Universidade de Aveiro
2021

Bruno José
Pires Silva

EasyBDI: Integração Automática de Big Data e Consultas Analíticas de Alto Nível

EasyBDI: Automatic Big Data Integration and High-Level Analytic Queries

Universidade de Aveiro
2021

Bruno José
Pires Silva

EasyBDI: Integração Automática de Big Data e Consultas Analíticas de Alto Nível

EasyBDI: Automatic Big Data Integration and High-Level Analytic Queries

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requesitos necessários à obtenção do grau de Mestre em Engenharia Informàtica, realizada sob a orientação científica de José Manuel Matos Moreira, Professor auxiliar do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro, e Rogério Luís de Carvalho Costa, Investigador no Centro de Investigação em Informática e Comunicações (CIIC) do Instituto Politécnico de Leiria.

Este trabalho foi parcialmente financiado pelo *Fundo Europeu de Desenvolvimento Regional (FEDER), Programa Operacional Competitividade e Internacionalização* no contexto do projeto PRODUTECH SIF – Soluções para a Indústria de Futuro – POCI-01-0247-FEDER-024541.

o júri / the jury

presidente / president	**Prof. Doutor Carlos Manuel Azevedo Costa** Professor Associado do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro
vogais / examiners committee	**Prof. Doutor José Manuel Matos Moreira** Professor Auxiliar do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro
	Prof. Doutor Pedro Nuno San-Bento Furtado Professor Auxiliar na Faculdade de Ciências e Tecnologia da Universidade de Coimbra

agradecimentos / acknowledgements

Aproveito para deixar um enorme agradecimento aos meus dois orientadores, José Moreira e Rogério Costa, que me apoiaram, desafiaram, e me motivaram a fazer mais e melhor. Sem a sua tutoria, este trabalho não seria possível. Aproveito também para agradecer aos meus amigos, que me alegraram ao longo deste percurso, e à minha família, em especial aos meus pais e avós, que me motivaram e apoiaram.

Palavras-Chave

Big data, integração de dados, bases de dados distribuídas, bases de dados heterógeneas, integração de dados lógica, OLAP, data warehousing, dados em tempo quase-real.

Resumo

O aparecimento de novas áreas, como a Internet das Coisas, que requerem o acesso aos dados mais recentes para ambientes de tomada de decisão, criou constrangimentos na execução de consultas analíticas usando as arquiteturas tradicionais de data warehouses.

Adicionalmente, o aumento de dados semi-estruturados e não estruturados levou a que outras bases de dados fossem criadas para lidar com esse tipo de dados, nomeadamente bases NoSQL. Isto levou a que a informação seja armazenada em sistemas com características distintas e especializados em diferentes casos de uso, criando dificuldades no acesso aos dados que estão agora espalhados por vários sistemas com modelos e características distintas.

Neste trabalho, propõe-se um sistema capaz de efetuar consultas analíticas em tempo real sobre fontes de dados distribuídas e heterogéneas: o EasyBDI. O sistema é capaz de integrar dados lógicamente, sem materializar os dados, criando uma vista geral dos dados que oferece uma abstração sobre a distribuição e heterogeneidade das fontes de dados. As consultas são executadas interativamente nas fontes de dados, o que significa que os dados mais recentes serão sempre usados nas consultas. Este sistema apresenta uma interface de utilizador que ajuda na configuração de fontes de dados, e propõe automaticamente um esquema global que apresenta a vista genérica e simplificada dos dados, podendo ser modificado pelo utilizador. O sistema permite a criação de múltiplos esquema em estrela a partir do esquema global. Por fim, a realização de consultas analíticas é feita também através de uma interface de utilizador que recorre ao drag-and-drop de elementos.

O EasyBDI é capaz de resolver problemas recentes, utilizando também soluções recentes, escondendo os detalhes de diversas fontes de dados, ao mesmo tempo que permite que utilizadores com menos conhecimentos em bases de dados possam também realizar consultas analíticas em tempo-real sobre fontes de dados distribuídas e heterogéneas.

Keywords

Big data, data integration, distributed databases, heterogenous databases, logical data integration, OLAP, data warehousing, near real-time data.

Abstract

The emergence of new areas, such as the internet of things, which require access to the latest data for data analytics and decision-making environments, created constraints for the execution of analytical queries on traditional data warehouse architectures.

In addition, the increase of semi-structure and unstructured data led to the creation of new databases to deal with these types of data, namely, NoSQL databases. This led to the information being stored in several different systems, each with more suitable characteristics for different use cases, which created difficulties in accessing data that are now spread across various systems with different models and characteristics.

In this work, a system capable of performing analytical queries in real time on distributed and heterogeneous data sources is proposed: EasyBDI. The system is capable of integrating data logically, without materializing data, creating an overview of the data, thus offering an abstraction over the distribution and heterogeneity of data sources. Queries are executed interactively on data sources, which means that the most recent data will always be used in queries. This system presents a user interface that helps in the configuration of data sources, and automatically proposes a global schema that presents a generic and simplified view of the data, which can be modified by the user. The system allows the creation of multiple star schemas from the global schema. Finally, analytical queries are also made through a user interface that uses drag-and-drop elements.

EasyBDI is able to solve recent problems by using recent solutions, hiding the details of several data sources, at the same time that allows users with less knowledge of databases to also be able to perform real-time analytical queries over distributed and heterogeneous data sources.

Contents

List of Figures

List of Tables

Acronyms

API	Application Programming Interface.
DB	Database.
DBMS	Database Management System.
DW	Data Warehouse.
EasyBDI	Easy Big Data Integration.
ETL	Extract-Transform-Load.
FDBMS	Federated Database System.
FDBS	Federated Database System.
GAV	Global-as-View.
GCS	Global Conceptual Schema.
HDFS	Hadoop Distributed File System.
IIoT	Industrial Internet of Things.
IoT	Internet of Things.
LAV	Local-as-View.
LCS	Local Conceptual Schema.
NoSQL	Not Only SQL.
OLAP	Online Analytical Processing.
OLTP	Online Transaction Processing.
UI	User Interface.
UX	User Experience.

Chapter 1

Introduction

Information is an important aspect of today's businesses. Analyzing data efficiently can lead to vital information that increases business performance. If more and better data exists and if it can be analyzed efficiently, the better information one can obtain, and therefore better decisions can be made. Not only in companies, but also in other institutions, such as health agencies that try to predict the areas with the greatest influx of registered cases of an epidemic, and thus better prepare resources to deal with the increasing demand for health services. However, this requires analyzing data that is located in various data sources, with different characteristics and organizations, making this a difficult task. In addition, obtaining the most recent data is also a necessary requirement in many use cases. The example mentioned above needs access to the most recent data so that it can detect that case as soon as possible. Access to the most recent information found in several data sources has been a recent problem studied in the scientific community.

This chapter presents a brief introduction to the area and background for the work done in this dissertation. Motivation, objectives, and contributions of this work are also discussed here.

1.1 Background

Database (DB) systems have been evolving over time due to the emergence of new requirements and needs. Initially, relational databases were prominently used by applications to store data of various domains, such as user information or bank accounts. This type of databases are optimized to store information in a centralized, structured and consistent manner [5].

Between late 2000's and early 2010's, the amount of data that needed to be processed or stored was already "too big, too fast or too hard for existing tools to process" [6]. The term Big Data is used to denote this phenomenon. In this context, the disadvantages of relational databases emerged, namely the fact that these could not scale as needed. Relational DBs also force data to have a specific type of structure. Nowadays, companies have interest in storing data that is semi-structured or unstructured, such as images or videos [5], but also data produced by sensors or machines in context of Internet of Things (IoT) and Industrial Internet of Things (IIoT) [7, 8], or even in data streams. According to [5], "a very large portion of the data (approximate 70% to 90%) in the business world today is unstructured data, while SQL can only handle the structured portion of data".

Several types of database systems were created later to address the disadvantages of relational DBs, such as higher availability, storage of unstructured data and horizontal scalability. These database systems are called Not Only SQL (NoSQL) and the philosophy behind them is that a state of consistency is not necessarily guaranteed after each transaction (data may be temporarily inconsistent or stale). It is preferred that these databases have a better performance on data handling, even if their state is not immediately consistent [5, 9].

A variety of NoSQL database systems already exist, each one more suitable for a specific purpose. Key-value databases, such as Redis, are good to store simple data that have no relationships [5], while graph databases, such as Neo4J, are designed to store information regarding the relationships between the data, useful for social networks or cloud management [5]. Columnar databases, like Cassandra, are designed for high availability and scalability and are often used for Online Analytical Processing (OLAP) systems [5]. Document databases, such as MongoDB, have no schema and therefore store data in the form of documents identified by an index that can have different fields [5].

Other types of systems store data in a distributed environment made up of nodes and can process large volumes of data in parallel, such as systems that operate over Hadoop Distributed File System (HDFS) or Google File System. The term *data source* will be used from this point to refer to a broader category of database systems, which can be relational or NoSQL DBs, HDFS systems or even flat files.

Given the wide variety of data sources available that cover multiple types of data and purposes, it is more and more frequent for large scale software systems to use several types of database systems for different modules. For example, an e-Commerce platform can use a key-value database to store information regarding a shopping cart and a document database to store the catalog of different products with distinct attributes [5]. But this causes data to be scattered between different database systems with distinct characteristics. It must also be considered that companies may have several databases geographically distributed, storing data for specific regions. Dealing with distributed databases means dealing with possible replicated or fragmented data and communication over the web, as well as dealing with distributed query processing [10].

1.2 Motivation

Business analysts need to analyze the data, regardless of their origin. However, when the data are spread across multiple systems with distinct data models and languages, it is necessary to know about their organization and inherent of all the data sources, which may not be suitable for some analytical scenarios involving several data source systems with distinct data models. Data integration offers solutions to unify all data residing in different data sources into one single point of access, which provides consistent data models while also maintaining semantic accuracy, and thus omitting the complexity of querying different data sources.

One of the most adopted solutions for data integration is known as data warehousing, which uses a process known as Extract-Transform-Load (ETL) to integrate data from multiple databases and load them into a single database. However, this is a complex and expensive process, and is only executed periodically, not showing the most recent data. However, in many use cases such as IoT, near-real time data access is a requirement for data analytics [7, 8], but traditional data warehousing techniques do not support near real-time querying. In traditional data warehouses, if changes need to be made to the internal organization of data in the warehouse the process to do so is expensive and will take time before those changes can be seen because data needs to be reloaded and reintegrated in the data warehouse. During this time, the system may be offline, and in this case, it cannot be used.

Another approach for data integration is the creation of a logical (or virtualized) data warehouse where data would not be integrated and loaded to a single database, but instead would be fetched directly from each data source while a unified schema is responsible to integrate all data, thus allowing for near-real time querying. Many systems have been developed showing different implementations of the aforementioned data warehouse logical approach [3, 7, 8, 11, 12, 13], however some are complex to use [7] and need lot of work to configure or to create queries, while others are easier to use, but take a step backwards by forcing users to know details regarding the internal organization of the data sources [12, 14]. Additionally, from the research made for systems that support queries against heterogenous and possibly distributed databases (which will be discussed in Section 2.4), none of the found systems support or were designed for data analytics applied to decision-making environments.

Regarding the advancements made in data analytics systems, new architectures of data warehouse systems have been proposed in order to achieve near real-time querying [15, 16, 17], but these systems have a more complex architecture, employ data materialization techniques and do not use the logical approach. Moreover, the task of executing analytical queries can also be made by users that do not have any experience in the area of database systems, and are only interested in performing analytical queries without needing to know which data sources are used, how to make the necessary configurations in order to be able to execute queries and how to execute queries using a querying language. None of the systems discussed employ a high-level design to assist these users in order to make analytical querying easier in order to include people from other areas.

1.3 Objectives And Contributions

The main objective of this dissertation is to create an abstraction over the underlying data source's distribution and characteristics, so that it is possible for users to issue queries to all systems simultaneously, as if only one database existed. The main use case are decision making environments based on distributed and heterogenous data sources (e.g., decision making based on IoT data). Analytical queries should therefore be the main type of queries executed over this abstraction. Traditional data warehouses present limitations, such as not being able to provide recent data, or the cost of the materialization processes, which make any changes to the underlying structure of data very lengthy.

Specifically, the objective is to develop a system that can meet the following requirements, all of which are an integral part of the previously named objectives:

- Distributed Data Source Querying - queries over distributed data sources must be processed and executed.

- Heterogeneous Data Source Querying - the system must deal with the heterogeneity of each data source, such as data model, and adapt a query such that it is compatible with each data source, while also allowing the creation of queries that refer to multiple data sources at once.

- Provides Location Transparency - details regarding the location of each data source must be hidden.

- Automatic Global Schema Creation - the system must propose a unified view of the overall data sources (in order to simplify the complexity of dealing with various schemas) which can be corrected by the user.

- Tools Enabling Analytical Queries Oriented Towards Decision-Making Environments - must support the execution of analytical queries by offering tools that support decision-making environments, such as multidimensional models.

- Provides a Guided Star Schema Creation - creation of a star schema should be done interactively, by selecting the tables to be mapped from the global schema, thus facilitating the configuration process.

- Deals with Physical Data Partitioning, Providing Transparency - data partitioning (horizontal or vertical) must be made transparent by aggregating all data as if only one node exists.

- Specification of Constraints Over the Global Schema - primary key and foreign key constraints must be possible to be created on attributes.

The methodology to attend to these requirements is to first search available systems capable of querying distributed and heterogenous data sources and conduct experiments in order to better establish its capabilities and limitations. In other words, the first two requirements will be achieved by using an already existing system as a base, as implementing such a system from scratch is a long and complex task. After a system is selected, more functionalities should be added using that system as base, depending on the selected system's limitations. Data integration techniques should also be researched and implemented in order to enable the automatic creation of a unified view (global schema) of the underlying data sources and to simplify the complexity of analysing multiple schemas. A way to submit queries to the underlying tool must also be implemented. Finally, in order to validate the developed system, concrete case studies will be used to test and demonstrate the system's capabilities.

