Data & Knowledge Engineering 146 (2023) 102185

Contents lists available at ScienceDirect

Data & Knowledge Engineering

journal homepage: www.elsevier.com/locate/datak

Logical big data integration and near real-time data analytics

Bruno Silva [a], José Moreira [a,b], Rogério Luís de C. Costa [c,*]

[a] Institute of Electronics and Informatics Engineering (IEETA), LASI, University of Aveiro, Aveiro, 3810-193, Portugal
[b] Department of Electronics, Telecommunications and Informatics (DETI), University of Aveiro, Aveiro, 3810-193, Portugal
[c] Computer Science and Communication Research Centre (CIIC), Polytechnic of Leiria, Leiria, 2411-901, Portugal

A R T I C L E I N F O

Keywords:
Big data integration
Distributed databases
Near real-time OLAP

A B S T R A C T

In the context of decision-making, there is a growing demand for near real-time data that traditional solutions, like data warehousing based on long-running ETL processes, cannot fully meet. On the other hand, existing logical data integration solutions are challenging because users must focus on data location and distribution details rather than on data analytics and decision-making.

EasyBDI is an open-source system that provides logical integration of data and high-level business-oriented abstractions. It uses schema matching, integration, and mapping techniques, to automatically identify partitioned data and propose a global schema. Users can then specify star schemas based on global entities and submit analytical queries to retrieve data from distributed data sources without knowing the organization and other technical details of the underlying systems.

This work presents the algorithms and methods for global schema creation and query execution. Experimental results show that the overhead imposed by logical integration layers is relatively small compared to the execution times of distributed queries.

1. Introduction

Traditionally, data analytics have been executed on large data warehouses whose data is periodically extracted from OLTP databases and loaded as part of a long-running ETL (extract, transform, and load) process. However, the landscape is changing rapidly. Near real-time data analytics is becoming a requirement in many of today's IT contexts [1]. For instance, in many IoT-based systems, multiple sensors generate data streams that need to be processed and analyzed on the fly to support decisions based on the most recent data. Applications of near real-time analytics also include anomaly detection and cybersecurity [2], Industry 4.0 [3], and urban planning [4].

However, storing and processing large volumes of data from heterogeneous sources in near real-time is challenging for traditional data management systems [5]. Over the years, several different data management systems appeared and the trend became to choose the most suitable platform for each use case. It means that current IT environments tend to have distinct platforms being used simultaneously, such as relational databases, NoSQL databases or distributed event streaming platforms [6]. Indeed, the use of distributed storage tools (including NoSQL databases) reduces the time needed for data retrieval of large datasets from heterogeneous data sources [5].

Many systems have been developed to enable the execution of distributed queries over heterogeneous data sources (e.g., CloudMdsQL [7], Presto [8], Apache Drill [9]). However, they are complex and need lots of work to configure and create queries. Frequently, the users must know the details regarding the data distribution and the internal organization of the data in their sources.

* Corresponding author.
E-mail addresses: bsilva@ua.pt (B. Silva), jose.moreira@ua.pt (J. Moreira), rogerio.l.costa@ipleiria.pt (R.L.d.C. Costa).

https://doi.org/10.1016/j.datak.2023.102185
Received 10 November 2022; Received in revised form 21 April 2023; Accepted 6 May 2023
Available online 12 May 2023

Moreover, they were not designed for data analytics and decision-making. There is a lot of work on solving specific problems related to data integration, but creating solutions that go all the way in the data-to-insights pipeline, i.e., from raw data to the desired outcome, remains an open issue [10].

EasyBDI [11] is an open-source system for logical data integration that hides the complexity of data distribution and the particularities of organization of the data in the local sources from the domain experts. Users may just plug data sources into the system and EasyBDI automatically extracts their local data organization and proposes a Global Schema by applying well-known algorithms for schema mapping and integration. The result is a logical and integrated representation of the underlying data sources. Then, the system automatically identifies data partitioning and maps global entities to the corresponding data snippets in the distributed sources. Users may fine-tune the proposed schema and create complex mappings to represent high-level abstractions. Finally, data scientists and analysts may define star schemas over the global abstraction layer and submit queries without writing code. Hence, users may submit analytical queries on fresh data based on high-level subject-oriented entities without worrying about data location and partitioning, or the technologies used in each data source.

In this work, we define the main concepts related to the multi-layered logical integration used in EasyBDI. We present the algorithms for automatic global schema creation (including schema matching, mapping, and integration) and distributed query generation. These are based on the current state of the art, and demonstrate how those concepts are used to add new functionality within EasyBDI. To the best of our knowledge, this is the first solution for logical data integration, guaranteeing location and fragmentation transparency, dealing with different database models, and providing high-level star-schema abstractions at the same time. We also detail two case studies, one based on benchmark data and queries and the other based on real-world data, and present performance evaluation results.

The following section reviews background and related works. Section 3 defines concepts as used by EasyBDI. Section 4 presents the main algorithms used for global schema creation and analytic query execution. Section 5 presents use cases and experimental results. Section 6 concludes the paper.

2. Background and related work

In traditional data warehouses, data is commonly materialized using an ETL process to extract data from each data source, transform those data into a consolidated schema and load it into a centralized database. However, this process may be incompatible with some use cases that require fresh data, such as IoT. The logical data integration approach is an alternative that may be more suitable for such needs. Queries are made directly to each data source, and only the necessary data is retrieved [12], eliminating the need for time-consuming data extraction and loading processes before data analysis can occur. Moreover, querying the data it resides guarantees that the information is up-to-date.

This section begins with a theoretical presentation on concepts and methods for creating global schemas in distributed databases. Next, the latest approaches to logical data integration and near real-time data analytics in modern data management systems are presented. It ends with a selection of research gaps considering the objectives and recommendations of the literature and the characteristics of the most recent frameworks for logical data integration.

2.1. Global schema generation

In the logical integration architecture, a global schema is created to consolidate data from various sources. This schema provides an integrated view of semantically similar data, allowing for logical relationships between data from different sources. To create a global schema, similarities between schema elements (e.g., tables and attributes) must be identified (schema matching), and multiple local schema elements must be combined into a single global schema element (schema integration). It is also required to create a mapping between global and local schema elements to enable the translation of global queries into local queries and assemble the answers into a single result (schema mapping).

2.1.1. Schema matching

The main goal of schema matching is to detect objects stored in different data sources that contain data belonging to the same conceptual domain. Usually, the algorithms compute a measure that estimates the similarity between two schema elements. If the similarity is higher than a certain threshold, it is considered that there is a match. Schema matching techniques include looking at data structures (*schema-based matching*) and looking at the format or content of instances of data (*instanced-based matching*).

Schema-based matching techniques use metadata, such as names, data types, inter-table relationships, and overall schema structure, to compare objects [13]. Similarity assessment can occur at the *element level*, considering names and data types, or at the *structure level*, considering the relationships between schema elements [14,15]. Linguistic techniques use the name of the schema elements to find how similar they are using measures such as *n*-grams, Jaccard index or Levenshtein distance. Thesaurus with synonyms and hypernyms can also be used to deal with semantically related concepts, such as ‘car’ and ‘automobile’ [13,14]. On the other hand, constraint-based techniques use information such as data type, primary keys, foreign keys and unique constraints to identify the relationships between schema elements.

Instance-based matching is a technique that analyzes data instances to identify matches between schema elements. This technique is suitable for semi-structured data sources where schema metadata is limited [13,15]. Instance-based matching can use either linguistic or constraint-based techniques. In linguistic techniques, the most common keywords or word combinations in instances are identified to determine the similarity between schema objects. On the other hand, constraint-based techniques analyze the ranges of values or try to identify patterns in the data to identify matches. The instance-based and constraint-based matching techniques can be combined to achieve better performance.

2.1.2. Schema integration and mapping

To build a global schema, it is necessary to merge all local schema elements identified as semantically identical in the schema-matching process into a global one with the same semantics.

The mappings between global and local schema elements enable the automatic translation of global queries into local queries. Schema mappings can be created from the local schemas to the global schema, known as Local-as-View (LAV), or from the global schema to the local ones, known as Global-as-View (GAV) [16]. GAV allows easier creation of mappings, but adding a new data source forces the redefinition of some mappings. On the other hand, LAV makes adding new data sources easier but makes the update of the mappings and query generation more difficult [16,17].

Typically, users create schema mappings manually, but ideally, they would only provide high-level information and the system would generate mappings automatically [18]. For instance, a user interface is used to create connections between schema elements and the system generates additional details for each mapping accordingly [19]. There are also solutions to create schema mappings through examples, meaning that the users must provide an incomplete set of schema mappings and the system creates the remaining ones [20,21]. In other systems, such as MusQ [22], the mappings between local and global schemas are established using a language to set the semantic correspondences between tables.

2.2. Logical data integration approaches

The logical approach uses an architecture known as mediator/wrapper in the literature. In this approach, a mediator creates and maintains a Global Conceptual Schema (GCS), also called a global schema or mediated schema. This schema provides an integrated view of each data source's schema, known as Local Conceptual Schema (LCS) or local schema. The mediator can receive a query defined on the global schema entities and break it down according to the data sources' schemas. The decomposed queries are then sent to wrappers that communicate and send the queries to the data sources [12]. The queries are executed in the data sources and the results are returned to the mediator. The mediator combines the results returned by each data source and prepares the final result according to the global schema. This approach is followed in systems like BigDAWG [23] and MusQ [22].

BigDAWG [23] is an open-source polystore system that allows users to configure islands for each type of data model they want to query. Users can connect data sources of the same DBMS on the same island. However, when combining data from different islands, users must use special operators to deal with each island's different data models and query languages. MusQ [22] is a A multi-Store Query System for IoT Data that uses schema-matching techniques to detect semantically similar schema elements and generate a global schema semi-automatically. It uses a Datalog-Like Language query language called MQL.

Query engines like CloudMdsQL, Apache Drill, and Trino (formerly Presto), can perform distributed querying across a wide range of data sources and DBMS. These systems use an SQL-like language to specify queries that are subsequently decomposed and sent to relational and NoSQL data sources. The communication and translation of queries into the native language of each data source are managed by wrappers [7,8,24]. They provide a simplified way to access raw data across different data sources, as users do not have to configure a global schema. However, users need to know the details of the schemas and how to combine data from each data source.

2.3. Near real-time data analytics

Traditional data warehousing involves periodically running data extraction, transformation, and loading (ETL) tasks. However, this approach is inadequate in applications that demand near real-time data, including IoT, e-commerce, health systems, and stock brokering [25]. Besides, ETL operations can interrupt analytical operations, posing a challenge for systems that operate 24 h a day [26]. Near real-time warehouses offer a solution by ensuring that fresh data is accessible without system downtime through lightweight and frequent ETL processes [27].

[25] proposed an architecture aimed at enabling the concurrent execution of analytical queries and continuous data integration with minimal performance degradation. To achieve this, they employed temporary tables without constraints or indexes, allowing for rapid insertion of data updates from ETL processes. These temporary tables contain only recent data and must be relatively small in size. Analytical queries must explicitly specify the tables to be queried, which could be the original tables with historical data, the temporary tables with recent data, or both. A similar approach is used in [27] where the data is split into partitions for easier separation of recent data from older data. Likewise, [28] present an architecture featuring a dynamic module that holds a limited amount of the most recent data and a static module that stores the remaining historical data. The star schema tables in both modules are identical, enabling seamless integration between the two. Analytical queries are automatically subdivided into subqueries sent to both modules by a merger component without user intervention.

2.4. Requirements and evaluation

Table 1 places side by side the latest proposals for logical data integration and real-time data analytics, considering a selection of desirable features identified in the literature [2,5,11,14,28,29].

