schema and ontology matching," in *Schema Matching and Mapping*. Berlin, Germany: Springer, 2011, pp. 3–27.
[31] M. T. Özsu and P. Valduriez, *Principles of Distributed Database Systems*. New York, NY, USA: Springer, 2011.
[32] P. A. Bernstein and L. M. Haas, "Information integration in the enterprise," *Commun. ACM*, vol. 51, no. 9, pp. 72–79, 2008.
[33] H. Hacígümüş, J. Sankaranarayanan, J. Tatemura, J. LeFevre, and N. Polyzotis, "Odyssey: A multistore system for evolutionary analytics," *Proc. VLDB Endowment*, vol. 6, no. 11, pp. 1180–1181, Aug. 2013.
[34] K. W. Ong, Y. Papakonstantinou, and R. Vernoux, "The SQL++ unifying semi-structured query language, and an expressiveness benchmark of SQL-on-Hadoop, NoSQL and NewSQL databases," *CoRR*, vol. abs/1405.3631, 2014.
[35] D. J. DeWitt, A. Halverson, R. Nehme, S. Shankar, J. Aguilar-Saborit, A. Avanes, M. Flasza, and J. Gramling, "Split query processing in polybase," in *Proc. Int. Conf. Manage. Data (SIGMOD)*. New York, NY, USA: ACM, 2013, pp. 1255–1266.
[36] A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Silberschatz, and A. Rasin, "HadoopDB: An architectural hybrid of MapReduce and DBMS technologies for analytical workloads," *Proc. VLDB Endowment*, vol. 2, no. 1, pp. 922–933, Aug. 2009.
[37] J. Chamanara, B. König-Ries, and H. V. Jagadish, "QUIS: *In-situ* heterogeneous data source querying," *Proc. VLDB Endowment*, vol. 10, no. 12, pp. 1877–1880, Aug. 2017.
[38] A. Gupta and I. S. Mumick, "Maintenance of materialized views: Problems, techniques, and applications," *IEEE Data Eng. Bull.*, vol. 18, no. 2, pp. 3–18, 1995.
[39] D. Calvanese, G. De Giacomo, D. Lembo, M. Lenzerini, and R. Rosati, "Data management in peer-to-peer data integration systems," *Global Data Manage.*, vol. 8, pp. 177–201, Jul. 2006.
[40] JSON. *Json*. Accessed: Sep. 9, 2017. [Online]. Available: http://www.json.org/
[41] WordNet. *WordNet*. Accessed: Sep. 9, 2017. [Online]. Available: https://wordnet.princeton.edu/
[42] C. A. Furia, B. Meyer, and S. Velder, "Loop invariants: Analysis, classification, and examples," *ACM Comput. Surveys*, vol. 46, no. 3, pp. 34:1–34:51, Jan. 2014, doi: 10.1145/2506375.
[43] ANTLR. *Antlr(Another Tool for Language Recognition)*. Accessed: Sep. 9, 2017. [Online]. Available: http://www.antlr.org/
[44] Thi-Van-Anh Nguyen, S. Bimonte, L. d'Orazio, and J. Darmont, "Cost models for view materialization in the cloud," in *Proc. Joint EDBT/ICDT, Workshops*, D. Srivastava and I. Ari, Eds. Berlin, Germany: ACM, Mar. 2012, pp. 47–54, doi: 10.1145/2320765.2320788.

HANI RAMADHAN received the master's degrees from the Informatics Engineering Department, Institut Teknologi Sepuluh Nopember, Indonesia, and the Complex System and Interaction Department, Universite de Technologie de Compiegne, France, in a joint-degree program. He is currently pursuing the Ph.D. degree with the Big Data Department, Pusan National University, South Korea. His research interests include data mining, machine learning, and computer vision.

FITRI INDRA INDIKAWATI received the bachelor's degree from the Department of Informatics, Institut Teknologi Sepuluh Nopember, Indonesia, in 2010, and the master's degree from the Department of Big Data, Pusan National University, South Korea, in 2017. She was a Senior Software Engineer with Samsung Electronics Indonesia, from 2010 to 2015. She is currently a Junior Lecturer and a Researcher with Ahmad Dahlan University. Her main research interests include the IoT data management and analytics, data mining, and machine learning.

JOONHO KWON (Member, IEEE) received the B.S., M.S., and Ph.D. degrees from the School of Electrical Engineering and Computer Engineering, Seoul National University, South Korea, in 1999, 2001, and 2009, respectively. He is currently a Professor with the School of Computer Science and Engineering, Pusan National University, South Korea. His current research interests include big data management and analytics, NoSQL databases, the IoT data query processing, XML indexing and query processing, Web services, RFID data management, and serious games.

BONYONG KOO received the Ph.D. degree in naval architecture and ocean engineering from Seoul National University, South Korea, in 2012. He is currently an Assistant Professor with the School of Mechanical Convergence System Engineering, Kunsan National University, South Korea. He has 15 years of industrial experience in naval shipbuilding and offshore engineering. His research interests include optimization and measurement technologies.