This dissertation's main contributions is a system that allows non-expert users to perform real-time analytical queries over distributed and possibly heterogenous data sources. Additionally, an abstraction must be created so that the complexity of dealing with multiple schemas from distinct data sources becomes imperceptible to the user. The system uses recent methods to solve modern problems, namely by not materializing data like most data warehousing systems, but by querying data where it resides.

Multiple tools and systems that can perform queries over distributed and heterogenous data sources or (near) real-time analytical queries already exist, either developed as part of research or as commercials systems. However, a system that offers both of these features simultaneously has not been found (to the best of our knowledge) in the literature. The system proposed here can perform both, thus contributing to the state of the art, by using the logical approach, which is less commonly used. By accessing data directly where it resides, without any data materialization, it is easier to change the overall organization of any multidimensional schema created.

Finally, this work resulted in a demo paper, accepted in International Conference on Extending Database Technology (EDBT) 2021 [18].

1.4 Dissertation Structure

The rest of this dissertation is organized as follows:

- Chapter 2 presents a literature review on data integration and querying distributed and heterogenous data sources. Examples of available systems that solve some of the previous problems are also presented, and an overall comparison is performed between the systems by verifying which of the aforementioned requirements in 1.3 each system attends to.

- Chapter 3 introduces and explains in more detail the distributed query engine used in this work, while also exploring its capabilities, limitations, how to use it and how it can be useful for the overall system being developed.

- Chapter 4 describes the system's architecture and discusses the main challenges found throughout the development of the system and explains the implemented solutions.

- Chapter 5 explains how the system works, showing all implemented features and the user interface, and describes how to use the system, while also justifying some design choices.

- Chapter 6 presents the evaluation of the implemented solution using two case studies in order to determine the system's capabilities and limitations.

- Chapter 7 offers final remarks and the key takeaways on the final system and its potential contributions and limitations. It also proposes guidelines for future work.

Chapter 2

State of Art

Integration of multiple distributed and possibly heterogeneous database systems is a field of research since the 1980's, even before the appearance of NoSQL databases or systems that operate over HDFS. Being able to use multiple distributed databases with different characteristics in one integrated system raises several challenges. As database systems evolve and with the increase of new constraints, such as Big Data, new issues and potential solutions are discussed in the literature. This section describes some of those issues and solutions, as well as some of the systems and frameworks created to handle distributed and possibly heterogeneous data sources. System that offer solution to the growing necessity of near real-time analytical querying are also presented here.

The scope of this dissertation uses concepts from multidimensional analysis and assumes the user is familiarized with these terms, namely facts tables, dimensions and measurements and different types of multidimensional schemas (although the focus will be on star schemas). A good introduction to multidimensional analysis can be found in [19].

First section will cover data integration challenges, concepts and architectures. The following section will focus on existing problems when querying distributed homogeneous and heterogenous data sources. Afterwards, techniques that allow the creation of a unified schema providing an overview of the data sources are presented. In the end, a discussion is presented on systems that use architectures more suitable for querying distributed and possibly heterogenous data sources or for near real-time querying.

2.1 Data Integration

Data integration combines data from different sites and offers a single point of access through a unified view of the overall data sources, such that it is possible to query all data from all sites as if it was only one data source. Usually, this single point of access is a global database, providing access to multiple distributed local data sources, or sites. The design to create such database can follow 2 approaches: top-down or bottom-up.

The top-down approach starts with a global database providing a single point of access and a unified view of the overall local sites [20, 10], and only then creates each individual data source, each one closely following the schema and constraints specified in the unified point of access. The global database provides a query interface consisting of mechanisms that provide ways to submit commands to the system and query engines that deal with data in different ways. The global database also provides a query language from which it is possible to query all data sources within the system regardless of their data model (such as relational, document, or any other unstructured data model) [10]. Relational data model is used here as an example for the purpose of this explanation, having also been the data model adopted in the literature to exemplify concepts. Data sources are distributed but homogeneous, meaning that all share the same data model, schema and query language, while also being tightly integrated with the system.

However, this is not the case in many contexts, such as large companies with multiple departments responsible to create and design a data storage site. Each department would choose a specific database system best suited for a certain task while also designing its schema or data organization in order to meet a particular need, thus introducing heterogeneity among all data sites. Because each data site is designed first, the single point of access must be created in such a way that it is possible to integrate all data in the unified view. This is known as the bottom-up approach and will be further discussed in Section 2.3.

The use of different database systems for a specific domain and target application is now a standard, even for medium to small companies or platforms because there are many databases with different characteristics, offering advantages that better suit different objectives [5], thus invalidating the use of top-down designs on many use cases. But this introduces new challenges, such as the use of different models between the databases and distinct ways to query each database due to semantics of query languages [20]. Distributed databases add the problem of communication over the web with each database.

The scope of this dissertation is only on bottom-up design methodology, because this work considers scenarios in which the databases are already created, i.e., already designed and deployed.

2.1.1 Challenges

Data integration's main challenges, when considering a bottom-up approach, are usually classified in 3 categories: distribution, heterogeneity and autonomy [20, 21].

Heterogeneity

Heterogeneity can be due to various factors in database systems or other DBMS: data model (relation, graph, key-value), query language, software and capabilities of the Database Management System (DBMS). The proliferation of multiple different data storage systems contributed for this type of heterogeneity. But even if the same type of database system is used, heterogeneity can still occur in the schema, which causes syntactic and/or semantic heterogeneity at the level of the data model [20, 22, 21]. For example, consider a company that has multiple subsidiaries with the same type of relational database system, each one containing data regarding its employees, sales and other business-related information. As each subsidiary is going to implement their own local database, the most likely scenario is that each one ends up with a different schema to store the same type of information. If a business analyst intends to query all this information on each database, then he is forced to know the details of each different database.

These differences in the design adopted by each entity is known as heterogeneity at schema level, and may be either structural or semantic [10]. The former includes conflicts such as data types, or constraints [10], and the latter refers to conflicts in the meaning of the data stored. More concrete examples of schema heterogeneities are [23]):

- Using different name variations to specify the same concept (such as a table or a table attribute).
- Using disparate data types for the same data domain (such as using varchar or numeric for the attribute 'employeeID'), table attributes, or overall table organization (such as missing table attributes and general mismatch in table structure).
- Using uneven scales for certain measurements in attributes.
- Attributes with syntactically similar meaning can have different semantic meaning. For example, consider 2 imaginary databases, DB1 and DB2. One attribute in DB1, stores averages of production cost per day without taxes, while other attribute in DB2 also stores production cost per day, but with taxes.

To understand that two components in different database schemas are semantically related despite these differences is known as schema matching [20], which will be discussed in detail in Section 2.3.1.

Distribution

Distribution refers to the physical distribution of data that may be in different data storage systems. These systems can be geographically distributed or distributed between machines [21].

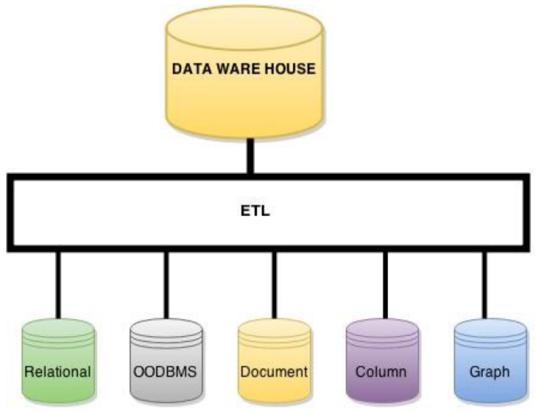
Autonomy

Autonomy defines the amount of operations related to data access or configuration that is possible to perform in a DBMS [20], or in other words, "the degree to which individual DBMSs can operate independently" [20]. A database system with no autonomy means that it cannot receive requests directly from the users or client applications, i.e., requests must be sent through a mediator. The level of autonomy of a database can also be variable, with higher autonomy database systems allowing the execution of more operations than less autonomous ones. Usually, a database system with high autonomy generates more heterogeneity [21]. An example of database with no autonomy would be to have a cluster of machines, each one with a database system, but no operations are allowed on an individual machine, and the only way to execute operations is by using a central DBMS that controls all databases in the cluster. In this case, each database in a cluster is not autonomous.

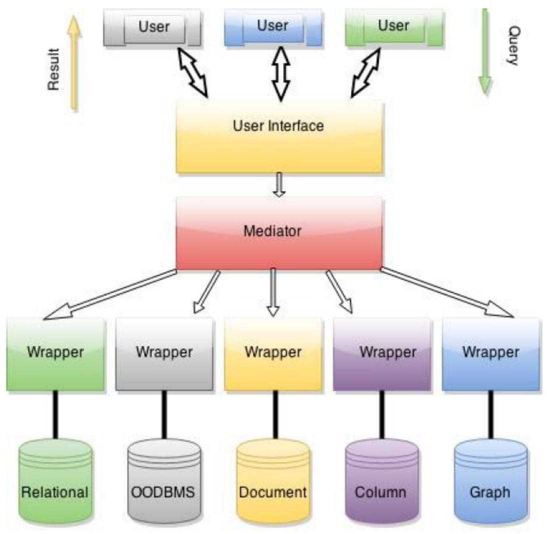
2.1.2 Methodologies And Approaches For Database Integration

When designing a distributed database system using a bottom-up methodology it is necessary to build a unified view of the overall data sources that already exist. These data sources are designated as local data sources (or local databases, but the broader term "data sources" is preferred here), each of them containing its own schema, which will be from here on out designated as Local Conceptual Schema (LCS) or simply local schema. The objective is to create a Global Conceptual Schema (GCS), also known as mediated schema or global schema, with a simplified and integrated view of all LCSs.

There are 2 ways to achieve this: the materialized approach and the logical approach [20, 9].



(a) Materialized Approach [9].



(b) Logical Approach [9].

Figure 2.1: Approaches for database integration.

The data materialization approach, depicted in Figure 2.1a, is currently the most used by enterprises for the implementation of decision support systems. These systems may use Online Analytical Processing (OLAP) to create and view multidimensional cubes and display data organized in different dimensions, while also performing specific operations present in such cubes, such as slice and dice. To extract data from each local data source, scripts are created for each one and data is then extracted, transformed and loaded (ETL) into one database, called data warehouse [9], which can support large volumes of data and is organized in such a way that allows users to perform analytical queries on data loaded from several data sources. OLAP operations aim to analyze historical data, therefore each table can be very large and queries can be very complex [10]. This is one of the downsides of performing ETL operations, as they tend to be costly and take a long time to integrate all data due to its large size. The materialization of data is a heavy process, that is costly if any changes to the data structures and organization need to be made, as these changes will take more time to take place. Data is extracted from each data source periodically which means the data stored in the data warehouse may not be up to date. This approach also generates data redundancy because the same information in each data source also exists replicated in the data warehouse, even if organized differently in the latter. Data warehouses are not very suitable for medium or small companies in an economic point of view, as setting up a data warehouse with a very large storage capacity is expensive and therefore not affordable for small to medium enterprises [24].

Other emerging fields, such as IoT or IIoT, have data constantly being generated from several sensors. These projects need to make analytical analysis on the most recent data, or near real-time analysis, but the materialized approach is not compatible with this because the complexity of ETL operations cannot guarantee that recent data is available in the data warehouse. OLAP operations use only relational structured data and cannot support the unstructured data of some NoSQL databases or other data sources, and both HDFS and NoSQL systems do not natively support OLAP operators existing in traditional data warehouses [25].

Finally, data warehouses are usually not the preferred approach to store data streaming, and instead HDFS tools are used to perform such tasks.

The logical approach, depicted in Figure 2.1b, provides an alternative that might suit these needs, as queries are made directly to each data site and only the needed data is used from each local data source [9], therefore reducing the complexity of extracting and transforming data as seen in ETLs. Directly querying the data where it resides also makes it being the up-to-date (or nearly up-to-date), which data warehouses could not ensure.

The logical approach uses an architecture commonly referred in the literature ([23, 20, 21]) as the mediator/wrapper. The base idea is that a mediator creates and holds the GCS discussed earlier and is able to receive a query using this view and decomposes it to be coherent with the LCS of each data source. The decomposed queries are forwarded to wrappers that are responsible to communicate with a data site [9] (as seen in Figure 2.1) and pass a query compatible with the data source's data model and query language. The decomposed query is executed by the management system of the data sources [10] and the results are sent to the mediator by the wrappers, which should again transform results from the local schema view to the global schema view.

A few problems can also be pointed out with this approach. For example, having local users that are issuing queries to a data source directly for Online Transaction Processing (OLTP) (i.e. a costumer consulting the list of flight schedules) and global users issuing complex analytical queries to these data sources from the global database at the same time can degrade the performance of each data source and increase the time of execution for OLTPs, thus giving to the end user an overall unpleasant experience on the platform [20]. Another aspect is that the logical approach does not have any incorporated tools for OLAP, as is the case with data warehouses.

This dissertation focuses on developing a system that logically integrates data in order to fulfill the objectives established, therefore, the remaining of this section will be focused exclusively on concepts and systems that employ the mediator/wrapper approach.

Most of the literature analyzed regarding this topic present or propose several taxonomies to classify different systems while also introducing variability in terms such as *multidatabase*. For example, Sheth et.al. [23] (from 1990) proposed that *multidatabase systems* was a more generic term and *federated systems* was a type of *multidatabase systems*. Other authors such as Chen et.al. [21] also use this taxonomy. However others claim that these terms are synonyms, such as Özsu et.al. in [20]. Therefore, this dissertation will use the term *multidatabase systems* to refer to systems that integrate multiple distributed and possibly heterogeneous data sources using a logical approach and a Bottom-Up methodology. A taxonomy will also be used and discussed in Section 2.4 that categorizes multidatabase systems in 2 groups and is in part influenced by the one presented by Tan et.al. in [3].