This table and the presentation in the previous subsections highlight that existing proposals do not fully and transparently support logical integration in modern Big Data systems. Namely, platforms like Big DAWG and MusQ are not able to deal with different data models and query languages, while distributed query engines like Trino, Apache Drill or CloudMDSQL, do not use a global schema, requiring users to know about the location of the data sources, partitioning and semantic relationships. In addition, near real-time data analytics platforms implement continuous data integration processes through data partitioning or temporary tables but pose restrictions on the size of the tables and partitions holding the most recent data. Furthermore, the separation between historical and current data is non-transparent to the users.

Table 1
Evaluation of existing tools.

	BigDAWG	Presto	Drill	CloudMdsQL	MusQ	Santos & Bernardino(2008)	Cuzzocrea et al. (2009)
Distributed Data Querying	yes	yes	yes	yes	yes	n/a	n/a
Heterogeneous Data Querying	yes	yes	yes	yes	yes	no[a]	no[a]
Provides Location Transparency	yes	no	no	yes	yes	yes	yes
Automatic Global Schema Creation	n/a	n/a	n/a	yes	n/a	n/a	n/a
Tools Enabling Analytical Queries	no	no	no	no	partial	yes	yes
Provides a Guided Star Schema Creation	n/a	n/a	n/a	n/a	no	no	no
Autonomously Deals with Partitioning	no	no	no	no	no	yes	yes
Specification of Global Constraints	no	n/a	n/a	n/a	no	yes	yes
No mandatory ETL/ view materialization	yes	yes	yes	yes	yes	no	no

[a]Requires ETL processes to query heterogeneous data.

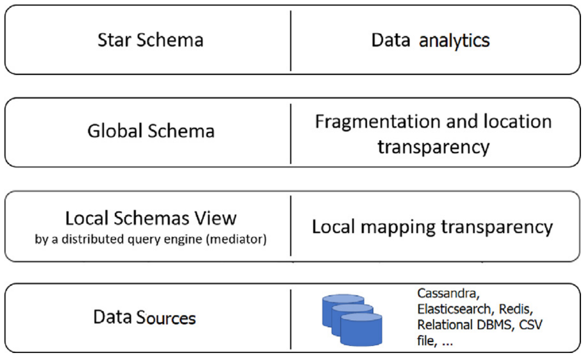

Star Schema
Data analytics
Global Schema
Fragmentation and location transparency
Local Schemas View
by a distributed query engine (mediator)
Local mapping transparency
Data Sources
Cassandra, Elasticsearch, Redis, Relational DBMS, CSV file, ...

Fig. 1. Layered architecture: overview.

3. Concepts and definitions

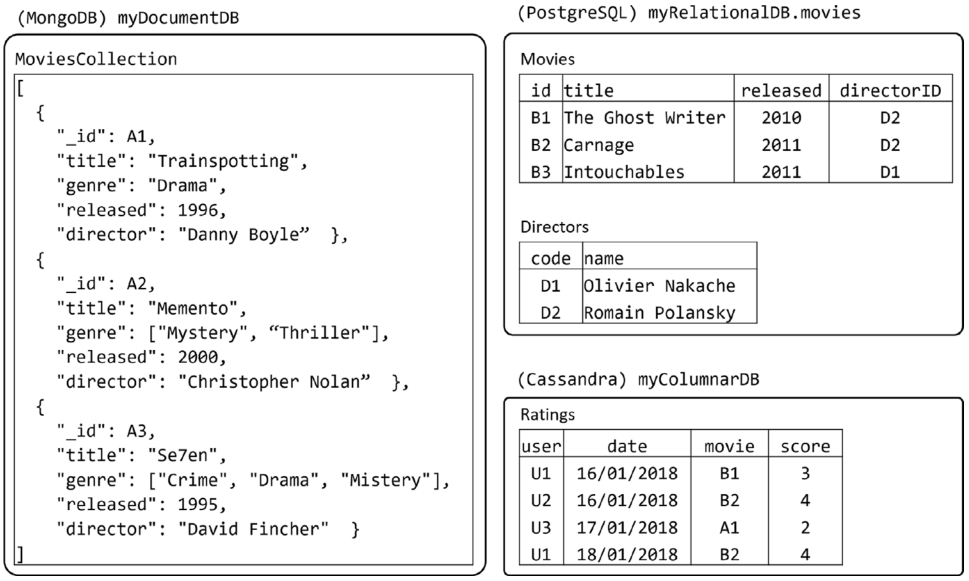
This section presents the theoretical foundations for logical data integration in EasyBDI. The aim is to design a system with several levels of transparency, namely, data requests should not need to specify where data is located, semantically related data can be fragmented in various sources but can be seen as a single entity, and data requests should be agnostic regarding the data models and query languages of the underlying data sources. Thus, EasyBDI is logically organized into four layers, as represented in Fig. 1.

The first layer represents the data sources. Each data source stores a part of the data in a distributed system. The second layer, the Local Schemas View, provides location mapping transparency and high-level relational abstractions for the underlying data sources, but it is still necessary to know how to deal with data fragments and their locations. The third layer is the Global Schema, which offers a unified and logical view of the underlying data sources. It enables users to write queries using global concepts without having to worry about data location and fragmentation. The fourth layer, the Star Schema, supports data analytics so that users can use concepts like facts, measures and dimensions to create star schemas based on global entities. This enables querying data based on high-level abstractions, such as global star schema entities.

EasyBDI can create a global schema using data integration techniques to detect entities representing semantically similar concepts and integrate them into a single entity. However, users can also manually edit the global schema. The system can also create and run the required commands to retrieve the requested data automatically, enabling a seamless and transparent data integration process for data analytics. For this, we need to define the concepts to represent the data in each abstraction layer, as well as the semantic mappings and the data transformation processes between layers.

Data source. A data source (*DS*) can be a relational database, a NoSQL database, or even a raw file. Data sources may offer composite data structures with several organization levels. Thus, to access specific data, we may use composition, e.g., *DS* = *Source.Schema*.

Considering the example in Fig. 2, the data sources are a MongoDB collection (*MongoDS* = *myDocumentDB*) with data about movies produced in the United States or the United Kingdom, a PostgreSQL database schema (*PostgresDS* = *myRelationalDB.movies*) with data about movies produced in other countries and a Cassandra database (*CassandraDS* = *myColumnarDB*) with movie ratings.



(MongoDB) myDocumentDB

MoviesCollection

```
[
  {
    "_id": A1,
    "title": "Trainspotting",
    "genre": "Drama",
    "released": 1996,
    "director": "Danny Boyle”  },
  {
    "_id": A2,
    "title": "Memento",
    "genre": ["Mystery", “Thriller"],
    "released": 2000,
    "director": "Christopher Nolan”  },
  {
    "_id": A3,
    "title": "Se7en",
    "genre": ["Crime", "Drama", "Mistery"],
    "released": 1995,
    "director": "David Fincher"  }
]
```

(PostgreSQL) myRelationalDB.movies

Movies

id	title	released	directorID
B1	The Ghost Writer	2010	D2
B2	Carnage	2011	D2
B3	Intouchables	2011	D1

Directors

code	name
D1	Olivier Nakache
D2	Romain Polansky

(Cassandra) myColumnarDB

Ratings

user	date	movie	score
U1	16/01/2018	B1	3
U2	16/01/2018	B2	4
U3	17/01/2018	A1	2
U1	18/01/2018	B2	4

Fig. 2. Example with three data sources in MongoDB, PostgreSQL and Cassandra.

Local schema view. A *Local Schema View* (*LSV*) provides an abstraction regarding the internal data model, formats and specific querying capabilities of a data source. For that purpose, non-relational data sources are abstracted as relational databases with the help of a mediator, as detailed in Section 4.2. This means that data fragments in a *DS* are mapped to entities (E) in an *LSV*. Each *DS* has a corresponding *LSV* composed of several entities.

$$LSV = \{E_1, E_2, \dots, E_m\} \tag{1}$$

Recalling the example in Fig. 2, each local schema view is mapped to a data source as follows:

- *MongoDS* ↤ *MongoLSV = {MoviesCollection}*,
- *PostgresDS* ↤ *PostgresLSV = {Movies, Directors}*,
- *CassandraDS* ↤ *CassandraLSV = {Ratings}*.

Hereafter, each entity in a local schema view is seen as a relation. Thus, an entity $E(A_1, A_2 ... A_n)$ has 1 or more attributes *(A)*, and each attribute has a data type denoted *type(A)*. There may be attributes in a data source that do not match directly with an attribute in a local schema view. These are known as virtual attributes and can be created using different methods. For example, virtual attribute values can be set to a constant or null, computed using built-in formulas, or defined as a query to the data source, as explained in Section 5. To refer to a particular entity or attribute in a local schema view, we use the notation *LSV.E* for entities and *LSV.E*.A for attributes.

The local schema views from all data sources is called local schema (L).

$$L = \{LSV_1, LSV_2, \dots, LSV_p\} \tag{2}$$

To make the notation and the formulation easier, we will denote an entity from a local schema view (*LSV.E*) as E^L and the set of all entities of all local schema views as *LE*.

$$LE = \bigcup_i \bigcup_j LSV_i.E_j \tag{3}$$

In the example, the set of local schema views is *L = { mongoLSV, PostgresLSV and CassandraLSV}*, and the set of all local entities is *LE = {mongoLSV.MoviesCollection, PostgresLSV.Movies, PostgresLSV.Directors, CassandraLSV.Ratings}*.

Global schema. A global schema (G) provides a conceptual view of the data that ensures location, mapping and fragmentation transparency. It is defined as:

$$G = \{E_1^G, E_2^G, \dots, E_r^G\} \tag{4}$$

such that a global entity, denoted as $E^G(A_1, A_2 ... A_q)$, can integrate data fragments from multiple local data sources.

The goal is to enable users to use global entities without needing to know the physical location of the data or how to map and integrate data from multiple sources. To achieve this, we need to define the semantic mappings between the global and local schemas. The mappings are defined at two levels of granularity. The first level is the mapping between global and local entities, defined as follows:

$$E^G \leftarrow\!\!\dashv \{E_1^L, E_2^L, \dots, E_s^L\} \;\;|\;\; E_i^L \in LE \tag{5}$$

In the example, we can define two global entities as follows:

- $Movies^G \leftarrow\!\!\dashv \{MoviesMNG^L, MoviesPG^L, DirectorsPG^L\}$,
- $Ratings^G \leftarrow\!\!\dashv \{RatingsCSD^L\}$,

such that,

- $MoviesMNG^L = MongoLSV.MoviesCollection$,
- $MoviesPG^L = PostgresLSV.Movies$,
- $DirectorsPG^L = PostgresLSV.Directors$,
- $RatingsCSD^L = CassandraLSV.Ratings$.

Second, at the attribute level, the mappings are achieved through queries on entities from one or more local schema views. There are four types of mappings, which depend on how the data of interest are partitioned across the data sources:

1. In a Simple Mapping (or 1-to −1 Mapping), the data are not partitioned, so a global entity is mapped to a single entity in a local schema view. Each attribute of a global entity corresponds to a single attribute of a local entity. It is not required for all attributes in the local entity to appear in the global entity. Hence, a simple mapping is defined as:

$$E^G(A_1, A_2, \dots, A_q) \subseteq E^L(A_{x_1}, A_{x_2}, \dots, A_{x_q}) \;\;|\;\; E^L \in LE \tag{6}$$

For instance, the entity *Ratings* in the Cassandra database can be mapped to a global relation as follows:

- $Ratings^G(dateTime, userID, movieID, rating) \subseteq Ratings^L(date, user, movie, score)$

and the resulting instance is depicted in Fig. 3.