2.2 Distributed And Heterogeneous Data Sources

When dealing with distributed queries, a few key problems must be considered, namely data partitioning (also known as fragmentation) and data replication, which should be imperceptible to the end user that executes queries using the global schema in the global database. When using homogeneous data sources, the only issues are data replication, data partitioning and query optimization (note that these issues are the same for database systems designed in a top-down approach), however with heterogeneous data sources more issues arise. Although query optimization is a concern that has been widely discussed in the literature, it will be omitted here because it is not within the scope of this dissertation.

A quick introduction to data fragmentation and data replication is presented next, followed by the issues that querying heterogenous data sources introduce.

2.2.1 Data Partitioning And Data Replication

If a database server fails and no replicas of such server exists, then clients will be unable to access such information. Creating replicas or partitions of a database server can improve fault tolerance, so even if one server fails, another exists to receive and process the request.

One solution is to replicate the data across several servers, but data must be kept consistent across all nodes, meaning data should be updated by all nodes regularly. However, a query may update data in one node and another query that references the same data may be issued to another node who does not have a copy of the data up-to-date yet when the transaction occurs. The updates of data across nodes may happen in two forms:

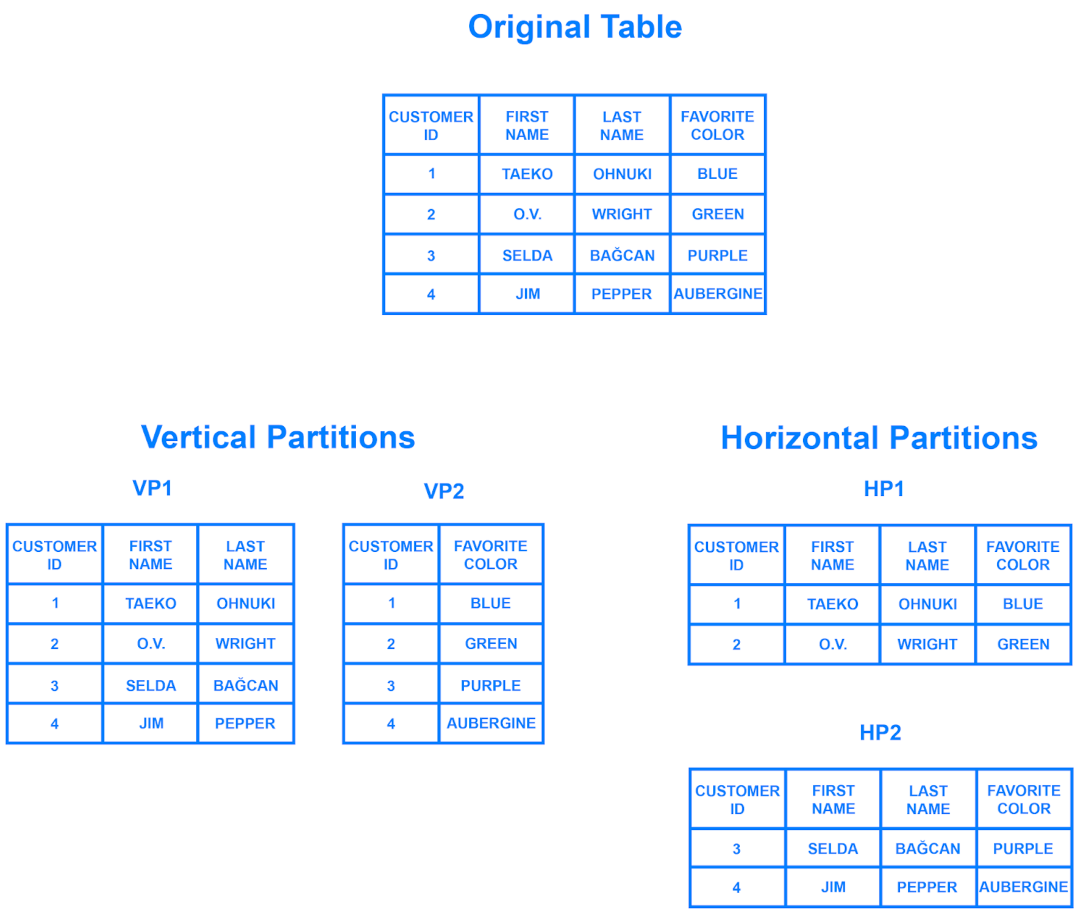
- lazy - A master node keeps all data up-to-date and the master propagates the updates to each node (lazy update). [10]. Queries made to slaves may not be updated.
- eager - All updates are applied to all nodes before any new transaction is performed (eager update). [10]. Queries made to any node always have updated data.

Other solution is to perform data partitioning, which is usually performed on tables from relational databases. Partitioning tables involves creating smaller tables with less records on each one and then transferring each new table to new servers, which can alleviate the load of each server because the requests can be spread by different servers, as well as to decrease the time needed to search the table's records. The main data partitioning techniques commonly used are vertical or horizontal partitioning, both methods illustrated in Figure 2.2.

In vertical partitioning, columns are divided across several tables. Each new table must have a primary key which also is a foreign key referencing the primary key of the first table. This is a good strategy to split more detailed information that may be less needed into other tables, and therefore it is less likely that those tables need to be accessed in the same query through joins. For the same reason, it is also adequate to create tables containing columns that are frequently accessed together [20].

In horizontal partitioning, a table's rows are divided into several new tables, whose columns remain the same. In order to balance the requests evenly for each table, one can partition the records through a strategic condition such that the same amount of records are split across tables and that records in each table have similar likelihood of being accessed. As a simple example, it would be wise to split a 'clients' table by age or gender of the costumer if there is an evenly distribution of entries across several tables. All clients with ages 20 through 30 could be placed in one table and clients 40 through 50 in another one. A bad example is to split by country, since some residents have much more population than other countries, creating an unbalanced distribution of data. If certain records are accessed less often than other's, partitioning can be made such that those records are placed in one table, and therefore less likely to be accessed [26].

A distributed database system will provide partitioning transparency, replication transparency and location transparency, which means that the location of the data and any replication or partitioning is completely invisible to the end user, thus making it much easier to deal with a set of distributed databases.



Original Table

CUSTOMER ID	FIRST NAME	LAST NAME	FAVORITE COLOR
1	TAEKO	OHNUKI	BLUE
2	O.V.	WRIGHT	GREEN
3	SELDA	BAĞCAN	PURPLE
4	JIM	PEPPER	AUBERGINE

Vertical Partitions

VP1

CUSTOMER ID	FIRST NAME	LAST NAME
1	TAEKO	OHNUKI
2	O.V.	WRIGHT
3	SELDA	BAĞCAN
4	JIM	PEPPER

VP2

CUSTOMER ID	FAVORITE COLOR
1	BLUE
2	GREEN
3	PURPLE
4	AUBERGINE

Horizontal Partitions

HP1

CUSTOMER ID	FIRST NAME	LAST NAME	FAVORITE COLOR
1	TAEKO	OHNUKI	BLUE
2	O.V.	WRIGHT	GREEN

HP2

CUSTOMER ID	FIRST NAME	LAST NAME	FAVORITE COLOR
3	SELDA	BAĞCAN	PURPLE
4	JIM	PEPPER	AUBERGINE

Figure 2.2: Example of relational table partitioning. Taken from [1].

2.2.2 Querying Heterogeneous Data Sources

On top of the previous issues with data replication or data partitioning, other issues arise with heterogenous data sources.

First, each data source may have different limitations in regards to operations that can be performed [10]. For example, some DBMSs may not support certain types of complex joins or aggregations [10]. Heterogeneous data sources may also have different query languages suited for different types of data models (document, relational, key-value), therefore the translation process of a query from the global database to a local data source is more complicated [10]. Autonomy of data sources is also a concern, as data sources with higher autonomy can, for example, end it's communication with other systems because it was terminated [10] by an administrator or maybe because its configuration details have changed.

Finally, semantic heterogeneity discussed in Section 2.1.1 must also be solved. Next section will focus in more detail on how the global schema is created and how to solve these issues.

2.3 Global Schema Generation

A global schema is created by integrating and mapping the LCSs of each local data source [27]. This global schema should map similar objects between all data sources into one object, therefore offering a simplified view of the data sources within the system. Additionally, this global schema should also contain information on how to translate each object in the global schema back to the local schema.

In order to create the global schema, three fundamental steps are required:

- schema matching - identify data similarities (syntactic and/or semantic) between different objects [20].
- schema integration - integration of multiple local objects with same information type into one global object [20].
- schema mapping - creation of mapping between all the global objects and local objects [20].

Each of these steps has its own challenges and will be explained in the following sections.

Different data sources may use varied structures to organize data, such as tables in relational or collections in document-oriented databases, however, for simplicity, relational model will be used in the examples given to better help explain certain concepts.

2.3.1 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. The objects to match can be either tables or columns, depending on the scope of schema matching. For example, if the objects are tables, schema matching should detect that the tables "employees" and "employees_info" located in different databases contain the same type of information, which is data regarding employees of the company.

This can be achieved in two ways: either by looking at the structural organization of the table or by looking at the format or content of instances of data. The former is known as *schema based matching* (or *structured matching*) and the later as *instanced based matching* [2]. Inside of each category, there are many strategies for schema matching. Several approaches can also be use together. A summary of all possible schema matching approaches can be organized as in Figure 2.3. This taxonomy was proposed by Rahm and Bernstein [2] and is widely adopted in the literature, such as [28].

Schema matching algorithms provide a measure of how similar two elements are. Usually a similarity value ranges between 0 and 1 is used, in which 1 means that the two elements are very similar, possibly the same. If the similarity between both elements is within a certain threshold, then they are considered semantically equivalent, and therefore a match.

Schema Based Matching

Schema Based Matching techniques use information of the schema metadata in the objects, such as names, data types, inter-table relations and overall schema structure [2]. There are also two categories of schema based matching that classifies what information is used to assess the similarity:

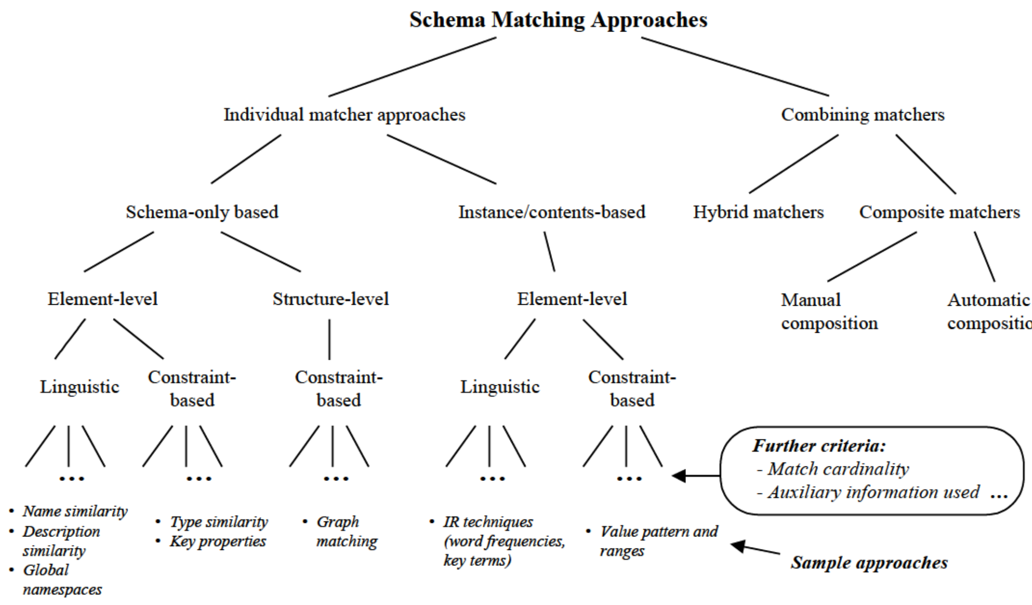


Figure 2.3: Schema matching approaches, as specified in [2].

- *element level* - uses data such as names and data types of the schema [10, 28], and two approaches exist: linguistic and constraint based.
- *structure level* - uses the relationship between the elements [10, 28], and only one approach exist: constraint based.

At element level schema matching, there are two approaches: linguistic and constraint-based. Linguistic approaches usually use the name of the elements and check how similar they are. Some metrics that can be used to measure name similarity are n-grams, Jaccard index, and Levenshtein distance. A dictionary with a list of synonyms can also be used to identify name similarities whose previous mentioned metrics would fail, such as 'car' and 'automobile' [2]. Dictionaries for hypernyms (a word that is more general than another), such as vehicle and car, and for homonyms (terms that are pronounced equally, but may spell the differently and mean different things) can also be used [10, 2]. This approach however, does not guarantee that two tables are actually semantically identical. For instance, two 'budget' tables can refer to two different concepts of budgets, such as budget for products, or budget for stores, two concepts that can be viewed as different [20].

Constraint approaches use information regarding constraints of the elements, namely, data type, primary key, foreign key and unique constraints. The data type of the elements could be a good decisive factor in some cases. If two columns contain data types such as 'date', they could be good candidates to match. Looking at Figure 2.4, the columns 'EmpNo' and 'Pno' have the same data type and both are unique ('EmpNo' is a primary key, which also makes it unique), which suggests that these are good matching candidates. However, some DBMSs have different names for the same conceptual data type, such as varchar and string. Similar data types may also be used for the same attribute, such as double or numeric, and can create imprecise matches [28]. An example in which this technique fails is if data regarding phone numbers are stored in one column as an integer, and on the other column is placed as a string, as these will not be considered a match, even though they are semantically identical.