2. A Vertical Mapping is defined as a join between local entities:

$$E^G(A_1, A_2, \dots, A_q) \equiv E_1^L \bowtie_{cond_1} E_2^L \bowtie_{cond_2} \dots E_s^L \;\;|\;\; E_i^L \in LE \tag{7}$$

such that, $cond_x$ is a join condition. This operation is often followed by a projection to select the attributes of interest.
For instance, to create a single relation representing the movies and their directors in the PostgreSQL database, a vertical mapping between the local entities MoviesPG^L and DirectorsPG^L is established as follows:

- $MovieDirectors \equiv MoviesPG^L \bowtie_{directorID=code} DirectorsPG^L$

and the result is depicted in Fig. 4.

3. An Horizontal Mapping is defined as the union of two or more local entities:

$$E^G(A_1, A_2, \dots, A_q) \equiv \bigcup_i E_i^L(A_{x_1}, A_{x_2}, \dots, A_{x_q}) \;\;|\;\; E_i^L \in LE \tag{8}$$

For instance, to create a global entity with all movies it is necessary to perform the union of two local entities as follows:

- $Movies^G(id, title, released, director) \equiv \pi_{\_id, title, releaseDate, director}\ MoviesMNG^L \cup \pi_{\_id, title, released, name}\ MoviesDirectors$

The resulting instance is depicted in Fig. 5.

4. In a Transformation Mapping, a global entity is defined as the result of user-defined transformations denoted as follows:

$$E^G \leftarrow\!\!\dashv transform(\{E_1^L, E_2^L, \dots, E_r^L\}) \;\;|\;\; E_i^L \in LE \tag{9}$$

such that, *transform* is a query that cannot be represented as any of the three previous definitions.
For instance, it is possible to create simple transformations to decompose dates into their simplest parts, as follows:

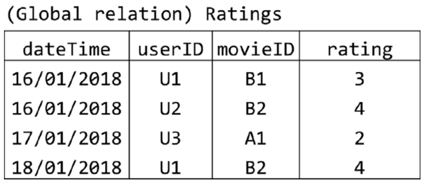
- $DatesD^G(date, yyyy, mm)^G \equiv \pi_{time, year(time), month(time)}\ RatingsCSD^G$

where *year* and *month* are functions returning parts of a timestamp. The resulting instance is depicted in Fig. 6. More complex transformations, such as *unpivot* operations, are exemplified in Section 5.1.

In Eqs. (6) – (9) it is implicit that there is a mapping between each attribute of a global entity with attributes of one or more local entities:

$$E^G.A \leftarrow\!\!\dashv \{E_1^L.A_{x_1}, E_2^L.A_{x_2}, \dots, E_s^L.A_{x_s}\} \tag{10}$$

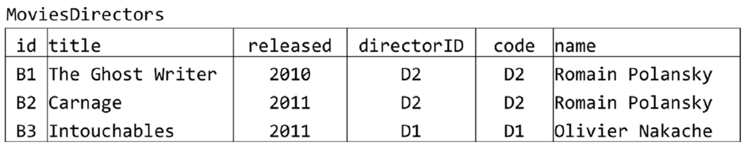
hereafter denoted as $map(E^G.A)$. For instance, $map(Movies^G.director) \leftarrow\!\!\dashv$ {MoviesMNG^L.director, DirectorsPG^L.name} We will also denote the set of mappings for all attributes of a global entity as $map(E^G)$. The function *corr(X)* returns the right-hand part of a mapping, e.g., $corr(map(E^G))$ returns $\{E_1^L, E_2^L, \dots, E_r^L\}$ and $corr(map(E^G.A))$ returns $\{E_1^L.A_{x_1}, E_2^L.A_{x_2}, \dots, E_s^L.A_{x_s}\}$, or, in the example, $corr(map(Movies^G.director))$ returns {MoviesMNG^L.director, DirectorsPG^L.name}.



(Global relation) Ratings

dateTime	userID	movieID	rating
16/01/2018	U1	B1	3
16/01/2018	U2	B2	4
17/01/2018	U3	A1	2
18/01/2018	U1	B2	4

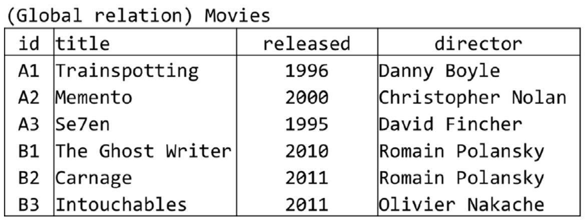
Fig. 3. The result of a simple mapping to create a global relation with the ratings in the Cassandra Database.



MoviesDirectors

id	title	released	directorID	code	name
B1	The Ghost Writer	2010	D2	D2	Romain Polansky
B2	Carnage	2011	D2	D2	Romain Polansky
B3	Intouchables	2011	D1	D1	Olivier Nakache

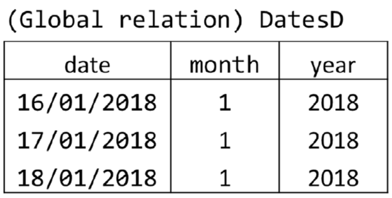
Fig. 4. The result of a vertical mapping to create a global relation joining the movies and directors data in the PostgreSQL database.



(Global relation) Movies

id	title	released	director
A1	Trainspotting	1996	Danny Boyle
A2	Memento	2000	Christopher Nolan
A3	Se7en	1995	David Fincher
B1	The Ghost Writer	2010	Romain Polansky
B2	Carnage	2011	Romain Polansky
B3	Intouchables	2011	Olivier Nakache

Fig. 5. The result of an horizontal mapping to create global entity with all the movies in the MongoDB and PostgreSQL databases.



(Global relation) DatesD

date	month	year
16/01/2018	1	2018
17/01/2018	1	2018
18/01/2018	1	2018

Fig. 6. The result of an horizontal mapping to create global entity with all the movies in the MongoDB and PostgreSQL databases.

At this level of abstraction, users can write queries on global relations. For instance, *Movies*^{G} is a global entity representing all movies in MongoDB and PostgreSQL and the users, e.g., programmers, do not need to know that these data come from different data sources.

Star schema. A star schema (S) is composed by a set of global entities, however some restrictions apply, namely, one of the global entities must represent the facts, denoted (E^F), and one or more global entities are denoted as dimensions (E^D). Thus, a star schema is defined as:

$$S = E^F \cup \{E_i^D\}, i \geq 1. \tag{11}$$

such that $E^F \in G$, $\{E_i^D\} \subset G$ and $E^F \cap \{E_i^D\} = \emptyset$. A global entity cannot represent the facts and a dimension at the same time.

The attributes of a fact relation can represent measures (A^M), or participate in the definition of foreign keys to dimension relations. The attributes of a dimension can represent the primary key or regular attributes. Hierarchies are not defined explicitly in this model. The definitions of primary keys and foreign keys are equivalent to those used in the relational model.

For instance, in the example above, it is possible to define a star schema where *Ratings*^{F} is the fact table and {*DatesD*^{D}, *Movies*^{D}} are the dimensions. There is a single measure in the fact table (A^M= {*Ratings*^{F}*.rating*}). The relationships between the fact and the dimension tables are *Ratings*^{F}*.movieID* references *Movies*^{D}*.id* and *Ratings*^{F}*.date* references *Dates*^{D}*.date*. Note that the user dimension has been omitted to keep the example in Fig. 2 short.

Query. A user query (Q) is defined over a star schema S (or a global schema G) and is not necessarily based on any language (e.g., it may be submitted via a GUI or an API). The user query is defined as a set of selected attributes (Q^A) and operations, which may include aggregations (AG), grouping functions (GF), individual (F) and aggregate filters (AF), pivoting (P) and ordering clauses

(OB), such that:

$$\begin{aligned} Q^A &= \{E^S_1.A_{x_1}, E^S_2.A_{x_2}, \dots, E^S_s.A_{x_s}\} \ \mid \ E^S_i \in S \\ Q &= (Q^A, AG, GF, AF, F, P, OB) \end{aligned} \tag{12}$$

The set of star schema entities referenced in Q^A is defined as Q^E:

$$Q^E = \{E^S_1, E^S_2, \dots, E^S_n\} \ \mid \ E^S_i \in S \tag{13}$$

In the Query Q definition, aggregations (AG) and grouping functions (GF) are used to specify how to group multiple tuples into a single one. Aggregations (AG) must contain a set of attributes whose values should not be grouped, and grouping functions (GF) represent the functions (e.g., *Sum*, *Count*, and *Average*) and the attribute on which they are applied.

Filters may contain a set of logical clauses that are used to filter the data in the resulting set. They may be specified over attribute values (F), as in a WHERE clause in a relational database, or over aggregation results (AF), as in a HAVING clause in a relational database. The Pivot operation (P) contains a set of attributes whose values should be viewed as columns instead of rows. Finally, Ordering clauses (OB) are also based on a set of attributes and how their values should be ordered (i.e., ascending or descending) in the query execution results. Each attribute A_i referenced in AG, GF, AF, F, P and OB must be an attribute in S.

4. Methods and algorithms

EasyBDI is composed by several modules, including a *Query Execution Manager* to translate the user input into a valid SQL code, a *Multidimensional Schema Manager* to manage the mapping between global schema entities and star schema entities, a *Configuration Manager* to manage the configuration files needed to connect to the data sources, and a *Schema/Metadata Storage* to store the metadata about the schemas (global schema, local schema views, and star schemas) and the mapping between schemas. It also includes a GUI through which users interact with the system to configure data sources, generate and refine global schemas, and create star schemas and global queries over such schemas. This section focuses on the *Multidimensional Schema Manager* and *Query Execution Manager*, and presents the algorithms implemented to help in the automatic creation of a global model (schema matching, integration and mapping) in a distributed data management system, and the methods used to translate the global queries into queries on the local data sources.

4.1. Creating a global schema

Creating a mapping between a global schema and several local schemas is a time-consuming task, particularly when there are many entities and attributes involved. Thus, EasyBDI implements several methods to automate these tasks, which users can later edit using a GUI to obtain the final design of a global schema. The procedures implemented to perform the schema matching, schema integration and schema mapping steps are as follows.

4.1.1. Schema matching

The first step consists of computing the similarity between the names of entities in the local schema view using the Levenshtein distance. The pairs of entities with name similarity above a predefined threshold are considered as matching. The correspondence between local entities is transitive, i.e., two entities that do not have a direct match can still be part of the same group of matching entities.

Consider a local schema view LSV_1, which contains three entities $LSV_1 = \{employees, employees\_info, inventory\}$. Fig. 7 exemplifies using the proposed schema-matching algorithm on LSV_1. The first step is to compare the names of pairs of entities based on the Levenshtein distance. In the example described in Step 2 of Fig. 7, entities *employees* and *employees_info* form a possible match, as their name similarity is above the threshold value of 0.6. Entity *inventory* has no matches. The threshold value was found by experimentation.

After computing the similarity between the names of entities, the proposed method aims to determine the correspondences between the attributes of matching entities.

The *attribute similarity* ($ATSim$) between two attributes $E^L_i.A_p$ and $E^L_j.A_q$ is calculated as the weighted sum of the *data type similarity* ($data\_type\_sim$), the *attribute name similarity* ($name\_sim$), and the *primary key similarity* (pk_sim), as defined in Eq. (14).

$$\begin{aligned} ATSim(E^L_i.A_p, E^L_j.A_q) &= data\_type\_sim(E^L_i.A_p, E^L_j.A_q) \times w_1 \\ +name\_sim(E^L_i.A_p, E^L_j.A_q) \times w_2 &+ pk\_sim(E^L_i.A_p, E^L_j.A_q) \times w_3 \end{aligned} \tag{14}$$

The data type similarity ($data\_type\_sim$) between two attribute is computed as:

- The data types of both attributes ($E^L_i.A_p$, $E^L_j.A_q$) are the same:
  $data\_type\_sim(E^L_i.A_p, E^L_j.A_q) = 1$.
- Consider data types may be grouped into categories (e.g., numeric, string, boolean, and date and time) depending on their domains. If attributes have different data types that belong to the same category (e.g., double and integer types are both numeric) then:
  $data\_type\_sim(E^L_i.A_p, E^L_j.A_q) = 0.8$.