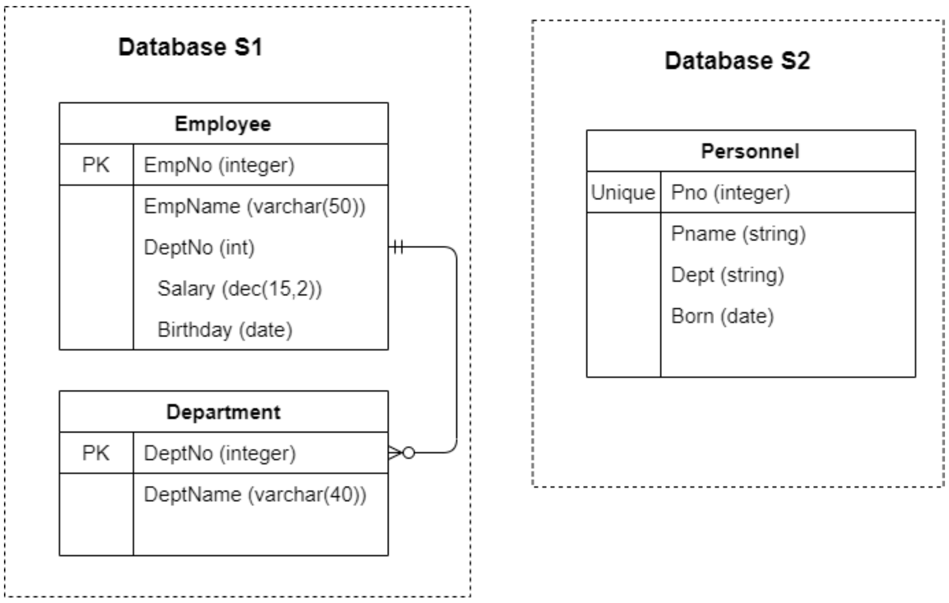


Figure 2.4: Schema matching using constraint based approach. Taken from [2].

Instance Based Matching

Instance Based Matching techniques consider the instances of data instead of the schema and analyze the formats of data in order to decide upon matches, which is better suited for cases when the data is semi-structured and schema provides little information [2, 28]. There is only one category of instance based matching, which is the element level. No instance based matching techniques exist at the structure level as with schema based matching, because only the instantiated data is analyzed. Inside of the element level category, there are two approaches for instance based matching:

- *linguistic approaches* - employ information retrieval techniques to textual information [10].
- *constraint approaches* - analyze ranges of values in data [10].

Both these approaches are similar to the ones seen on schema based matching at element level, however the techniques here employed focus on instances of data. Linguistic approaches can look at instances and count the most frequent keywords or combinations of words [2]. If two elements contain the same frequent words, they could be matches.
Constraint based approaches can analyze the ranges of numbers in 2 elements, or averages of character patterns [2, 10]. This could allow the detection that 2 elements contain data with a consistent format, such as dates or phone numbers. Instance based matching techniques are less efficient because we have to instantiate and work with the data directly.

There are also approaches that employ machine learning to perform schema match, called learning-based approaches. As all machine learning projects with supervised learning, it requires a dataset with examples of correct schema matches. This set is called training set, and it is used to 'learn' when to establish matches using probabilistic models [20]. When feeding another set of elements to the model, it should predict with a certain accuracy if they are or not a match. This accuracy depends greatly of the quality of the training set and the methods and parameters used for the model.

Combination Of Schema Matching Approaches

Previously described schema matching approaches can be used together to improve matching results. For example, one could look at the schema information and use linguistic approaches to check how similar the names of elements are, while simultaneously use constraint-based approaches to check their data types, and look at the instances of data in order to analyze their format. The combined results of each approach may yield a more accurate similarity result.

It is possible to combine matching approaches in 2 ways: composite or hybrid.

The hybrid way of combining match algorithms joins them all into one algorithm to analyze elements and give a similarity score between the elements [20]. Considering the previous example, we can create an hybrid algorithm that analyzes the name and the datatype of the elements and returns a similarity score between them.

Composite works by successively applying matching approaches to elements, obtaining a similarity score for each approach and then combining each individual score into one final score [20]. This final score can be obtained in various ways, from average to more sophisticated ranking algorithms [20]. By using a composite combination, we could apply a data type comparison algorithm between elements, followed by a name similarity algorithm, and then use a certain function to combine the values of similarities obtained from both matches.

No matter how many combinations of algorithms are used, there is no guarantee that the match is correct, so human intervention is always necessary. What can be done and is currently proposed in the literature are algorithms, methodologies or tools to facilitate users in the schema matching step, since this this can be an arduous task.

2.3.2 Schema Integration

To build the global schema, it is necessary to merge all tables that were identified as semantically identical in the schema matching process into one global table that retains the same semantic meaning for each matching local table. But at the same time, local tables that had no match (meaning they are unique and there is no other table semantically similar) must also be mapped into one global table, but this is a trivial process, as it is enough to just create an equal table in the global schema with the same columns and make that local table as a match to the global table. The focus of schema integration is to integrate multiple local tables that match to more than one global table.

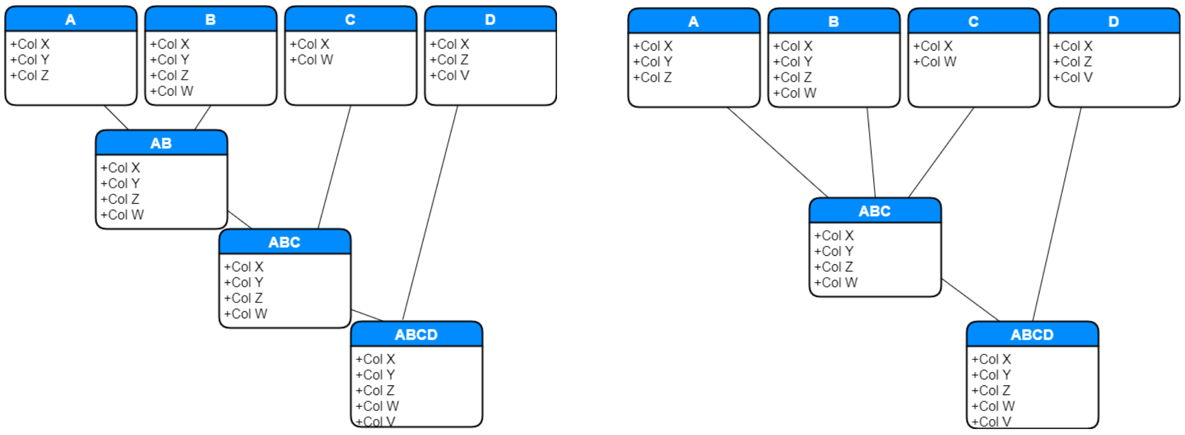
Note that there are cases in data integration in which the global schema is already defined and this stage is unnecessary [20], such as data warehouses, in which the global schema is pre-defined and the local schemas are then mapped to that global schema [20]. But for the purpose of this work, the global schema must be constructed dynamically, as it is not known in advance what data sources will have to be integrated and newer data sources may be added to the system in the future.

There are two main ways to perform schema integration: *binary* or *n-ary* integration.

In *binary* integration, depicted in Figure 2.5a, tables are merged two by two. Each merge creates another table which is used to merge with the remaining tables [20]. For example, in Figure 2.5a, local tables A, B, C and D were identified as similar and must all be a part of one global table. This approach starts with local table A and local table B, and created global table AB, whose attributes are the union of the attributes in both tables (attributes in both tables that are similar create a new attribute in the new table, and attributes that are different create a new attribute each). Table AB is now integrated with table C, and so on until all local tables are integrated and a global table is formed.

In *n-ary* integration, depicted in Figure 2.5b more than 2 tables are merged at a time [20]. Instead of merging tables A and B and creating table AB, tables A, B and C may be joined and create global table ABC, which will then be merged with the remaining local table D, as depicted in Figure 2.5b.
In the end, both approaches return the global table resulting from the semantically equivalent local tables.

Any partitioned or replicated tables, should be identified in the schema match as similar between each other, and schema integration would merge those tables into one.



(a) Binary Integration. (b) *n*-ary integration.

Figure 2.5: Schema integration approaches.

2.3.3 Schema Mapping

After the previous steps it is possible to identify which local tables match to a single global table, but there is no specification on how to actually move data from one schema to another while maintaining semantic coherence. It becomes necessary to specify how elements in the global schema and the local schema translate between them such that it is possible to query data from one schema to the other while maintaining a consistent semantic meaning in both schemas [10, 29]. To do this, mappings are created between the global and the local schemas and then used in queries between both schemas in order to preserve semantic consistency [30, 10]. These mappings can either be created from the local to the global schema, named Local-as-View (LAV) or from the global to the local schema, named Global-as-View (GAV) [24]. GAV allows an easier creation of mappings but adding a new data source forces some mappings to be redefined while LAV allows an easier addition of data sources but mappings are much harder to create and queries also harder to generate [24, 31].

Mappings are constituted by a transformation formula and a filter. The former changes a value(s) in one schema such that it makes semantic sense in the other schema and the latter restricts the values that can be used [30]. Figure 2.6 shows as an example a global table with sales of every store which considers only one currency while local tables have different currencies and some consider taxes while others do not. Functions are applied to convert to the correct currency and filters are used to consider only non negative values, therefore achieving semantic consistency. With these mappings defined, the idea is to create mappings for each local table correspondence such that a query capable of correctly retrieving data from the local table can be created [10]. This query should use these functions and filters in order to convert data from the local schema to the global schema.

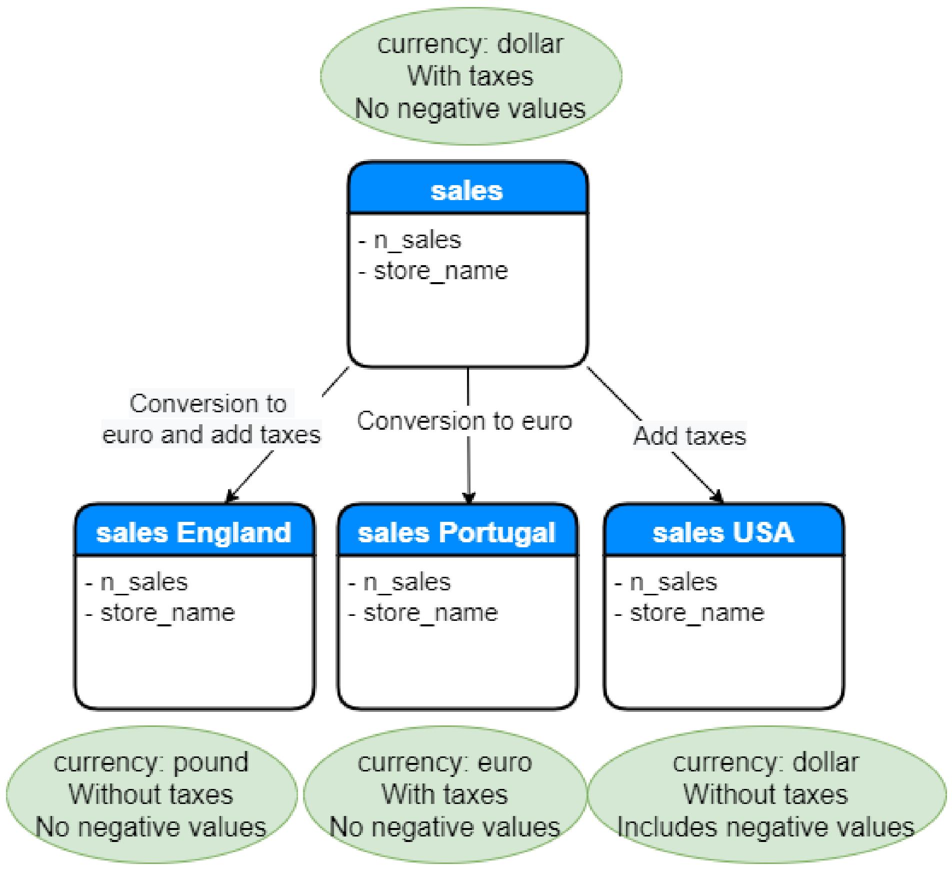


Figure 2.6: Example of schema mapping with multiple local tables containing different currencies and the global table using mappings to create semantic consistency.

Mappings are usually created manually by users. Ideally, only high-level details about the mappings should be provided by users and the system should generate all mappings accordingly, therefore simplifying this task for the user [32]. An example would be a user interface in which user's make visual correspondences between the schemas with mapping details in each correspondence [33] (not to be confused with schema matching correspondences, which are used only to pair similar objects, whereas schema mappings are used to create mappings between said relationships). There are also systems that have users program by example, meaning they supply a complete dataset or an incomplete dataset, which will then be used to generate schema mappings, such as Beaver [29].

2.4 Multidatabase Systems

Multidatabase systems use the mediator/wrapper architecture discussed earlier to logically integrate data. This section, discusses two types of multidatabase systems using the aforementioned taxonomy adopted for this dissertation, which divides multidatabase systems in two types: federated or polystores/multistores. Both types of systems will be explained and examples of some systems found in the literature or commercialized by companies will be presented.

2.4.1 Federated Multidatabase Systems

A Federated Database System (FDBS) is constituted by multiple autonomous database systems, each one with its own schema. A single database system can simultaneously participate in one or more federations and continue to carry its local operations [23]. The FDBS a set of local data sources that belong to the federation and a Federated Database System (FDBMS) which provides a single point of access to the local data sources [23]. The global schema used by the FDBMS contains schema mappings between tables in the form of views [34].

In order to make the LCS of a local data source accessible to the FDBS, an export schema must be supplied, which will be accessed by the FDBS [23]. The export schema is located between the data sites and the FDBMS and represents a subset of the schema of each data site [23], which means that database administrators can select which data can be made available for the federation [20, 23]. The export schemas of one or more database systems are integrated in a federation schema located in the FDBMS and presents a unified view of the export schemas of each database in the federation and information regarding the location of each database [23]. The federated schemas are then used to generate an external schema to be used by each user/application of a federation.

Users that want to access the federation using the FDBMS, here designated as global users, may be interested in different subsets of the information in the federation schemas. Therefore, an external schema is also used between the FDBMS and the global user, and allow a view of only a subset of the data present in the federated schema [23, 34]. The main purpose of the export schema and the external schema is to control and limit the exposed information to the global users [23].

There are two variations of FDBMS, illustrated in Figure 2.7:

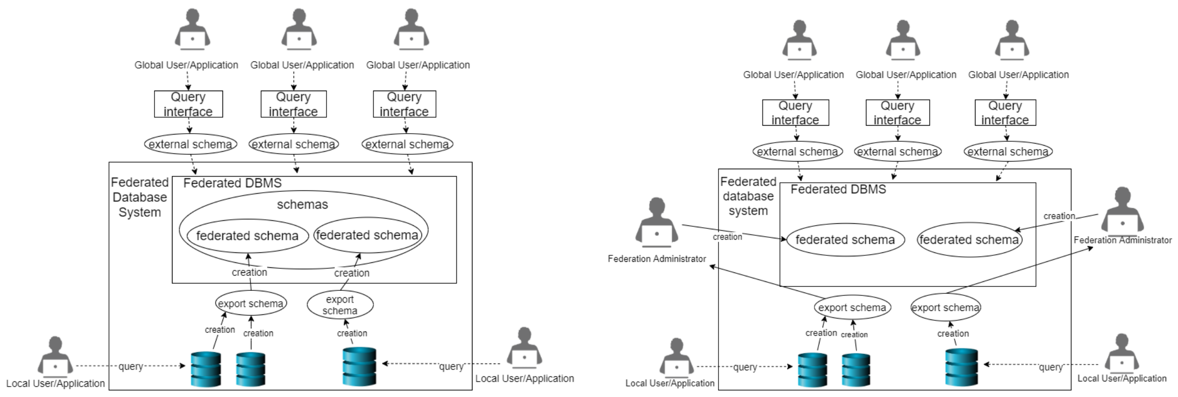
- Loosely coupled FDBMS: in this scenario, each administrator of each database must create the federated schema and make it available to the federation [21, 23]. Figure 2.7a illustrates the architecture of these systems.

- Tightly coupled FDBMS: the administrator of the FDBMS is responsible to create the federated schema for the federation [21, 23]. Figure 2.7b illustrates the architecture of these systems.

Below are some examples of systems found in the literature that are classified as FDBS.

MDB1 And MDB2 FDBS

Students of De La Salle University developed 2 multidatabase platforms as part of their thesis project: Samuel Cua, Gilbert Gaw, et.al. developped MDB1 in 1999 and Andres



(a) Loosely Coupled FDBMS. (b) Tightly Coupled FDBMS.

Figure 2.7: The two types of FDBMS systems.

Barlahan Jr., Ludwig Lim et.al. developped MDB2 in 2000. These systems serve as a good example of database integration systems because each one uses one of the data integration approaches discussed in Section 2.1.2. Note that in the year of this paper's publication, NoSQL databases and HDFS systems had not yet emerged, therefore the heterogeneity of databases considered in this paper was much smaller than today, as only different relational DB systems existed.

Both platforms have a user interface to manually and visually perform schema integration across databases with different schemas, which means users can solve the schema matching problems specified in Section 2.1.1 in a more efficient and less error prone way.

The first system, MDB1, is a tightly coupled FDBS [22] that does not perform a logical data integration as it is usual with most multidatabase systems, but instead performs materialized data integration because it physically instantiates and stores the global schema in a global database [22]. The system contains a global schema resulting from the manual configurations made by users. The data stored in the global database is periodically pulled from each database and data integration is performed each time, thus solving schema heterogeneities using rules previously specified by the user [22]. The drawbacks this system presents are the same as the ones presented in materialized data integration, which is that the data is not up-to-date in the global database. Additionally, the global database used in this system might not be able to hold all the data [22].

The second system, MDB2, is a loosely coupled FDBS [22] that logically integrates data because it fetches data directly from each database only when it is needed and only what is needed by dividing a query into sub-queries for each DB [22]. For example, if a user wants to aggregate data from DB1 and DB2 the system decomposes the query into subqueries and translates them using the global schema in order fetch necessary data from DB1 and DB2 and then performs aggregation using an external DBMS that is part of MDB2 [22], and finally returns the results to the user. The disadvantages presented by this system is the delays that may occur when transmitting the subqueries to each database and the the conversion of data from global schema to local schema each time a query is executed [22], problems that are commonly associated with most systems that logically integrate data.

2.4.2 Polystore And Multistore Systems

FDBS were the first to be discussed and conceptualized in the literature (back in 1990) and did not considered querying data sources with different data models, and instead focused only on relational data model. More recent work has been done to develop systems that deal with the variety of data models using additional solutions to deal with this extra level of heterogeneity, and therefore these systems distinguish themselves from FDBS by this key-difference. These systems allow data integration from data sources with distinct data models, such as relational and all NoSQL data models, and sometimes, even with systems using HDFS [10], commonly designated as SQL-on-Hadoop, such as Hive or Spark SQL.

The taxonomy defined in [3] classifies polystores and multistores as equivalent, except that polystores have more than one query interface, meaning users can interact with the system using more than one query engine, while multistore systems only have one [3]. An example of a system with multiple query interfaces is SparkSQL, which allows users to write SQL queries and interact with data in relational format or by using Spark's Dataframes and Datasets, which is a distributed collection of data commonly used by Spark [3, 35].However, other works offer no distinction between these terms, such as Valduriez et.al. [10].

These systems can be further classified into 3 categories: loosely coupled, tightly coupled and hybrid [10]. Loosely Coupled contain wrappers that use a local catalog representing its data source's data, keeping the data sources autonomous, and a Query Processor interacts with these wrappers to process queries, [10]. Tightly Coupled use the Query processor to directly access the data sources through its query interface which results in a better performing system but each data source has less autonomy [10]. Hybrid combine these methods by having wrappers access data sources directly and query processor deal with data received by through the wrappers [10].

BigDAWG Polystore System

BigDAWG is an open source polystore system developed by Intel Science and Technology Center for Big Data (ISTC) and follows the mediator/wrapper architecture. It is intended to simplify the use of data in scenarios where data resides in databases with different data models and other distinct characteristics. It uses a concept of islands as mediators [7], in which different islands represent a specific data model, query language, and a set of data storage systems [7]. An island offers a query interface from which to query all data sources within the island [7]. Ideally, users should configure islands for each type of data model they want to query and connect data sources with similar data models in the same island, but it is possible to connect data storage systems that are of different data models .

To connect a data source to an island, a shim is written and used as the wrapper to translate a query to the language of a target data storage system [7]. If a user wants to query data sources belonging to the same island (single island query), a query is decomposed into subqueries and then rewritten using the shims into the same language as the query interface of the target data source [3, 7], but the user must use a 'cast' operator to copy objects between data sources [7, 10]. However, if a query specifies data from data sources in different islands (cross-island query) then it is necessary to use a 'scope' operator of the BigDAWG query language to write queries for other islands [3, 7]. For instance, if a user wishes to join data between 2 data sources that are on the same island, then it must be specified in the query through the use of a 'cast' operator in order to convert from the data model from one DB to the other one. However, if a query is made that references objects in data sources from different islands, a scope operator must be used and the query language of the other island must be placed inside that operator.

Overall, this forces users to know details regarding the underlying databases and the organization in each island. Figure 2.8 shows the architecture of BigDAWG.

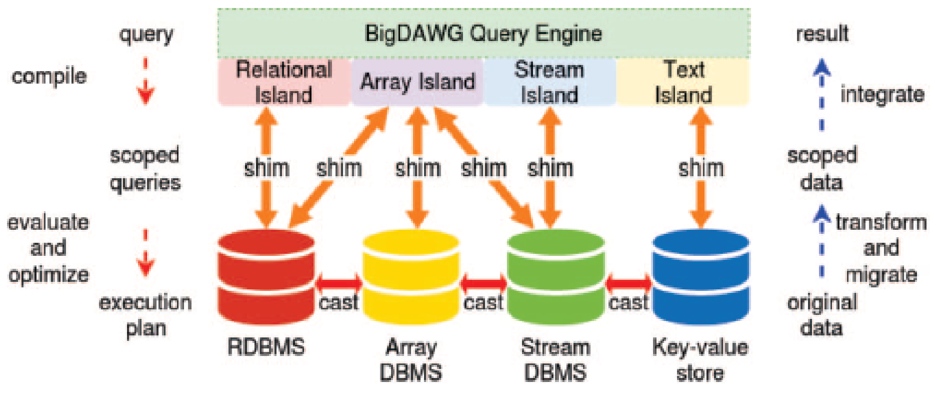


Figure 2.8: BigDAWG Architecture. Taken from [3].

Apache Drill

A more recent architecture of a logical data integration platform is Apache Drill, which is a distributed query engine ready to be scalable. It enables a much easier and straightforward way to access raw data over various data stores, removing the burden of schema configuration. Because no global schema is created, this is considered to be a schema-less system [8]. It is a distributed system because it is designed to be installed on a cluster of machines, but it can also run on one machine only. Queries are written in SQL query language. The architecture of Drill is represented in Figure 2.9 and is comprised by Drillbits, an installation of Drill on a machine (node) that handles the query (acts as mediator), and plugins that connect to data storage systems (act as wrapper).

When a user issues a query, a Zookeeper responsible to manage the cluster [3] is responsible for assigning an available Drillbit to receive the query, thus becoming the foreman [3]. The latter is responsible to process, decompose and generate an optimized plan for the other Drillbits to execute each subquery resulting from the query decomposition [3]. A subdivision of a query that agglomerates data from 2 different data sources can create 2 subqueries: one to fetch data from one data source and another to fetch data from the other data source. A Drillbit uses a plugin to communicate with the respective data source and fetch the necessary data [13]. The Foreman Drillbit finishes any processing and returns data to the user [13].

Presto

Presto is also a distributed query engine capable of processing big data developed by Facebook. It can handle large amounts of data originating from heterogenous data sources, while also allowing for an easy way to quickly access that data, because, similarly with Drill, no

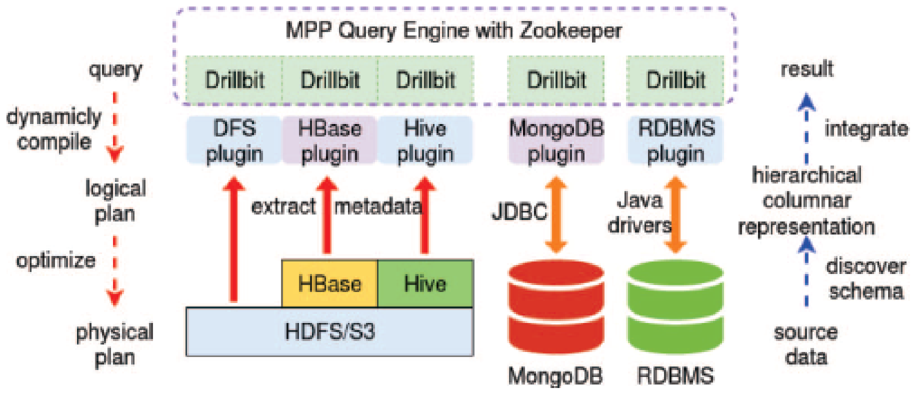


Figure 2.9: Apache Drill Architecture. Taken from [3].

global schema is used in Presto, meaning that it is schema-less [8]. Users use SQL as the query language to issue queries to the system. It uses a master-slave model, with the master being named coordinator, and slaves called workers. The coordinator is responsible for receiving the query and then parsing it, planning and optimizing it as well [12]. The coordinator creates subqueries and generates a distributed plan to assign subqueries to workers which are then responsible to fetch data from data sources [12]. Workers can cooperate in order to support the processing of more than one query simultaneously [12]. Connectors act as the wrapper that allow communication with a specific data source. They are extensible because they can be created and modified by anyone. Figure 2.10 illustrates the architecture of Presto.

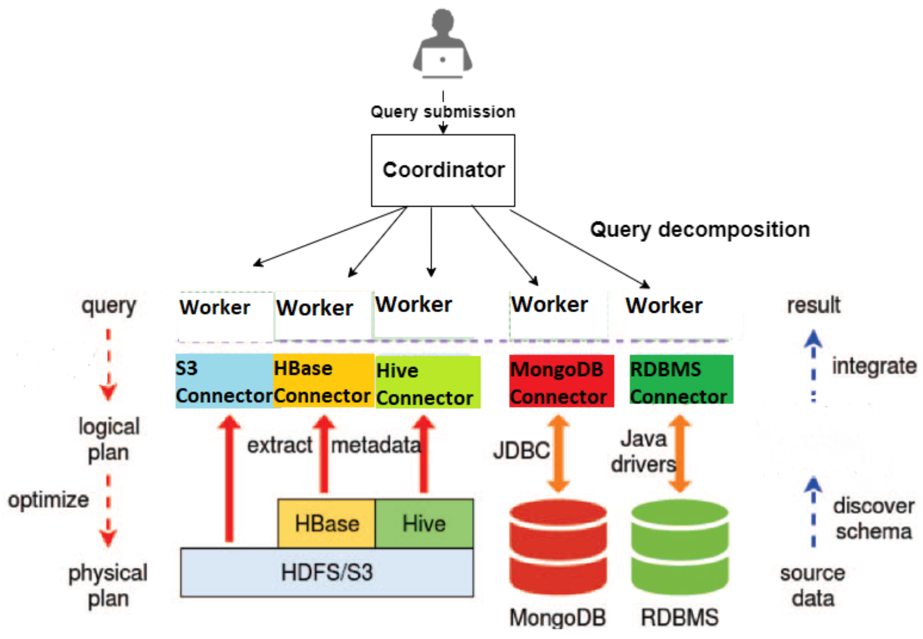


Figure 2.10: Presto Architecture. Adapted from [3].

Other System Examples

Other example of systems include CloudMdsQL, a query engine capable of querying heterogenous data sources within a single query [11]. It provides a new query language based on SQL queries, but with the possibility of including nested queries using native languages of the data stores to retrieve tables, or by using Python expressions, and assigning a name to the resulting tables [11]. Its architecture is based on the mediator/wrapper approach, and is composed by masters and nodes, distributed on clusters of computers. Each node contains one master, responsible to plan a query, and a worker to execute it, although queries are decomposed and assigned to several workers [11]. Queries are also optimized using a cost model for each query plan. CloudMdSQL does not create a global schema, which makes it robust whenever the schemas of each data source change (schema evolution) [11]. Because no global schema is created, queries require a nested query that refers to a particular data source's data structure, usually by formulating the nested queries in the native query language of the target data source.