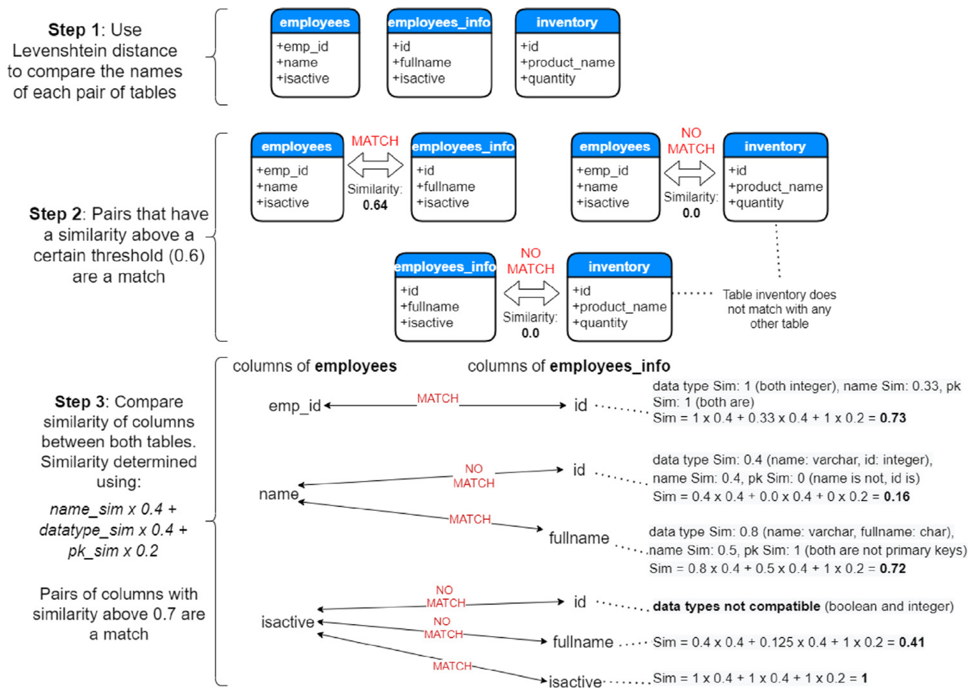

Step 1: Use Levenshtein distance to compare the names of each pair of tables
employees
+emp_id
+name
+isactive
employees_info
+id
+fullname
+isactive
inventory
+id
+product_name
+quantity
Step 2: Pairs that have a similarity above a certain threshold (0.6) are a match
MATCH
Similarity: 0.64
NO MATCH
Similarity: 0.0
Table inventory does not match with any other table
Step 3: Compare similarity of columns between both tables. Similarity determined using:
name_sim x 0.4 + datatype_sim x 0.4 + pk_sim x 0.2
Pairs of columns with similarity above 0.7 are a match
columns of employees
columns of employees_info
data type Sim: 1 (both integer), name Sim: 0.33, pk Sim: 1 (both are)
Sim = 1 x 0.4 + 0.33 x 0.4 + 1 x 0.2 = 0.73
data type Sim: 0.4 (name: varchar, id: integer), name Sim: 0.4, pk Sim: 0 (name is not, id is)
Sim = 0.4 x 0.4 + 0.0 x 0.4 + 0 x 0.2 = 0.16
data type Sim: 0.8 (name: varchar, fullname: char), name Sim: 0.5, pk Sim: 1 (both are not primary keys)
Sim = 0.8 x 0.4 + 0.5 x 0.4 + 1 x 0.2 = 0.72
data types not compatible (boolean and integer)
Sim = 0.4 x 0.4 + 0.125 x 0.4 + 1 x 0.2 = 0.41
Sim = 1 x 0.4 + 1 x 0.4 + 1 x 0.2 = 1

Fig. 7. Proposed schema matching method - example.

- If the attributes $(E_i^L.A_p, E_j^L.A_q)$ have different data types and such types belong to categories where conversion is sometimes possible (e.g., string and numeric conversions are sometimes possible):
  $data\_type\_sim(E_i^L.A_p, E_j^L.A_q) = 0.4$.
- Otherwise:
  $data\_type\_sim(E_i^L.A_p, E_j^L.A_q) = 0.0$ (i.e., attributes do not match)

If the data type similarity is above zero, then the system computes the attribute name similarity ($name\_sim$) between the attributes $E_i^L.A_p$ and $E_j^L.A_q$ considering the Levenshtein distance, as in entity name similarity.

The primary key similarity ($pk\_sim(E_i^L.A_p, E_j^L.A_q)$) between attributes of two entities is equal to 1 if both the attributes are part of the primary key, otherwise, pk_sim is zero. Experimentally, we used the same weight for both name and data type similarity between attributes, but considered that primary key information is slightly less important.

Step 3 of Fig. 7 presents an example of attribute matching between the entities *employees* and *employees_info*. The algorithm calculates the similarity of each attribute of the entity *employees* with the ones of *employees_info*. When $ATSim(E_i^L.A_p, E_j^L.A_q)$ is above a threshold value, it considers that $E_i^L.A_p$ and $E_j^L.A_q$ match.

In the example, similarity evaluation starts with the attributes *emp_id* and *id*. The $ATSim(employees.emp\_id, employees\_info.id)$ equals 0.73, which is above the defined threshold value (0.7). So these attributes match and there is no need to compare *emp_id* with the remaining attributes of *employees_info*. Then, the next attribute of the entity *employees* (i.e., *name*) is compared with the first attribute of *employees_info* (i.e., *id*), which leads to a similarity score ($ATSim(employees.name, employees\_info.id)$) below 0.7. Next, the similarity of *employees.name* and *employees_info.name_full* is evaluated. The system considers them to match. Then, the system evaluates the next attribute of the first entity (i.e., *isactive*). This attribute fails to match *employees_info.id* due to a low similarity score and also fails to match *employees_info.full_name* because its data types (boolean and integer) are not convertible. Finally, *employees.isactive* corresponds to the attribute with the same name in the second entity. In this example, all attributes of considered entities (i.e., *employees* and *employees_info*) have a match.

4.1.2. Schema integration

A binary integration strategy is used to merge all similar attributes among matching entities into a single global attribute [14]. The procedure starts by merging the first two entries in a list of matching entities (local) created in the schema matching step, and a new global entity is created. For the example of the matching entities **employees** and *employees_info* (Fig. 7), the global entity **employees** would be created. The system names the global entity using the name of the first entity in the binary matching, but users may rename the global entity, if necessary.

The pairs of attributes of these two local entities that match are merged into a single attribute in the global entity. The attributes that do not match are copied to the global entity. For instance, the attributes of the global entity *employees* (which results from the match between local entities *employees* and *employees_info*) are *emp_id*, *name* and *is_active*. The system names the attributes in the global entity using the names of the attributes of the first entity in the binary matching. Users can rename the global entity attributes, the same way they can do for global entities.

Then, the global entity is merged with the next entry in the list of matching entities and the procedure repeats until all entries have been processed. This procedure is repeated for all lists of matching entities. In addition, each local entity without a match originates a global entity with the same name and structure of the local one.

The name of a global entity is equal to the name of the first local entity in a list of matching entities. When two local attributes are merged, the name of the global attribute is equal to the name of the local attribute that appears first, and the datatype is identical, if both have the same data type, otherwise the most generic data type is selected. A data type is considered to be more generic than the other if it allows to represent a wider range of values. For instance, double is more generic than integer, and varchar is more generic than integer. For this purpose, we use a data type compatibility matrix.

4.1.3. Schema mapping

Finding the type of data partitioning requires running algorithms on the mappings between entities and their attributes. When a global entity matches only one local entity then we have a simple mapping. Otherwise, if a global entity matches multiple entities in the local schema, then it is necessary to determine which type partitioning exists: vertical or horizontal.

To detect horizontal partitioning, the following criteria is used:

- A global entity matches more than one local schema view entities.
- The number of attributes of the global entity and in the matching local entities must be equal.
- The name and the data type of the matching attributes in the global entity and in the local entities must be the same.

The function *isHorizontal()* in Algorithm 1 checks if the previous conditions are met for a given global entity (E^G). The first condition is tested in line 2. The second and third conditions are tested for each local entity with a match from the global entity (the loop in line 5). If all conditions hold, the horizontal mapping is assigned.

Algorithm 1: Detect horizontal mapping

```
   Input  : A global entity, including entity and attribute mappings (globalEntity)
   Output: True or False
1  Function isHorizontal(globalEntity):
2      if map(globalEntity).size() $\leq 1$ then return false;
3      for localEntity in corr(globalEntity) do
4          if localEntity.getNbAttributes() != globalEntity.getNbAttributes() then return false;
5          for globalAttribute in globalEntity.getAttributes() do
6              if ! localEntity.attributeExists(globalattribute.getName(), globalattribute.getdata type(), globalattribute.isPrimaryKey())
                then return false;
7          end
8          return true;
9      end
10     return false;
```

For vertical mapping, one or several entities must have a foreign key referencing a primary key of another entity, which should be the original entity that was partitioned into other entity. The function *isVertical()* in Algorithm 2 tries to identify foreign key references in order to determine if there is or not vertical partitioning. First, line 2 obtains the list of local entities containing attributes corresponding to the primary key attribute in global entity, and line 3 obtains the local entities containing attributes that are foreign keys and primary keys. If there are no entities, than vertical mapping is not possible. Line 7 gets the attribute that is primary key but not foreign key(s). Finally, lines 8–18 verify that each of the primary key attributes that are foreign keys references the primary key of the original entity. If such a check fails for any of the foreign keys, then the function returns false.

If neither simple, vertical or horizontal mapping is detected, then undefined mapping is assigned. After the automatic generation of the global schema, the user is requested to edit the global schema generated automatically, using a GUI to interact with the system. This interface allows to fix the incorrect mappings or to create user defined mappings.

To create a star schema it is required to create constraints on the global entities, namely primary keys and foreign keys in order to create a valid star schema.

It is possible to create several star schemas over a single global schema. To do so, it is needed to select which global entities are the dimensions and which is the facts entity. In the latter, the attributes that will be used as measures (if any) should also be selected. Note, however, that only addictive or semi-additive measures are supported, meaning that it only makes sense to perform aggregations across all or some of the dimensions in the star schema. The user uses the interface to perform the mapping of entities to a star schema. It is important that foreign key constraints are correctly defined between the facts entity and the dimensions, a step that is performed on the global schema creation via the user interface.