MusQ is a system adopting the mediator/wrapper approach, and capable of querying heterogenous data sources. It was developed primarily for IoT and is capable of generating a semi-automatic global schema by employing schema matching techniques to detect semantically similar matches, and schema mapping to create mapping definitions between local and global schema tables, which are then validated by the user [8]. The query language used is MQL, and submitted queries are parsed followed by the generation of a query plan in order to decompose and create subqueries, which are then sent to wrappers that communicate with the data sources.

2.5 Near Real-Time Data Warehousing

Traditional data warehousing techniques and architectures are not compatible with some modern uses cases, such as e-Commerce, health systems and stock brokering [15] or even other areas such as IoT, because they all require fresh data, but traditional DWs ETL methodologies only extract data periodically, with a large interval between extractions, such as hours or days. Additionally, while ETL is being performed, analytical operations cannot occur, thus causing constraints for systems that need be operating 24 hours every day [36].

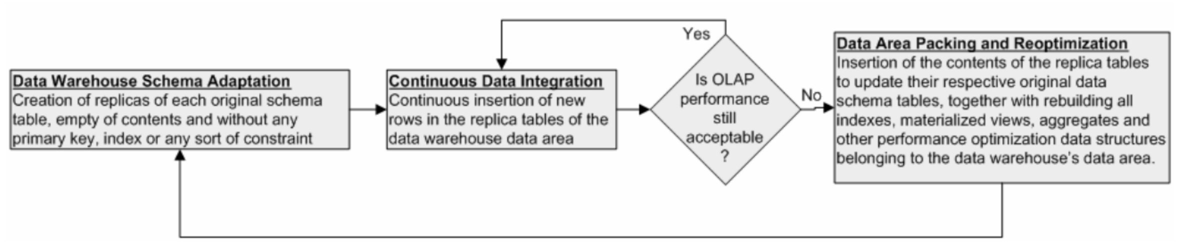
Near real-time data warehouses propose to solve these problems and make available recent data without any system downtime by having the ETL process more lightweight and more frequent [16]. Near real-time data warehouses architectures and systems have been proposed by many authors.

Santos and Bernardino ([15]) proposed an architecture whose objective is to solve some of the problems found in traditional DWs, namely the simultaneous execution of analytical queries and continuous data integration, with minimum performance loss. To do so, they proposed the creation of temporary tables where only the most recent records will be inserted, but without any constraints or indexes in order to quickly insert data updates from ETL processes. The temporary tables need to be relatively small, and their data will be added to the original tables in the DW to give space for more recent data in these temporary tables. Figure 2.11a depicts this procedure. Any analytical queries made explicitly specify the tables to be queried: the original tables with historic data, the temporary tables with recent data, or both, in which case a *union all* should be used to present data from both star schemas. The main advantages are the reduction of the lengthy downtime seen in traditional DW when extracting and loading data to the warehouse and also allows for a quicker way to access fresh data. One disadvantage is that average query execution times increase when comparing with traditional DWs [15].

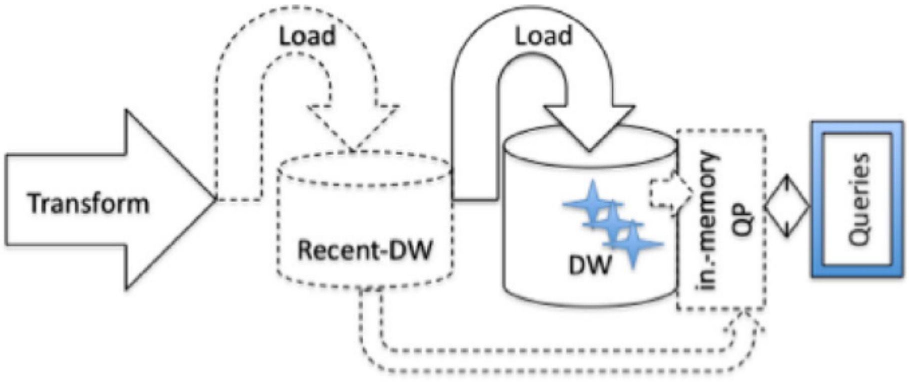
Zuters ([16]) also suggests that near real-time data should be separated from older data by placing recent data in partitions, each one with data from different time ranges, (one partition with data from 10 minutes and other with data from from 1 hour) [16]. Staging areas also exist to first receive the data and then load it into the partitions, in order to not degrade query performance made in the partitions [16], thus showing an improvement over the work of Santos and Bernardino.

Cuzzocrea et.al. ([17]) propose an architecture similar to the previous works, containing two modules: a dynamic DW module that holds a small amount of the most recent data extracted, thus adding the "near real-time" aspect to this architecture, and a static DW module storing the remaining historic data which constitutes the majority of information stored. Figure 2.11b presents this architecture. The dynamic DW module stores only the most recent data of the last day (spanning 1 day) without any constraints, indexes or materialized views, thus reducing the complexity and processing times of data insertion, which allows for the extract and load processes to run without the need for the DW to be taken offline [17]. Response time of queries made to this module are also very fast [17]. The "oldest" and majority of the data are stored in the static DW, resembling traditional DWs, and contains indexes and summary tables to speed up query answering [17]. The star schema tables present both modules is the same. Analytical queries made to the system are automatically split into subqueries and submitted to both modules by a merger component without user intervention, unlike the work of [15]. Queries are processed in parallel, and the results of each subquery from the two modules are merged and presented as if only one module existed [17].

In conclusion these systems are able to provide near real-time solutions to traditional warehouses, however, the architecture becomes increasingly complex when compared with data warehouses. These systems materialize the data, which is a complex and slow task, thus only allowing these systems to be *near* real-time, instead of *real-time*. This still makes these systems harder to make changes to the internal data organization.


Data Warehouse Schema Adaptation
Creation of replicas of each original schema table, empty of contents and without any primary key, index or any sort of constraint
Continuous Data Integration
Continuous insertion of new rows in the replica tables of the data warehouse data area
Is OLAP performance still acceptable ?
Yes
No
Data Area Packing and Reoptimization
Insertion of the contents of the replica tables to update their respective original data schema tables, together with rebuilding all indexes, materialized views, aggregates and other performance optimization data structures belonging to the data warehouse's data area.

(a) Santos et.al. Near real-time DW architecture [15].



(b) Cuzzocrea et.al near real-time DW architecture [17].

Figure 2.11: Near real-time DW architecture proposals.

2.6 Considerations

Overall, the presented polystore and multistore systems discussed here solve the problem of querying distributed and possibly heterogenous data sources in Section 2.4.2. Most of them have similarities in their architecture and implementation, however, they have downsides that motivate the development of a system that can solve such weaknesses. Table 2.1 summarizes, for most of the systems here discussed (multistore/polystore systems and near real-time DW), which of the requirements established in Section 1.3 are fulfilled by each system.

Systems such as BigDAWG or CloudMdSQL for example, force users to know details regarding the data model and the location of each data source. Executing a query across multiple data sources requires more writing, and sometimes even require users to write with the data source's native language. Apache Drill and Presto are very similar systems, as they both use the SQL Query Language and can query various data sources without any schema configuration, but both still require to know details of the internal organization of each data source (such as server names, schemas, tables). However, both these systems show potential and meet some of the requirements for this dissertation's work, and can be used as basis to implement more functionalities that can fulfill the proposed requirements.

Note however, that most of the polystore/multistore systems shown here do not have full support for analytical queries (as seen in table 2.1). Presto and Drill support analytical queries, but do not offer OLAP support, which may be acceptable for some scenarios, but not when it is necessary to deeply analyze data from multiple perspectives, in which case becomes harder without OLAP. Musq does not create any star schema. Traditional DWs that perform analytical queries use OLAP, and a focus towards shifting them towards near real-time is

Table 2.1: System Requirements

	BigDAWG	Presto	Drill	CloudMdsQL	MusQ	Santos et.al. [15]	Cuzzocrea et.al. [17]
Distributed Data Source Querying	Yes	Yes	Yes	Yes	Yes	N.a.	N.a.
Heterogeneous Data Source Querying	Yes	Yes	Yes	Yes	Yes	No (requires ETL process)	No (requires ETL process)
Provides Location Transparency	Yes	No	No	Yes	Yes	Yes	Yes
(Semi-)Automatic Global Schema Creation	N.a.	N.a.	N.a.	Yes	N.a.	N.a.	N.a.
Tools Enabling Analytical Queries Oriented Towards Decision-Making Environments	No	No	No	No	Partial	Yes	Yes
Provides a Guided Star Schema Creation	N.a.	N.a.	N.a.	N.a.	No	No	No
Deals with Physical Data Partitioning Autonomously	No	No	No	No	No	Yes	Yes
Specification of Constraints Over the Global Model	No	N.a.	N.a.	N.a.	No	Yes	Yes

already a subject of research in the literature, however, the proposed architectures are more complex and may be more difficult to manage.

The work to be presented in this dissertation, proposes a system that will be able to solve both problems: near real-time querying on heterogeneous data sources.

Chapter 3

Presto - Distributed Query Engine

Previous chapter presented several systems capable of performing queries across distributed and heterogeneous data sources. As mentioned in Section 1.3, one of these systems must be selected and experimented against in order to determine its limitations. Then, additional functionalities must be implemented in order to fulfill the established requirements in Section 1.3.

After some deliberation, Presto was the chosen tool to be tested in order to understand how to use it and it's possible limitations. This distributed query engine fulfills the main objective established in Section 1.3, as it is capable of querying distributed and heterogenous data sources in real-time. Presto supports a wide variety of data sources (more than Apache Drill at the time of writing) and its documentation is accessible. Full support with SQL is also an advantage because it makes interaction with the system easier as this is a very well known language.

Currently Presto has 2 main development versions, PrestoDB and PrestoSQL. The former is maintained by Facebook, and the latter by the community. Overall features and functionalities are similar in both versions. However, PrestoSQL supports about 29 official connectors at time of writing, while PrestoDB supports only about 23 out of those supported by PrestoSQL. For this dissertation, PrestoSQL was used for development of the platform and will now be referred to simply by Presto unless explicitly stated. The reason for this choice, besides the number of supported connectors, is the active community contributing for this version of Presto. The community is very open about it's development, as it provides support and help to new users of Presto, as well as suggestions on how to improve it. Note that, at a final stage of the development of this dissertation, PrestoSQL was renamed to Trino, however the old name will be adopted instead in this dissertation.

This chapter presents studies on the evaluation and comparison of Presto against other similar tools. It will also feature instructions on how to install, set up and use Presto (these instructions are valid to both PrestoDB and PrestoSQL), experiments performed in order to understand how to work with Presto and how Presto works, it's limitations or issues and data source compatibility.

3.1 Presto Evaluation And Comparison Studies

Performance-wise studies have been made, such as the ones found in [6, 37] that have shown Presto to be faster than Apache Drill and other distributed query engines mentioned in Section 2.4 and other SQL-on-Hadoop systems.

The experiment made in [37] used a cluster with 4 nodes, each one containing one octa-core CPU 1.80GHz, 16 GB of RAM and one SATA disk with sizes ranging between 120 and 240 gigabytes. TPC-H was used to benchmark the execution time of queries between Presto, Apache Drill, HAWQ, HIVE, SPARK, and Impala. Results show both HAWQ and Presto as the fastest systems on datasets with size of 10, 30 and 100 GB, although Presto could not complete some queries that failed on the 100 GB dataset because the volume of data could not be fit into memory [37].

The study conducted in [6] consists on evaluating Presto, Spark, Apache Drill and Hive by having each one query Hive tables created using a modified TPC-H dataset generator [1], commonly used to create and populate tables, which also provides a set of queries used to benchmark decision support systems. The infrastructure used is composed by 4 machines with commodity hardware, meaning that its specifications are not as powerful as typical clusters found on big companies or projects with high funding. The main objectives of this study is to test the effectiveness of these systems on modest hardware, used by many small to medium projects or businesses.

The main conclusions drawn from the benchmark results are that, on multi-user environments where each system executes distinct queries over the same cluster environment, "Presto is around ten times faster than Spark and around seven to eight times faster than Drill and Hive, respectively" [6]. On single user environments, Presto was also considered to be the fastest system, having completed most of the queries in less time than the other systems [6].

Another study was performed with the intent of comparing the usefulness and effectiveness of several SQL-on-Hadoop tools, such as Presto, Hive, SparkSQL, Apache HAWQ, and Drill [38]. These tools were selected because they were mentioned in more than half of the analysed published materials regarding the subject of SQL-on-Hadoop tools comparison. The study evaluated several parameters, namely:

- performance - analysis of performance test results in other published materials show Presto being the second best performing tool, after Hive [38].

- SQL standard compatibility - results of this comparison show that both Presto and HAWQ provide a SQL interface that is very close, if not identical to SQL standards.

- supported features - Presto is the second tool (first is Spark SQL) with the most supported features, such as a wider range of data source support, ability to combine data from different sources, machine learning libraries and creation of user defined functions [38].

Thus, this study concluded that Presto is the most effective tool in the previously mentioned criteria [38].

A disadvantage within Presto is that it runs and processes queries in-memory, which can fail due to lack of memory [37]. Even though the authors of [6] reported having running most of the queries with modest hardware on a 4 cluster machine. For the development of this dissertation, no more than 2 machines will be used.

[1]http://www.tpc.org/tpch/

3.2 Supported Data Sources

The data sources supported by Presto include database management systems such as Redis, Cassandra, MySQL and PostgreSQL. There is also support for distributed data stream processors, such as Kafka, distributed search engines, such as Elasticsearch, HDFS systems such as Hive, and others, such as Google Sheets and TPC-DS/TPC-H. The wrappers used by Presto that allow interoperability between Presto and each data source are designated as connectors. Each of the previously mentioned data sources have an official connector already included with Presto.