Algorithm 2: Detect vertical mapping

```
Input  : A global entity, including entity and attribute mappings (globalEntity)
Output: True or False
Function isVertical(globalEntity):
    $localentities \leftarrow globalentity.getPrimKeyattribute().getLocalentities()$;
    $fkentities \leftarrow getOnlyPKAndFKentities(localentities)$;
    if fkCols.size() == 0 then return false;
    $primKeyOriginalattribute \leftarrow localentities.getPrimKeyNoFK()$;
    for localentity in fkentities do
        for attribute in localentity.getattributes() do
            if attribute.isPrimaryKey() and attribute.isForeignKey() then
                $referencedattribute \leftarrow attribute.getReferencedattribute()$;
                if referencedattribute.equals(primKeyOriginalattribute) then continue;
                else return false;
            end
        end
    end
    return true
```

4.2. Distributed query definition

EasyBDI transforms each star schema-based user query Q into one or more SQL commands, that are sent for execution by Trino. More than one SQL command may be executed for a single user query to run analytic operators that the distributed query engine does not support. EasyBDI uses nested queries on which the outer query contains the operators requested by the user within the star schema entities. Inner queries retrieve data from the distributed sources associated with the global entities. Hence, inner queries are written considering local schema view entities, thus creating temporary views of the local schema data. The query generation process is as follows:

- *Identify global entities and joins* - User queries are defined over a set of star schema attributes (Q^A). The first step is to use metadata to identify if each star entity in Q^E represents dimensions or facts and what global entity represents each star entity (as defined in (11)). The system retrieves the relations between each referenced entity, and uses this information to create join operations. The result is composed by a set of global entities and attributes, and a set of system-generated join clauses (J).
- *Create a base query with global operators* - Considering the global attributes (Q^A) and entities (Q^E), and the set of system-generated join clauses (J), write a base query defined over global entities.

```
SELECT <attributes_list>
FROM  <tables_list>
WHERE <conditions_list>
```

Listing 1: Base query for query generation.

In Listing 1:
– <attributes_list>= $Q^A = E^G_1.A_{x_1}, E^G_2.A_{x_2}, \dots, E^G_g.A_{x_g}$
– <tables_list>= $Q^E = E^S_1, E^S_2, \dots, E^S_n$
– <conditions_list>= J = *system generated join conditions for tables in* Q^E
For instance, let us consider the global relations *Ratings* and *Movies* defined in Section 3 and whose instances are exemplified in Fig. 3 and Fig. 5, respectively. Consider a user wants to query the average rating for each movie. A sample base query with global operators on these relations is:

```
SELECT Movies.id, Movies.name, Ratings.rating
FROM Movies, Ratings
WHERE Movies.id = Ratings.movieID
```

Listing 2: Sample base query.

Note that the query in Listing 2 does not contain the *Average* function and the *Group By* clause required to compute the average rating per movie. These will be added to the base query in a later step.
- *Map global attributes and local attributes* - For each entity E^S_i in the star schema Q^E, the system extracts the local schema correspondences. Each star schema entity maps to a transformation function to the global schema, and global schema entities map to a transformation function over the local schema:

$$\forall E^S_i \in Q^E; Q^L_i \leftarrow corr(map(E^G)) \tag{15}$$

- *Create local queries for local mappings* - For each mapping in Q_i^L, write a base query B_i^L defined over local entities. The query writing process must consider the mapping type between global and local entities (e.g., *Union* and *join* operations must be used for horizontal and vertical partitioning), as in the following:

  – Simple mapping (defined in Eq. (10)):

```
(SELECT <attributes_list>
 FROM  <Local_table>) as <global_table>
```

Listing 3: Simple mapping query.

In Listing 1:
– *<attributes_list>=* $E^L.A_{x_1}$ *as* $A_1, E^L.A_{x_2}$ *as* $A_2, \dots, E^L.A_{x_q}$ *as* A_q
– *<local_table>=* E^L
– *<global_table>=* E^G
– $A_1, A_2, \dots, A_q$ are attributes from E^G.
For instance, global entity *Ratings* (Fig. 3) would be mapped to a local table *CassandraDS.Ratings* using the following query:

```
(SELECT CassandraDS.Ratings.user as userID,
     CassandraDS.Ratings.date as dateTime,
     CassandraDS.Ratings.movie as movieID,
     CassandraDS.Ratings.score as rating
 FROM CassandraDS.Ratings) as Ratings
```

Listing 4: Simple mapping query example.

  – Vertical mapping (defined in (7)):

```
(SELECT <attributes_list>
 FROM  <Local_tables>
 WHERE <conditions_list>) as <global_table>
```

Listing 5: Vertical mapping query.

In Listing 5:
– *<attributes_list>=* $E_1^L.A_{x_1}$ *as* $A_1, E_1^L.A_{x_2}$ *as* $A_2, \dots,$
$E_2^L.A_{x_1}$ *as* $A_{p+1}, E_2^L.A_{x_2}$ *as* $A_{p+2}, \dots, E_n^L.A_{x_l}$ *as* A_q
– *<local_table>=* $E_1^L, E_2^L, \dots, E_n^L$
– *<conditions_list>=* $E_1^L \bowtie_{cond_1} E_2^L \bowtie_{cond_2} \dots E_r^L$ *= system generated join conditions*
– *<global_table>=* E^G
– $A_1, A_2, \dots, A_q$ are attributes from E^G.
For instance, the global entity *MovieDirectors* 4 would be mapped by the following query:

```
(SELECT PostgresDS.Movies.id as id,
     PostgresDS.Movies.title as title,
     PostgresDS.Movies.released as released,
     PostgresDS.Directors.directorID as directorID,
     PostgresDS.Directors.code as code,
     PostgresDS.Directors.name as name
  FROM PostgresDS.Movies, PostgresDS.Directors
  WHERE PostgresDS.Movies.directorID =
          PostgresDS.Directors.code
 ) as MovieDirectors
```

Listing 6: Vertical mapping query example.

  – Horizontal mapping (as defined in Eq. (8)):

```
(SELECT <attributes_list1>
 FROM  <local_table_1>
 UNION
 SELECT <attributes_list2>
 FROM  <local_table_1>) as <global_table>
```

Listing 7: Base query for horizontal mapping.

In Listing 7:
– *<attributes_list1>=* $E_1^L.A_{x_1}$ *as* $A_1, E_1^L.A_{x_2}$ *as* $A_2, \dots,$
$E_1^L.A_{x_l}$ *as* A_l
– *<local_table_1>=* E_1^L
– *<attributes_list2>=* $E_2^L.A_{x_1}, E_2^L.A_{x_2}, \dots, E_2^L.A_{x_l}$
– *<local_table_2>=* E_2^L
– *<global_table>=* E^G
- $A_1, A_2, \dots, A_q$ are attributes from E^G.
Listing 8 presents the mapping for the global entity *Movies* (Fig. 5), which comprises using horizontal and vertical mapping together.

```
(SELECT MongoDS.MoviesCollection.id as id,
    MongoDS.MoviesCollection.title as title,
    MongoDS.MoviesCollection.released as released,
    MongoDS.MoviesCollection.name as director
 FROM MongoDS.MoviesCollection
 UNION
 (SELECT PostgresDS.movies.id as id,
    PostgresDS.movies.title as title,
    PostgresDS.movies.released as released,
    PostgresDS.Directors.name as director
    FROM PostgresDS.movies, PostgresDS.Directors
    WHERE PostgresDS.movies.directorID =
        PostgresDS.Directors.code
 ) as MovieDirectors
) as Movies
```

Listing 8: Horizontal mapping query example.

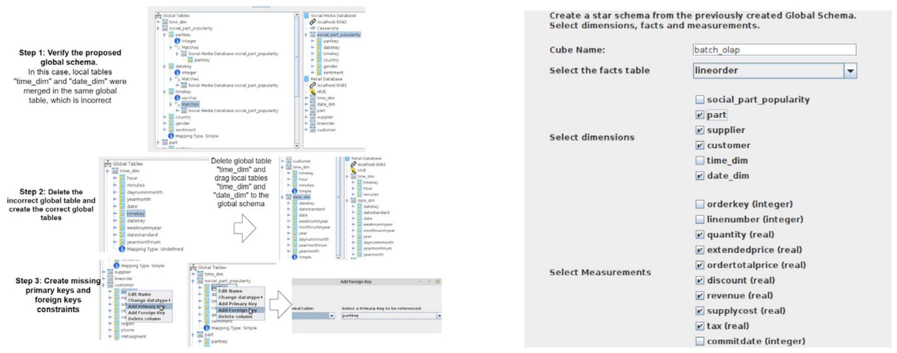
– Transformation mapping:
B_i^L = ***Transformation command***

- *Add inner queries for local mappings* - Add queries defined over local entities as inner queries over the base query.
- *Add user-defined operations* - User-defined operations are included in the global query level. This include complementary operations that are part of the definition of Q in Eq. (12) (i.e., aggregations - AG, grouping functions - GF, individual – F - and aggregate filters - AF, and pivoting - P - and ordering clauses - OB).
  Group functions include functions such as *sum*, *count* and *average*. Aggregation definitions AG leads to placing columns in a *Group By* clause. User-defined filters may be defined over individual tuples (F) or over aggregation results (AF). Filters on tuple values are added to a *Where* clause (which may contain system-defined join conditions - J). Aggregation filters (AF) are defined in a *Having* clause. An *Order By* clause would contain any ordering clauses defined by users (OB).

Let us consider again the example of a user who wants to query the average rating for each movie (base query in Lsting 2. The mapping queries would be added as inner tables in the base query with global operators. Then, the grouping clause and average function would be added for the outer query. Finally, the query to get the average rating per movie is in Listing 9.

```
SELECT Movies.id, Movies.name, avg(Ratings.rating)
FROM (
    (SELECT MongoDS.MoviesCollection.id as id,
        MongoDS.MoviesCollection.title as title,
        MongoDS.MoviesCollection.released as released,
        MongoDS.MoviesCollection.name as director
     FROM MongoDS.MoviesCollection)
     UNION
     (SELECT PostgresDS.movies.id,
        PostgresDS.movies.title,
        PostgresDS.movies.released,
        PostgresDS.Directors.name
     FROM PostgresDS.movies, PostgresDS.Directors
     WHERE PostgresDS.movies.directorID =
     PostgresDS.Directors.code) as MovieDirectors
     ) as Movies,
    (SELECT CassandraDS.Ratings.user as userID,
        CassandraDS.Ratings.date as dateTime,
        CassandraDS.Ratings.movie as movieID,
        CassandraDS.Ratings.score as rating
    FROM  CassandraDS.Ratings)  Ratings
WHERE Movies.id = Ratings.movieID
GROUP BY Movies.id, Movies.name
```

Listing 9: Local query - complete example.


Step 1: Verify the proposed global schema. In this case, local tables "time_dim" and "date_dim" were merged in the same global table, which is incorrect
Step 2: Delete the incorrect global table and create the correct global tables
Delete global table "time_dim" and drag local tables "time_dim" and "date_dim" to the global schema
Step 3: Create missing primary keys and foreign keys constraints
Edit Name
Change datatype
Add Primary Key
Add Foreign Key
Delete column

Create a star schema from the previously created Global Schema.
Select dimensions, facts and measurements.

Cube Name: batch_olap

Select the facts table: lineorder

Select dimensions:
- [ ] social_part_popularity
- [✓] part
- [✓] supplier
- [✓] customer
- [ ] time_dim
- [✓] date_dim

Select Measurements:
- [ ] orderkey (integer)
- [ ] linenumber (integer)
- [✓] quantity (real)
- [✓] extendedprice (real)
- [✓] ordertotalprice (real)
- [✓] discount (real)
- [✓] revenue (real)
- [✓] supplycost (real)
- [✓] tax (real)
- [ ] commitdate (integer)

(a) SSB+ - Global Schema configuration using EasyBDI. (b) Batch OLAP star schema.

Fig. 8. Schema mapping and star schema definition.

5. Application examples

This section describes usage scenarios and demonstrates how to organize and use data in a star schema using EasyBDI. It presents the query generation process, from the user-defined star schema in EasyBDI to an SQL command executed by Trino to query the local schemas.