All data, regardless of the data source it resides and its data model, is mapped to relational model in order to create a common model for any data source connected to Presto and hiding details regarding the characteristics of different data sources, thus making it possible to cross-query data from distinct data sources. Only relational model is used to write the query and results are presented in a table. Anyone who develops a new connector for a data source must map all data to relational concepts (schemas, tables and columns) [39].

3.3 Installation And Setup

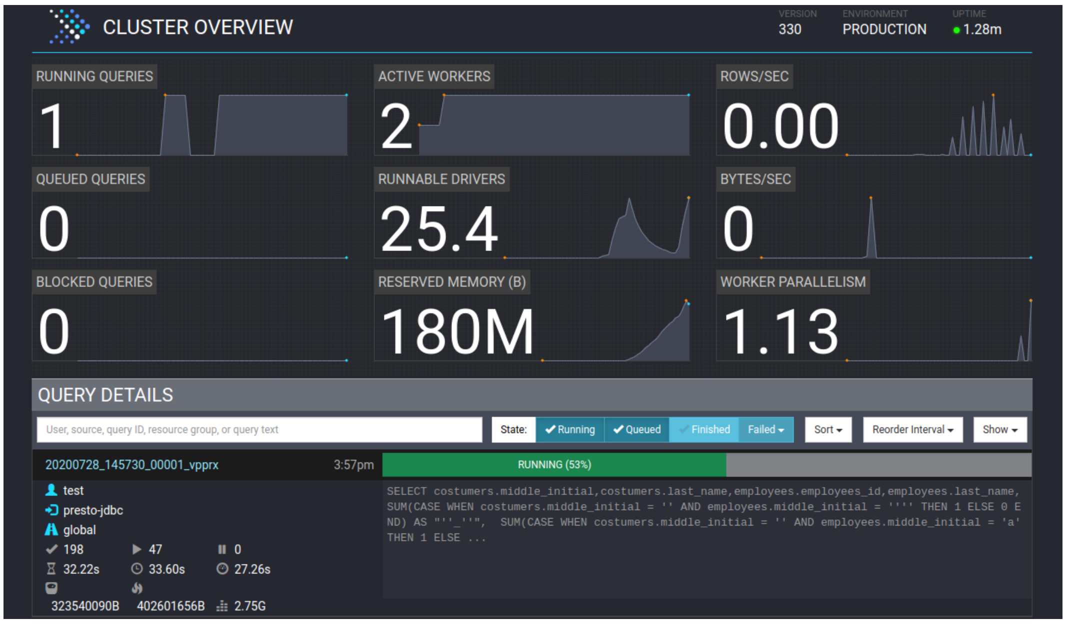
To install and use Presto two components are necessary and can be downloaded from Presto's website:

- Presto Server, which is the distributed query engine that can be configured to communicate with other Presto server installations on other machines.
- Presto Client, which is how a user sends queries to the system and receives it's results.

Both require at least a 64-bit Linux Operative System, Java Runtime Environment 8 (although 11 is recommended) and Python version 2.6. A Presto Server installation contains the core logic of Presto, including database querying, data source wrappers, designated as connectors, and distributed plans processing to handle Big Data. Multiple Presto servers can be installed in different machines and then configured such that Presto can process queries in a distributed way. A Presto client only allows users to issue queries to a Presto Server configured as Coordinator. Each Presto Server installation is a node and can be configured to be a coordinator or a worker. Coordinator nodes are responsible to receive queries, validate them and then create a distributed plan to process the query results produced by worker nodes [40]. A coordinator must exist in a Presto distributed environment. Worker nodes are responsible to fetch data from data sources using connectors, which are the wrappers that allow worker nodes to communicate and extract data [40]. Workers follow the distribution plan created by the coordinator node and change data between themselves, returning the final results to the client when the processing finishes [40]. A distributed query plan is composed by stages, each one a part of a decomposed query and executed by coordinators [40]. For example, a query that contains a union operation may split each select statement in two stages. The default limit of stages for Presto is 100, but this can be changed.

Each Presto Server displays a dashboard on localhost:8080, depicted in Figure 3.1, with information regarding the number of workers in the cluster, the queries running and statistics regarding the queries being executed. The dashboard indicates the amount of distributed workload on the CPU on more than one node, and it stays at 0 if only one node is working, and increases if the workload is being distributed. In particular, the dashboard shown in

Figure 3.1 indicates that 2 workers are detected under "active workers". and that a query is being executed by 2 workers.



20200728_145730_00001_vpprx 3:57pm RUNNING (53%)

test
presto-jdbc
global
198 47 0
32.22s 33.60s 27.26s
323540090B 402601656B 2.75G

```
SELECT costumers.middle_initial,costumers.last_name,employees.employees_id,employees.last_name,
SUM(CASE WHEN costumers.middle_initial = '' AND employees.middle_initial = '''' THEN 1 ELSE 0 E
ND) AS "''_''",  SUM(CASE WHEN costumers.middle_initial = '' AND employees.middle_initial = 'a'
THEN 1 ELSE ...
```

Figure 3.1: Presto Dashboard.

To install Presto Server, one simply has to download and unzip the files. The instructions specified at [41] allow an easy and initial configuration of Presto, such as JVM usage and node configurations. The most relevant configuration file is the "config.properties", which determines if a Presto node acts as a Coordinator or a Worker. All worker nodes must contain the URL of the Presto Coordinator node in order to be discovered so that the Coordinator can communicate with the other nodes using an embedded discovery service. To use this embedded discovery service however, all nodes should add "discovery-server.enabled=true" to this configuration file.

Directories included in the installation are:

- etc - containing the configuration files of the node, such as type of node (worker/coordinator), maximum amount of memory allowed to use, Java Virtual Machine configurations and a folder "catalog" containing all configuration files for connectors in order to establish a connection with data sources.
- bin - files to launch Presto.
- lib and plugin - Presto related libraries and connectors implementations.
- var - run and error logs.

A catalog file must be created in the directory "etc/catalog" for each connector, containing the data source name, URL, username, password, and other parameters that vary according to the data source. This file must then be named and end with ".properties", therefore creating a new catalog for Presto to use. A catalog is a reference to a data source, and Presto will use the catalog configuration file to know which data source model connector to use (MySQL, MongoDB, Hive, etc...), and to locate it in order to establish a connection with it. Presto's website contains a list with all supported connectors[2] and templates of configuration files for each data source.

Presto is a multistore system that does not create a global schema [8]. This means that in each created query, the full qualified name of an object (such as tables or columns) must be used. If a query references a table, the fully qualified name would be "catalogue.schema.table". Likewise, for a column, the fully qualified name would be "catalogue.schema.table.column". For example, by creating a configuration file for a MySQL server, and naming it "mysql.-properties", to query the table "clients" in schema "company", the nomenclature would be "mysql.company.clients".

3.4 Experiments And Limitations On Databases

In order to test Presto's capabilities, ease of use, and detect any possible issues, two main experiments were conducted with Presto, which also served as way to understand how it works and how to use it. The first experiment used a single Presto installation in one machine with 2 databases also installed: MongoDB and PostgreSQL. Both databases contain simple records that were manually inserted. This installation of Presto was configured to operate as both a coordinator and a worker. The catalogue configuration files needed to access both databases were also created.

To verify that Presto was configured correctly and that communication to both databases has been established, a simple query that selected data from both databases was executed. The query executed successfully and results from both databases were shown, thus showing that Presto can show data from distinct databases with only one query.

The second experiment used a MySQL installation on a remote machine, which also had another Presto installation configured to be a worker, while the Presto node on the local machine will act as both a worker and a coordinator. The main objective is to be able to configure both nodes (on the local machine and on the remote machine) into communicating with each other and share work loads. It was observed that the worker parallelism (as seen in the dashboard depicted in Figure 3.1) only started to increase when heavy queries were being executed, as Presto only distributes the workload when the I/O begins to bottleneck the machine.

From these experiments, it was also noticed that Presto maps each data type into its own set of supported data types[3], such as money being mapped to double, or datetime being mapped to timestamp.

However, some limitations were found within Presto after some usage with Presto, not only after the previous experiments, but in the development of the work presented in this dissertation. The main limitations found are as follows:

- First, as Presto was being used to query MongoDB, it was noticed that fields of a collection were not being correctly updated. In a given collection, Presto will map each document's fields into relational model, however this schema representation of Presto only considers the first inserted element in a given collection. Any documents inserted afterwards with different fields are not considered in Presto's relational schema. For

[2]https://prestosql.io/docs/current/connector.html

[3]https://prestosql.io/docs/current/language/types.html

instance, a document with fields named "A" and "B" that is inserted first in a MongoDB collection will be correctly represented in Presto's schema, but a second inserted field with fields "A", "B" and "C" will not be correctly represented as Presto will not map the "C" field to its relational representation.

- A second observation is that, for some data sources and on older versions, Presto executes only simple queries to each data source (select and filters), but later versions of Presto, and as long as the connector supports it (PostgreSQL for example), also push simple aggregate functions, such as count and sum, to the data sources [42]. Joins, sorts and aggregation operations that cannot be pushed to data sources are executed by Presto itself using the workers [12, 42, 39]. The advantage of pushing down to the data sources as much operations as possible, such as filters and aggregations, is that it can reduce network traffic, as well as reduce the load on Presto's workers, thus improving the overall query performance [42]. For other data sources, such as Cassandra or Hive, Presto tries to be as efficient as possible by using filters to read only the necessary partitions, rows or blocks depending on the filter applied [39].

- Presto is sometimes unable to query databases, tables or columns whose name contains at least one uppercase letter. For example, when querying the table 'MyTable' located in PostgreSQL, Presto will throw the following error: 'Table 'mytable' does not exist', because Presto converts all names of databases, schemas, tables and columns to lowercase. This is a know limitation with open issues on Presto's Github [4]. Presto might however be able to identify that an uppercase database/table/column exists in a "show" command, but is unable to query that table. This also depends on the connector used. For example, on MySQL, it is possible to query tables or columns containing uppercase letters, while in PostgreSQL it is not. This is a big issue and unless it gets fixed, the only way to ensure Presto can query a specific table or column is to not add any uppercase characters.

- The schema "information_schema", present in most relational databases, contains metadata such as primary keys and foreign keys, a list of all tables and its columns, and other metadata useful to obtain information regarding the data organization. However, Presto does not show and cannot access all of the tables in this schema [5]. The main issue is that, in most tables, there is an attribute with a data type that Presto does not support (sqlidentifier). Creating a view containing the intended query while also casting the attributes to a supported data type (such as varchar) fixes the issue. This means that the administrator of the database would have to create such view in each of their databases. This dissertation's code repository [43] presents a set of scripts ready for MySQL and PostgreSQL that create such views to query both primary key and foreign key information.

- To add an additional data source, a new catalogue configuration file containing correct configuration to connect to the data source is required. If Presto is running, the newly added catalogue is not visible to Presto, and therefore a restart is required [6].

[4]https://github.com/prestodb/presto/issues/10288
[5]https://github.com/prestodb/presto/issues/11509
[6]https://github.com/prestosql/presto/issues/2110

3.5 Experiments And Limitations On Files

A non official connector named *flex*[7] exists for Presto that allows the execution of queries over local or remote files, namely csv, excel, tsv, txt and raw. However, it is not included with a Presto installation and must be manually added by downloading the project, generating the jars of said connector, and placing them on the *plugin* directory of Presto. A catalogue configuration file file must also be added, but this file contains only a single line: "connector.name = flex" and can be used for any file, in other words, a catalogue configuration file does not need to be created for each new file intended to be queried. However, querying raw files using this connector poses some minor differences when comparing with other data sources seen previously, which were mostly data storage systems. These differences justify the creation of this section to discuss about this connector and its possible usefulness and implications.

First, the structure to query files using the flex connector takes the following syntax: flex_catalog.file_type."path_to_file" (note the double quotes used to escape characters that would otherwise create invalid syntax errors on the SQL query if not escaped), which follows the usual naming convection used in Presto queries, with the file type being the schema, and the path to the file being the equivalent of a table. As a concrete example on how to use the connector, and assuming the catalogue file is named "flex.properties", the query 'select * from flex.csv."file:///home/user/sales.csv"' will list the contents of the csv file "sales.csv". The "file://" at the beginning of the file location means that it is a local file, otherwise it should start as "https://".

The structure of the file is presented in a relational format. For example, a csv file will have the first line of the columns being treated as a relational table's attribute. The same happens with tsv and excel files, but the contents of raw and txt files are treated as a single column table with one row.

Being able to quickly perform analytical queries to raw files that may reside in a data lake or on files belonging in a dataset without having to import data to a data storage system only improves the usefulness and overall versatility of the developed tool.

A considerable limitation present in this connector is that, like some of the other connectors, it cannot query files whose name or location contain an uppercase letter. This means that, in order to query a certain file, it must be moved and/or edited such that its path and file name does not contain any uppercase letters. An effort was made in order to fix this issue but unsuccessfully, as the root of the problem takes place within Presto source code.

Another limitation this connector introduces is that all columns in the file have the varchar datatype, regardless of the actual value of the data. If a column contains only numbers, this connector will not automatically convert the column to an integer. This can introduce some limitations when working with this data.

3.6 Considerations

After familiarizing and experimenting with Presto, one may assume that it can fulfill all objectives previously established in Section 1.3. Queries using data from two or more distinct data sources can be executed independently of the data source's location or data organization. In addition, Presto does not require any schema creation, which makes the configuration process even easier. However, this abstraction between data sources and their organization is

[7] https://github.com/ebyhr/presto-flex

not sufficient, because users require knowledge regarding the internal organization of each data source in order to reference a column or a table in Presto because the full qualified name of each element is required (a nomenclature in the form "catalog.schema.table.column"). In other words, Presto does not provide location transparency, or even fragmentation transparency. This is one of the trade-offs of not creating a global schema. To create an abstraction layer capable of identifying each object without user indication of the details of each data source, an additional layer on top of Presto is required.