The configuration used for the tests include Trino, version 330, configured to behave as both a coordinator and a worker on a single node configuration. The server runs Linux Mint 19.1 Cinnamon and has 8 GB of RAM, an Intel Core i7-8550U at 1.80 GHz with 4 cores processor, and a solid-state drive. Our source-code is currently available at EasyBDI [30].

5.1. Case study 1: Batch and streaming OLAP

This case study uses the SSB+ benchmark [31], which includes a dataset and a set of queries to evaluate the performance of decision support systems. SSB+ is an extension of the SSB benchmark proposed by O'Neil et al. [32], which in turn is a star schema adaptation of the TPC-H benchmark. The SSB+ data model contains star schemas for batch and streaming OLAP. The batch OLAP star schema contains facts and dimension tables used in the context of retail data. The schema for streaming OLAP contains social media data, representing the retail store's popularity on social media. In this section, we use the batch OLAP star schema of SSB+.

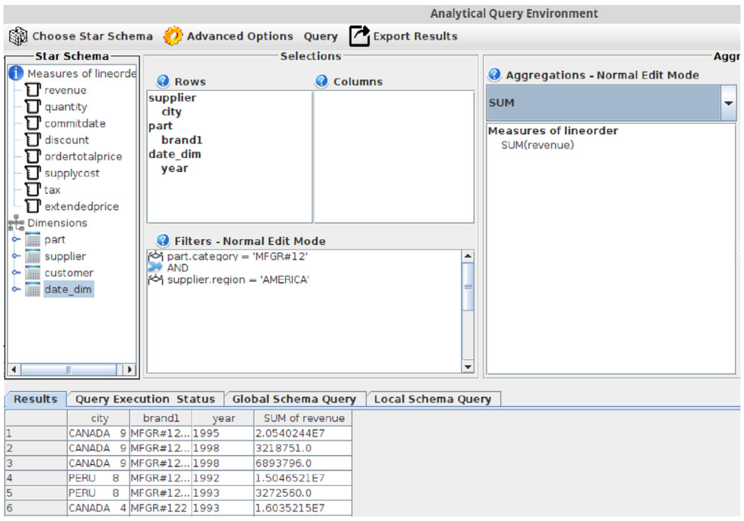
We instantiated the SSB+ data model using the code available in SSB+ Github repository [33] and used Cassandra and Hive 3.1.2 to store the database. Cassandra stores the social media tables, while Hive stores retail data in a database called *minhodb*.

We used EasyBDI to access both sources and propose a global schema. Fig. 8(a) presents a partial screenshot of the local schema views (on the left) and the global schema (on the right) proposed by EasyBDI. In the left panel, the *Cassandra* data source is labeled as the *Social Media Database*, and the *Hive* data source is labeled as the *Retail Database*. In this step, the global schema generated by EasyBDI was almost accordingly to what was expected, having correctly identified nearly all mappings. Two local tables ("date_dim" and "time_dim") were incorrectly merged in the global schema, therefore it was necessary to delete the global table "time_dim" and create two global tables, "date_dim" and "time_dim", with a one-on-one mapping to the corresponding local tables. This mistake happened because both tables have similar column types and names, thus schema matching assumed there was a matching. Foreign keys and Primary keys also needed to be added to the global schema. The whole process of global schema correction via EasyBDI is depicted in Fig. 8(a). Afterwards, with the global schema correctly configured, it was necessary to create a star schema, in which the user has to choose a facts table, dimensions, and measures, as exemplified in Fig. 8(b).

In this work, we use queries 2 and 4 of the benchmark. Query 2 computes the revenue for a certain month, for products that were sold in certain quantities and with specified discounts. Query 4 computes the revenue for the year and brand, for sales of products of certain categories and of suppliers in a specified region. The SQL listing of these queries is available in [33].

Fig. 9 presents the specification of query 4 in EasyBDI's GUI. This query contains several operations, namely joins, filters, aggregations, ordering, and grouping operations. EasyBDI's SQL generation algorithm detects an aggregate function and places adequate columns in the *group by* clause. EasyBDI automatically specifies the join clauses between the facts and the dimensions tables by checking metadata for the relationships between global tables. Filters and ordering clauses are also specified.

Fig. 10 illustrates the query generation process. Initially, EasyBDI generates the base SQL query on global entities. Listing 10 shows the global schema generated query by EasyBDI using the configuration made in EasyBDI's query GUI depicted in Fig. 9. It corresponds to *a)* in the flow depicted in Fig. 9. This global schema query must then be further developed to query the local schema, as it has no notion of local schema entities at this stage. To this end, the mapping between each query's table and local tables is fetched from the Metadata Storage. After obtaining the information, a local schema query is created for each different local schema



Rows:
supplier
  city
part
  brand1
date_dim
  year

SUM

Measures of lineorder
  SUM(revenue)

part.category = 'MFGR#12'
AND
supplier.region = 'AMERICA'

	city	brand1	year	SUM of revenue
1	CANADA 9	MFGR#12...	1995	2.0540244E7
2	CANADA 9	MFGR#12...	1998	3218751.0
3	CANADA 9	MFGR#12...	1998	6893796.0
4	PERU 8	MFGR#12...	1992	1.5046521E7
5	PERU 8	MFGR#12...	1993	3272560.0
6	CANADA 4	MFGR#122	1993	1.6035215E7

Fig. 9. SSB+ Query 4 in EasyBDI.

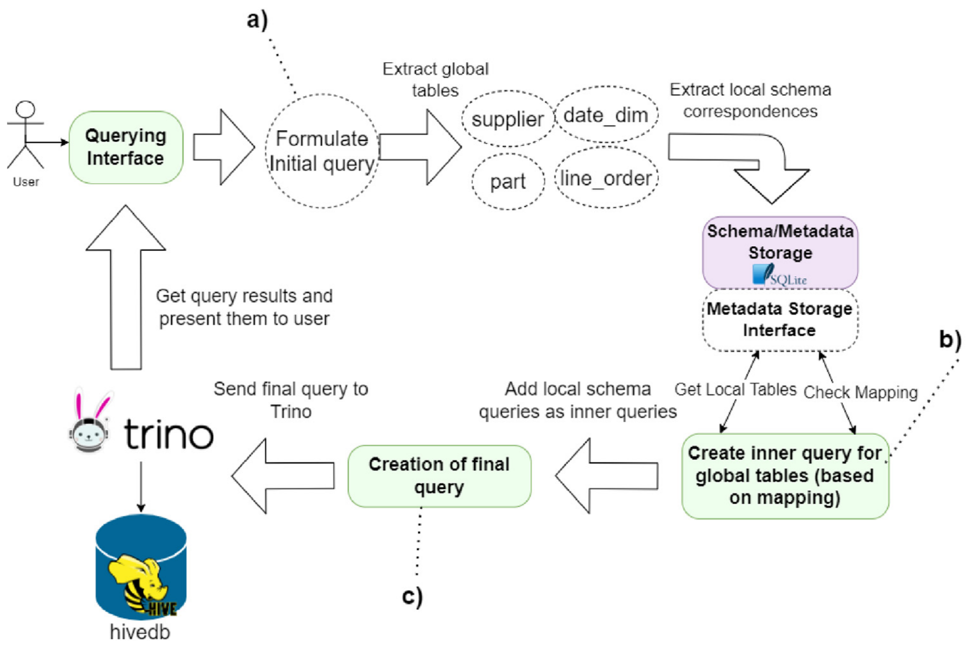
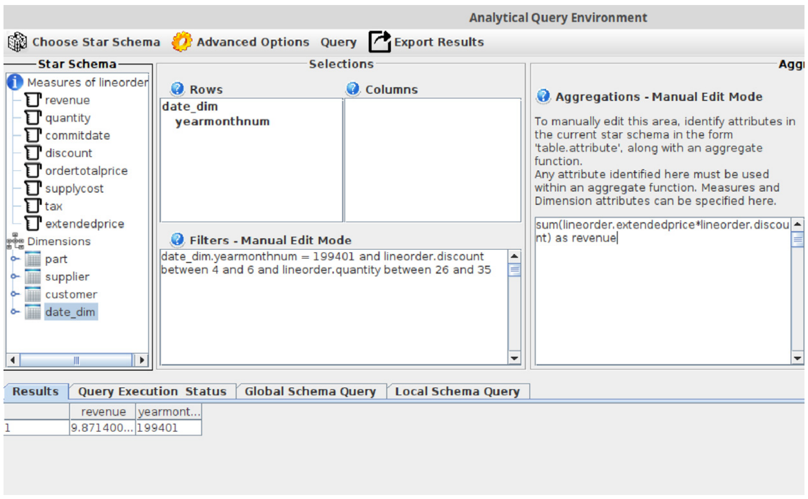

a)
User
Querying Interface
Formulate Initial query
Extract global tables
supplier
date_dim
part
line_order
Extract local schema correspondences
Schema/Metadata Storage
SQLite
Metadata Storage Interface
b)
Get Local Tables
Check Mapping
Create inner query for global tables (based on mapping)
Add local schema queries as inner queries
Creation of final query
c)
Send final query to Trino
trino
hivedb
HIVE
Get query results and present them to user

Fig. 10. Example of the query generation process in EasyBDI.

table, as seen in listing 11, and it is generated in *(b)* in the flow of Fig. 9. Finally, the global query in listing 10 and the local queries in listing 11 are combined into one query that contains local schema entities and the corresponding mapping to global schema entities, depicted in listing 12, which will also be executed by the distributed query execution engine to query the datasources. This final query corresponds to *(c)* in Fig. 9.

```
SELECT supplier.city, part.brand1, date_dim.year, sum(revenue) as "sum of revenue"
FROM supplier.part, date_dim, lineorder
WHERE (supplier.suppkey = lineorder.suppkey AND part.partkey = lineorder.partkey AND date_dim.datekey = lineorder.
  orderdate) AND (part.category = 'MFGR#12' AND supplier.region = 'AMERICA')
GROUP BY (supplier.city, part.brand1, date_dim.year)
ORDER BY (part.brand1, date_dim.year)
```

Listing 10: EasyBDI generated global schema query resulting from user input.



Rows: date_dim yearmonthnum

To manually edit this area, identify attributes in the current star schema in the form 'table.attribute', along with an aggregate function.
Any attribute identified here must be used within an aggregate function. Measures and Dimension attributes can be specified here.

```
sum(lineorder.extendedprice*lineorder.discount) as revenue
```

```
date_dim.yearmonthnum = 199401 and lineorder.discount
between 4 and 6 and lineorder.quantity between 26 and 35
```

	revenue	yearmont...
1	9.871400...	199401

Fig. 11. SSB+ Query 2 in EasyBDI.

```
SELECT hive.minhodb.part.brand1, hive.minhodb.part.partkey, hive.minhodb.part.category
FROM hive.minhodb.part

SELECT hive.minhodb.date_dim.year, hive.minhodb.date_dim.datekey
FROM hive.minhodb.date_dim

SELECT hive.minhodb.supplier.city, hive.minhodb.supplier.suppkey, hive.minhodb.supplier.region
FROM hive.minhodb.supplier

SELECT hive.minhodb.lineorder.orderdate, hive.minhodb.lineorder.revenue, hive.minhodb.lineorder.partkey, hive.minhodb.
  lineorder.suppkey
FROM hive.minhodb.lineorder
```

Listing 11: EasyBDI queries for each local schema.

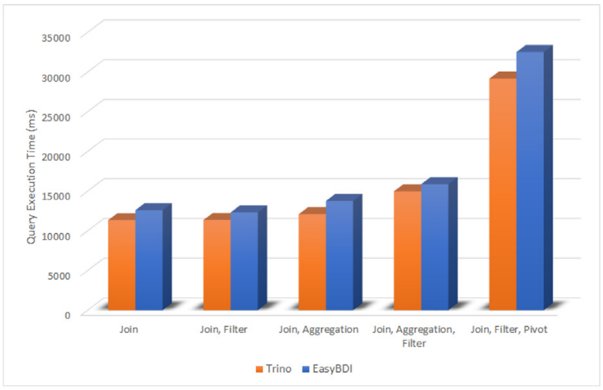
```
SELECT supplier.city, part.brand1, date_dim, year, SUM(revenue) AS "sum of revenue"
FROM
    (SELECT hive.minhodb.supplier.city, hive.minhodb.supplier.suppkey, hive.minhodb.supplier.region
    FROM hive.minhodb.supplier) AS supplier,
    (SELECT hive.minhodb.part.brand1, hive.minhodb.part.partkey,        hive.minhodb.part.category
    FROM hive.minhodb.part) AS part,
    (SELECT hive.minhodb.lineorder.orderdate, hive.minhodb.lineorder.revenue, hive.minhodb.lineorder.partkey, hive.
  minhodb.lineorder.suppkey
    FROM hive.minhodb.lineorder) AS lineorder,
WHERE (supplier.suppkey = lineorder.suppkey
AND part.partkey = lineorder.suppkey
AND date_dim.datekey = lineorder.orderdate)
AND (part.category = 'MFGR#12' AND supplier.region = 'AMERICA')
GROUPBY (supplier.city, part.brand1, date_dim.year)
ORDER BY (part.brand1, date_dim, year);
```

Listing 12: Example of the SQL executed by the query engine, capable of querying the local schema but also correlating to the global schema.

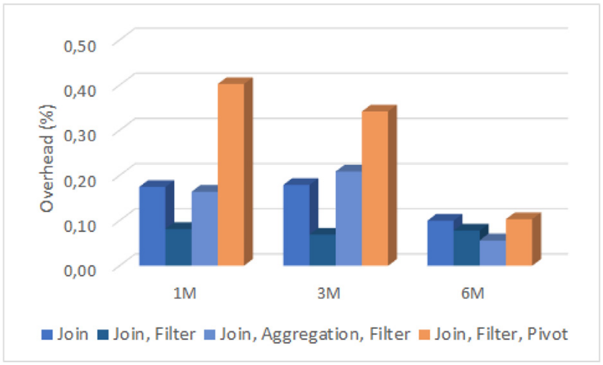
Query number 2 is composed of an aggregation operation that multiplies two different attributes, a join operation between the facts and dimension tables, and a filter. Fig. 11 presents the query specification through EasyBDI's interface. The generated query by EasyBDI is fairly similar to the original query except that inner queries are added and that joins are made using *where* conditions.

To verify the overhead EasyBDI introduces on distributed query execution time, we compared the execution time of queries submitted through the EasyBDI GUI interface with the ones of SQL commands executed by Trino. We executed several queries with increased complexity over SSB+ tables while also varying the number of records in the facts table.

Fig. 12(a) presents that query execution times in EasyBDI are close to that of Trino. The highest difference is on queries with PIVOT operations. Indeed, Trino does not support the PIVOT SQL operator. To perform the corresponding operation in Trino, it was necessary to manually specify in the SQL code each distinct value of the column that is being pivoted. EasyBDI, on the other hand, obtains such values through a database query. Hence, Trino requires the specification, in the SQL command, of data location and transformations from original models to the high-level star schema objects based in global entities. EasyBDI hides such complexity with a little overhead. Also, as database size increases, EasyBDI's execution time overhead becomes less relevant (Fig. 12(b)).

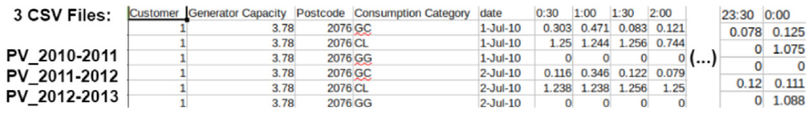


(a) Query execution time - 6 Million records.



(b) EasyBDI query execution time overhead.

Fig. 12. EasyBDI and Trino query execution time - multiple query types and table sizes.


3 CSV Files:

PV_2010-2011
PV_2011-2012
PV_2012-2013

Customer	Generator Capacity	Postcode	Consumption Category	date	0:30	1:00	1:30	2:00		23:30	0:00
1	3.78	2076	GC	1-Jul-10	0.303	0.471	0.083	0.121		0.078	0.125
1	3.78	2076	CL	1-Jul-10	1.25	1.244	1.256	0.744		0	1.075
1	3.78	2076	GG	1-Jul-10	0	0	0	0	(...)	0	0
1	3.78	2076	GC	2-Jul-10	0.116	0.346	0.122	0.079		0.12	0.111
1	3.78	2076	CL	2-Jul-10	1.238	1.238	1.256	1.25		0	1.088
1	3.78	2076	GG	2-Jul-10	0	0	0	0			

Fig. 13. PV data source.

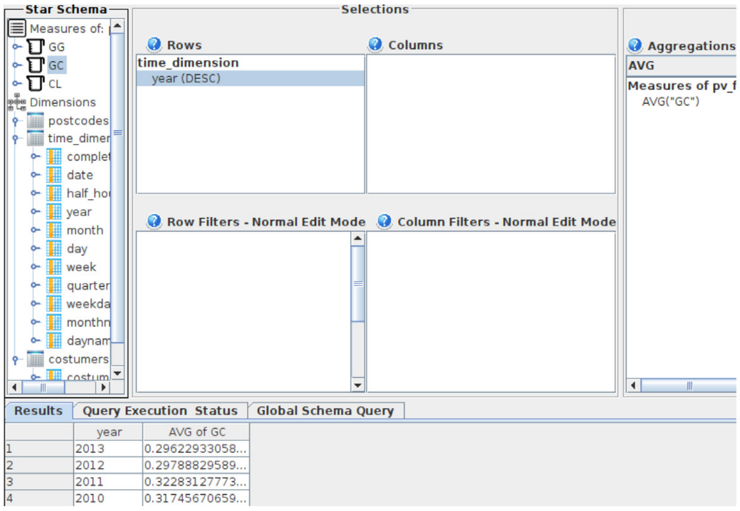
5.2. Case study 2: Ausgrid dataset

The second case study presents how to execute OLAP queries on flat files with real-world data on photovoltaic panel electricity production and consumption in Sydney, Australia. The photovoltaic data (PV) is available in [34] and contains data of 300 randomly selected customers from 1 July 2010 to 30 June 2013. Fig. 13 presents a sample of the comma-separated files (CSV) recording the gross generation (GG), general consumption (GC), and controllable load (CL), for each customer and every half-hour.

The other data sources are a MySQL database to store customer data (e.g., the *id* and the customers' generator capacity) and a *time* table, and a PostgreSQL database to store location data (e.g., postal codes).

EasyBDI accesses each data source (CSV, MySQL and PostgreSQL), retrieves their logical data organization and creates an abstraction through Local Schema Views, and proposes a Global Schema for the distributed database. The three CSV files are mapped to a single logical entity with horizontally partitioned data (based on time). But in the CSV files, each client's PV panel generation data is set as a column per each half hour. In the facts table, such values would be represented in rows. Hence, the mapping of the half-hour columns into rows is like an unpivoting operation. The different types of electricity generation data categories (*GG*, *GC*, and *CL*) are set as rows in the CSV files. These fields are of interest for group operations, and these 3 categories need to be pivoted in the global schema mapping. Hence, in the global schema definition, there is a transformation mapping that does the unpivoting of time columns and the pivoting of data categories columns.

Once the global schema is tuned and the star schema is specified with the identification of facts and dimensions, it is possible to create OLAP queries to access data of the three CSV files and the two DBMS on the fly. Fig. 14 presents the specification of two sample queries using EasyBDI. In Fig. 14(a), the query computes the average consumption per year. The query in Fig. 14(b) computes the total energy production per client between 2010 and 2012, presenting each year as column (a PIVOT operation on



Rows: time_dimension — year (DESC)

Columns:

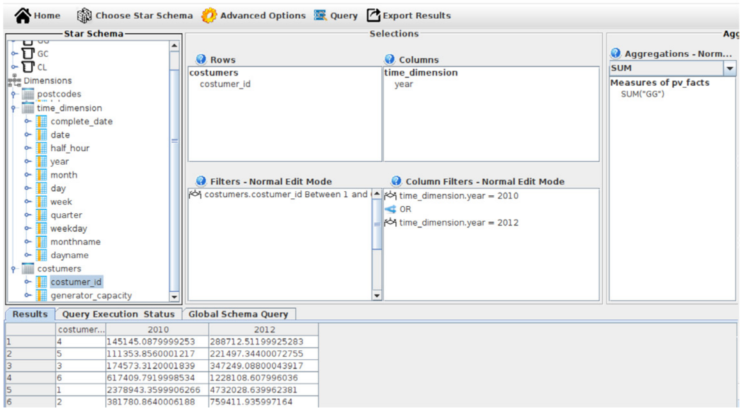
Row Filters - Normal Edit Mode

Column Filters - Normal Edit Mode

Aggregations: AVG — Measures of pv_f — AVG("GC")

	year	AVG of GC
1	2013	0.29622933058...
2	2012	0.29788829589...
3	2011	0.32283127773...
4	2010	0.31745670659...

(a) Average consumption per year.



Rows: costumers — costumer_id

Columns: time_dimension — year

Filters - Normal Edit Mode: costumers.costumer_id Between 1 and

Column Filters - Normal Edit Mode: time_dimension.year = 2010 OR time_dimension.year = 2012

Aggregations - Norm...: SUM — Measures of pv_facts — SUM("GG")

	costumer...	2010	2012
1	4	145145.0879999253	288712.51199925283
2	5	111353.8560001217	221497.34400072755
3	3	174573.3120001839	347249.08800043917
4	6	617409.7919998534	1228108.607996036
5	1	2378943.3599906266	4732028.639962381
6	2	381780.8640006188	759411.935997164

(b) Energy production per clients in 2010 and 2012.

Fig. 14. Sample queries on energy consumption and production.

CSV data specified by the user at query level). EasyBDI displays the query results to users accordingly the high-level abstractions of the star schema. Query execution uses fresh data obtained directly from the data sources.

The above examples show that EasyBDI provides abstractions which subject experts may use to execute OLAP operations over distributed data. EasyBDI enables query formulation over high-level objects, hides the complexity of SQL commands, and provides data location and partitioning transparency.

6. Conclusion

A wide range of distributed sources and devices is currently generating large amounts of data. In current scenarios, e.g., IoT, the decision-making processes depend largely on analyzing fresh data. This requirement is incompatible with traditional data warehouses requiring time-consuming ETL processes. Therefore, novel methods and tools are needed to conceptually integrate distributed data sources and enable the execution of advanced analytical near real-time queries over fresh data.

EasyBDI is a platform that implements methods and tools to extract the schema representation of several data sources, applies schema matching, integration, and mapping techniques, and automatically proposes a global schema of the distributed sources. Users may fine-tune the proposed global schema and use it to define star schemas representing the data in distributed sources. Finally, users may submit queries over the high-level star schema objects. The system provides fragmentation and location transparency. Thus, it executes queries over distributed sources using global concepts and the users do not need to know about the data location or fragmentation during query formulation. Also, it does not instantiate intermediary data like in a warehouse, thus providing direct access to fresh data.

In this paper, we present the concepts and the methods used by EasyBDI to execute schema matching, integration, and mapping, and how to transform high-level user queries (based on star schema and global entities) into SQL statements executed by Trino. We define how the mappings between the star, global and local schemas generate valid queries that retrieve data from local schemas and present results accordingly with the user-defined star schema entities, all while abstracting the local schema and its heterogeneity. As future work, we plan to add methods of provenance information management to know where the data comes from and how the results of analytical queries on distributed and heterogeneous data sources were produced.

CRediT authorship contribution statement

Bruno Silva: Conceptualization, Methodology, Software, Validation, Investigation, Data curation, Writing – original draft, Writing – review & editing, Visualization. **José Moreira:** Conceptualization, Methodology, Investigation, Writing – original draft, Writing – review & editing, Visualization, Supervision, Project administration. **Rogério Luís de C. Costa:** Conceptualization, Methodology, Investigation, Writing – original draft, Writing – review & editing, Visualization, Supervision, Project administration.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work is funded by national funds through FCT – Fundação para a Ciência e a Tecnologia, Portugal, I.P., under the Scientific Employment Stimulus - Institutional Call - CEECINST/00051/2018, and in the context of the projects UIDB/04524/2020 and MIT-EXPL/ACC/0057/2021.

References

[1] R.L. de C. Costa, J. Moreira, P. Pintor, V. dos Santos, S. Lifschitz, A survey on data-driven performance tuning for big data analytics platforms, Big Data Res. 25 (2021) 100206, http://dx.doi.org/10.1016/j.bdr.2021.100206, URL: https://www.sciencedirect.com/science/article/pii/S221457962100023X.

[2] R.A. Ariyaluran Habeeb, F. Nasaruddin, A. Gani, I.A. Targio Hashem, E. Ahmed, M. Imran, Real-time big data processing for anomaly detection: A survey, Int. J. Inf. Manage. 45 (2019) 289–307, http://dx.doi.org/10.1016/j.ijinfomgt.2018.08.006, URL: https://www.sciencedirect.com/science/article/pii/S0268401218301658.

[3] T. Küfner, S. Schönig, R. Jasinski, A. Ermer, Vertical data continuity with lean edge analytics for industry 4.0 production, Comput. Ind. 125 (2021) 103389, http://dx.doi.org/10.1016/j.compind.2020.103389, URL: https://www.sciencedirect.com/science/article/pii/S0166361520306230.

[4] M. Babar, F. Arif, Smart urban planning using big data analytics to contend with the interoperability in Internet of Things, Future Gener. Comput. Syst. 77 (2017) 65–76.

[5] Y. Hajjaji, W. Boulila, I.R. Farah, I. Romdhani, A. Hussain, Big data and IoT-based applications in smart environments: A systematic review, Comp. Sci. Rev. 39 (2021) 100318, http://dx.doi.org/10.1016/j.cosrev.2020.100318, URL: https://www.sciencedirect.com/science/article/pii/S1574013720304184.

[6] R. Tan, R. Chirkova, V. Gadepally, T.G. Mattson, Enabling query processing across heterogeneous data models: A survey, in: 2017 IEEE International Conference on Big Data, BigData 2017, 2017, pp. 3211–3220, http://dx.doi.org/10.1109/BigData.2017.8258302.

[7] B. Kolev, P. Valduriez, C. Bondiombouy, R. Jiménez-Peris, R. Pau, J. Pereira, CloudMdsQL: querying heterogeneous cloud data stores with a common language, Distrib. Parallel Databases 34 (4) (2016) 463–503, http://dx.doi.org/10.1007/s10619-015-7185-y.

[8] R. Sethi, M. Traverso, D. Sundstrom, D. Phillips, W. Xie, Y. Sun, N. Yegitbasi, H. Jin, E. Hwang, N. Shingte, C. Berner, Presto: SQL on everything, in: Proceedings - International Conference on Data Engineering, Vol. 2019-April, 2019, pp. 1802–1813, http://dx.doi.org/10.1109/ICDE.2019.00196.

[9] Apache Software Foundation, Apache drill - architecture, 2019, URL: https://drill.apache.org/architecture/. (Last Accessed 07 December 2019).

[10] D. Abadi, A. Ailamaki, D. Andersen, P. Bailis, M. Balazinska, P. Bernstein, P. Boncz, S. Chaudhuri, A. Cheung, A. Doan, et al., The seattle report on database research, ACM SIGMOD Rec. 48 (4) (2020) 44–53.

[11] B. Silva, J. Moreira, R.L.C. Costa, EasyBDI: Near real-time data analytics over heterogeneous data sources, in: Proceedings of the 24th International Conference on Extending Database Technology, EDBT 2021, 2021, pp. 702–705, http://dx.doi.org/10.5441/002/edbt.2021.88.

[12] B. Garg, K. Kaur, Integration of heterogeneous databases, in: Conference Proceeding - 2015 International Conference on Advances in Computer Engineering and Applications, ICACEA 2015, 2015, pp. 1033–1038, http://dx.doi.org/10.1109/ICACEA.2015.7164859.

[13] E. Rahm, P. Bernstein, A survey of approaches to automatic schema matching, VLDB J. 10 (2001) 334–350, http://dx.doi.org/10.1007/s007780100057.

[14] M.T. Özsu, P. Valduriez, Principles of Distributed Database Systems, fourth ed., 2020, http://dx.doi.org/10.1007/978-3-030-26253-2.

[15] A. Ali, A. Nordin, M. Alzeber, A. Zaid, A survey of schema matching research using database schemas and instances, Int. J. Adv. Comput. Sci. Appl. 8 (10) (2017) http://dx.doi.org/10.14569/ijacsa.2017.081014.

[16] A. Adamou, M. d'Aquin, Relaxing global-as-view in mediated data integration from linked data, in: Proceedings of the International Workshop on Semantic Big Data, SBD '20, New York, NY, USA, 2020, pp. 1–6, http://dx.doi.org/10.1145/3391274.3393635.

[17] Y. Katsis, Y. Papakonstantinou, View-based data integration, in: Encyclopedia of Database Systems, 2018, pp. 4452–4461, http://dx.doi.org/10.1007/978-1-4614-8265-9_1072.

[18] A. Bonifati, G. Mecca, P. Papotti, Y. Velegrakis, Discovery and correctness of schema mapping transformations, in: Schema Matching and Mapping, 2011, pp. 111–147, http://dx.doi.org/10.1007/978-3-642-16518-4_5.

[19] B. ten Cate, P.G. Kolaitis, W.-C. Tan, Schema Mappings and Data Examples, in: Proceedings of the 16th International Conference on Extending Database Technology, EDBT '13, New York, NY, USA, 2013, pp. 777–780, http://dx.doi.org/10.1145/2452376.2452479.

[20] Z. Jin, C. Baik, M. Cafarella, H.V. Jagadish, Beaver: Towards a declarative schema mapping, in: Proceedings of the Workshop on Human-in-the-Loop Data Analytics, HILDA 2018, New York, New York, USA, 2018, pp. 1–4, http://dx.doi.org/10.1145/3209900.3209902.

[21] R. Fagin, L.M. Haas, M. Hernández, R.J. Miller, L. Popa, Y. Velegrakis, Clio: Schema mapping creation and data exchange, in: Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), Vol. 5600 LNCS, 2009, pp. 198–236, http://dx.doi.org/10.1007/978-3-642-02463-4_12.

[22] H. Ramadhan, F.I. Indikawati, J. Kwon, B. Koo, MusQ: A multi-store query system for IoT data using a datalog-like language, IEEE Access 8 (2020) 58032–58056, http://dx.doi.org/10.1109/ACCESS.2020.2982472.

[23] J. Duggan, A.J. Elmore, M. Stonebraker, M. Balazinska, B. Howe, J. Kepner, S. Madden, D. Maier, T. Mattson, S. Zdonik, The BigDAWG polystore system, SIGMOD Rec. 44 (2) (2015) 11–16, http://dx.doi.org/10.1145/2814710.2814713.

[24] M. Rodrigues, M.Y. Santos, J. Bernardino, Big data processing tools: An experimental performance evaluation, in: Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery, Vol. 9, No. 2, 2019, http://dx.doi.org/10.1002/widm.1297.
[25] R.J. Santos, J. Bernardino, Real-time data warehouse loading methodology, in: ACM International Conference Proceeding Series, Vol. 299, 2008, pp. 49–58, http://dx.doi.org/10.1145/1451940.1451949.
[26] A. Wibowo, Problems and available solutions on the stage of extract, transform, and loading in near real-time data warehousing (a literature study), in: 2015 International Seminar on Intelligent Technology and its Applications, ISITIA 2015 - Proceeding, 2015, pp. 345–349, http://dx.doi.org/10.1109/ISITIA.2015.7220004.
[27] J. Zuters, Near real-time data warehousing with multi-stage trickle and flip, in: J. Grabis, M. Kirikova (Eds.), Perspectives in Business Informatics Research, Springer Berlin Heidelberg, Berlin, Heidelberg, 2011, pp. 73–82.
[28] A. Cuzzocrea, N. Ferreira, P. Furtado, A rewrite/merge approach for supporting real-time data warehousing via lightweight data integration, J. Supercomput. 76 (5) (2020) 3898–3922, http://dx.doi.org/10.1007/s11227-018-2707-9.
[29] S. Bimonte, E. Gallinucci, P. Marcel, S. Rizzi, Data variety, come as you are in multi-model data warehouses, Inf. Syst. 104 (2022) 101734, http://dx.doi.org/10.1016/j.is.2021.101734.
[30] B. Silva, EasyBDI github repository, 2021, URL: https://github.com/bsilva3/EasyBDI. (Last Accessed 04 October 2021).
[31] C. Costa, M.Y. Santos, Evaluating several design patterns and trends in big data warehousing systems, in: Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), Vol. 10816 LNCS, 2018, pp. 459–473, http://dx.doi.org/10.1007/978-3-319-91563-0_28.
[32] P.E. O'Neil, E.J. O'Neil, X. Chen, The star schema benchmark (SSB), 2009, URL: https://www.cs.umb.edu/~poneil/StarSchemaB.PDF. (Last Accessed 11 October 2021).
[33] C. Costa, Big data benchmarks - repository, 2018, URL: https://github.com/epilif1017a/bigdatabenchmarks. (Last Accessed 26 February 2021).
[34] Ausgrid, Solar home electricity data, 2013, URL: https://www.ausgrid.com.au/Industry/Our-Research/Data-to-share/Solar-home-electricity-data. (Last accessed 04 October 2021).

Bruno Silva received a Master's Degree in Computer Science from the University of Aveiro in 2021. In the same university, he participated in a few research projects in the areas of Spatio-Temporal Data and Data Integration and published two articles in well-established international conferences, such as EDBT. His research interests include Big Data, Data Integration, and Machine Learning.

José Moreira received his Ph.D. in Computer Science and Networks from École Nationale Supérieure des Télécommunications de Paris (France), currently known as Télécom ParisTech, and Faculdade de Engenharia da Universidade do Porto (Portugal), in 2001. He is Assistant Professor at the Department of Electronics, Telecommunications, and Informatics of Universidade de Aveiro and researcher at IEETA, a non-profit R&D institute affiliated to the same university. He also served as guest professor of the Information and Communications Technologies Institute of Carnegie Mellon University and OTH Regensburg (Germany). His main teaching and research topics are on data and knowledge engineering, and he is particularly interested in modeling, representing, and analyzing spatiotemporal data in databases. He is member of the Program Committee of the IEEE International Conference on eScience since 2014.

Rogério Luís de Carvalho Costa received a Ph.D. in Computer Engineering from the University of Coimbra (UC), Portugal, in 2011, and an M.Sc. in Informatics from the Pontifícia Universidade Católica do Rio de Janeiro (PUC-Rio), Brazil, in 2002. He has over 15 years of teaching experience. He participated in research projects in Brazil and Portugal and published papers at international conferences and leading journals. Rogério currently is a researcher at the Polytechnic of Leiria, Portugal. His research interests include big data, machine learning, data integration, and data quality.