Simply using Presto to query data sources is possible, but forces users to have to know details regarding the underlying data sources, which may not be ideal for some cases, such as less experienced users. Presto does not support any creation and maintenance of multidimensional schemas, which are very common for analytical queries. These reasons are enough to justify the creation of an additional layer on top of Presto. This layer can also add many other features that Presto alone cannot, such as configuration of data sources and creation of global schemas, multidimensional schemas. Additional features, such as storing previously executed queries or saving queries results would enhance the system's usefulness and help data analysts using such system.

Because the user would interact with this layer and not Presto, this layer should also make transparent Presto related configurations, such as automated creation of properties files to add a new data source with user inserted information, restart upon adding new data sources and management of Presto related errors.

Chapter 4

Proposed Solution

Previous chapter presented some details about Presto's features and weaknesses, while establishing the importance of a layer on top of Presto to create a transparency between the data source's organization and their characteristics, as well as analytical query support, among other useful features that Presto alone cannot offer. This layer is named Easy Big Data Integration (EasyBDI) and is the main focus of this dissertation.

This chapter explains on a technical level the development of EasyBDI, from it's architecture, integration with Presto, schema creation and management, and query execution. Each of these topics will contain a discussion of the adopted solutions, as well as problems and challenges found throughout development, including compromises made and possible alternative solutions considered.

There are multiple challenges to overcome in order to build such a system, as discussed in Chapter 2. The main challenges for the development of this system are:

1. Integration with Presto, such that it is possible to control it from EasyBDI. It must be possible to submit queries and other useful commands to Presto and receive and process the answers to those commands.

2. Construction of a local schema view from user selected data sources. This view will provide a simplified view of each data source.

3. Construction of a global schema that turns the overall data source distribution and heterogeneity transparent to the user

4. Creation of star schemas and mapping a global schema to a star schema.

5. Query transformation from global schema to local schema and local schema to global schema.

6. Creation of interfaces for the configuration of the data sources, global schema, cube creation and execution of analytical queries.

7. Creation of methods to convert queries from the global schema to the local schema, in order to submit to Presto, which will then communicate directly with appropriate data sources in the local schema. Results must also be presented accordingly with the global schema.

4.1 Used Technologies

Java 8 was the main programming language selected and Maven was used for the main project in order to manage dependencies. Java can communicate with Presto through a JDBC interface in order to submit commands and receive results. This interface's methods and data structures are very similar to those used with other databases.

Java Swing is used to create a desktop based application containing a User Interface (UI). It is lightweight and platform independent, meaning it can run on any computer as long as Java Runtime Enviroment (JRE) is installed. Creating an interface using Java's UI libraries provides an easier way of integrating the back end logic of the application with the front end logic.

To store schema information and multidimensional mappings (hereinafter referred to as metadata) persistently a storage solution is needed. There were several options, such as a relational database or java serialization. Serializable files, however, are harder to organize data and to avoid data redundancy, which can also affect query times negatively. Therefore a relational database was used. There are multiple database systems, such as PostgresSQL, but in the end, SQLite was chosen because it requires less computational resources than a full fledged database system. If implemented to store data source, global schema and star mapping information, it is not expected that the SQLite database has to store enormous amounts of data, as it is unlikely that a real company has hundreds of databases, or millions of tables. Concurrency is also not considered, as only one user operates the system at a time.

4.2 Architecture

The architecture of EasyBDI is illustrated in Figure 4.1. Modules are used for a better identification of the different elements that integrate the overall system. The module at the bottom of Figure 4.1 represents Presto as the distributed query engine, used by EasyBDI to query the different data sources using the queries generated by EasyBDI. Presto must be installed as previously explained in Chapter 3. All other modules are integrated in a single Java Maven project and the source code can be found on Github [43].

Note that Presto could be replaced by other query engine that accepts SQL queries and communications using JDBC. This change could also imply additional implementations on EasyBDI to configure new data sources to the query engine.

An overview of each layers and modules is given next and following sections will discuss in more detail each module's responsibilities, how they work, and challenges faced during development as well as their respective solutions and issues (if any).

API Communication Handler - Contains logic to submit commands and process results for external systems. Two types of interfaces exist, one for Presto, and the other for SQLite, being the latter used to store schema metadata.

- **Distributed Query Execution Interface -** Receives queries created by the *Query Execution Manager* and uses the JDBC interface to send the query to Presto. It is also used for other commands besides normal queries, and it is also responsible for receiving and parsing the results from Presto.

- **Metadata Storage Interface -** this module is responsible for the persistent storage of metadata of all schemas: local schema view, global schema, and multidimensional

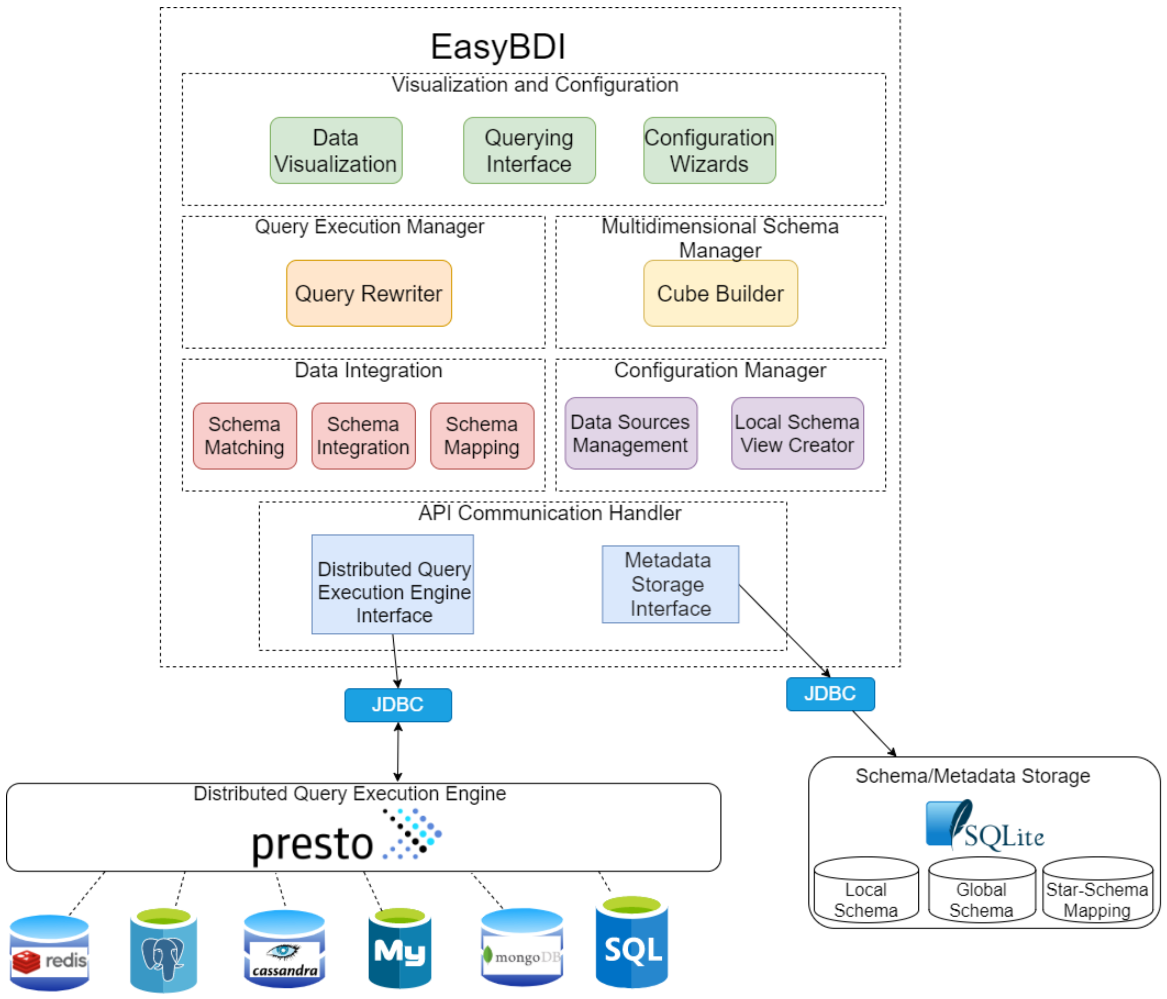


Figure 4.1: EasyBDI architecture and main modules.

schema. These schemas are stored in SQLite in a relational representation. JDBC is used to access and communicate with the SQLite database to perform operations such as inserts or selects and other metadata maintenance. It is possible to replace SQLite by other solution as long as it supports JDBC communication and relational schema models, otherwise, additional code would have to be written.

Configuration Manager - Handles data source related information, namely user inserted information for data sources, and local schema creation/management

- **Data Sources Management -** Handles information regarding each connected data source and generates a configuration file in order for the Query Execution Engine to use and to be able to access the data source.
- **Local Schema View Creator -** Creates a local schema view by querying each data source (through Presto) in order to retrieve its schema information.

Data Integration - this layer is responsible for the creation of a Global Schema, which is a unified and simplified high level representation of data from the local schema. Creating a global schema using data integration techniques is not an easy task, but this is not the main focus of this work. For this reason, a simplified set of metrics (from the ones explained in Section 2.3) are used in order to create a proposed global schema, which can then be fine tuned by users through a user interface in the *Configuration Wizards* module.

- **Schema Matching -** Applies schema matching techniques to identify and match similar tables.
- **Schema Integration -** Applies schema integration techniques to group similar local tables and create a global table from those tables.
- **Schema Mapping -** Detects and assigns a mapping between each global table and its corresponding local table(s).

Multidimensional Schema Manager - This manager is responsible to maintain the multidimensional structures by creating mappings between tables in the global schema to a multidimensional one. These mappings include the indication of which global table is the facts table, which columns are measures and which global tables are dimensions. Only star schemas can be created. The *Cube Builder* maps tables of a global schema, into a star schema by assigning which tables are dimensions and the facts table (including measure attributes).

Query Execution Manager - This layer is composed by the *Query Rewriter* which is responsible to generate ANSI SQL queries containing the entities defined in the star schemas and in the global schema from the user input created in the *Querying Interface.* The generated query will then be modified to include the entities defined in the local schema. This query is submitted to Presto, which distributes the query by the appropriate underlying data source systems in the local schema. It also integrates the results of each local query into a single output, presented accordingly to the global schema and star schema known by the users who analyze the data in the *Data Visualization.*

Visualization - All logic related to user interface and information visualization, as well as user interaction with the platform is located in this layer. Some of the screens are used for data source or schema configurations. Other screens are for querying and visualizing results. These screens can be used by subject-expert users and data administrators. For instance, in the global schema editor wizard, users can display the global schema and edit it (e.g. add or remove mappings between local and global entities).

- **Data Visualization -** Shows information such as query logs and query results.
- **Querying Interface -** Allows users to drag and drop elements of the star schema to certain areas in order to create queries.
- **Configuration Wizards -** Wizard-like windows that help users in the configuration process of data sources, global schema correction, and star schema creation.

Different interface windows interact with different modules of the system. For example, the global schema editor window receives data from the Data Integration layer and saves the global schema defined by the user in the Schema/Metadata Storage.

4.3 Integration With Presto

An essential part for the proposed system is to be able to query distributed and heterogenous data sources. As seen in Chapter 3, Presto fulfills these requirements. The main usage of Presto is to perform analytical queries over several data sources and to gather metadata information regarding all data sources involved in the local schema specified by the user in order to build the local schema view, which will be saved persistently in SQLite.

In order to do this, queries and other useful commands will be issued to Presto by JDBC, which allows the reception of command and query results in Java.

Before Presto can query a data source, a catalogue configuration file must be created as discussed in Section 3.3. These configuration files contain different parameters depending on the data source. This means that, for each data source connected through EasyBDI, the correct configuration file must be generated in the correct directory. In addition, Presto must be restarted automatically so that these configuration files are detected and enable Presto to connect to the data sources. Such actions are performed by EasyBDI in a transparent way, thus ensuring that the user does not have to deal with Presto related logic when adding new data sources, and instead rely exclusively on EasyBDI to create the configuration file and other Presto related configurations.

To configure a new data source, EasyBDI must provide a UI in which the user inserts data source information, such as URL, model and if necessary, its credentials. Each configuration file has different parameter names depending on the data source. A basic configuration file for each available data source is available in Presto's website [1] and this model is used to generate a configuration file using user input. However, due to data source heterogeneity (core functionality, data model and other aspects), more advanced configuration parameters in its connector's configuration files exist, but for now, only the default configuration parameters necessary to establish connection are considered. Further work could be made to allow users to fill in more advanced settings for each specific data source if they wish to do so.

Other data source related details must also be taken into account, such as MySQL's similarity between schema and database. Creating a new schema or database in MySQL is basically the same operation. Multiple databases can be stored in one MySQL server and a database cannot be accessed individually, unlike PostgreSQL that maintains database and schema as separate concepts and, therefore, it is possible to access one specific database in a PostgreSQL server. In a MySQL connector configuration file, if a database is specified in the server's URL (in the form http://my_server/database_name) Presto throws an exception.

Database-specific configuration details are however dealt with on the UI, which shows different fields depending on the data source selected and take such differences into consideration when generating a configuration file. More details are discussed in Section 5.3.2.

EasyBDI can start, restart or even stop a Presto coordinator node, but cannot install or manage a cluster of Presto nodes. An administrator must be responsible to install and manage the cluster, and must also specify the path of the coordinator node to EasyBDI using a field on the interface. The Presto Coordinator node must be installed on the same machine as EasyBDI.

[1] https://prestosql.io/docs/current/connector.html

4.4 Data Integration And Schema Creation

This section will cover the adopted solutions to create each of the three schemas: local, global and star schemas. In EasyBDI, the users intervene to configure each of the schemas, but this section will mainly focus on how the schemas are organized, created and managed by EasyBDI. For global schema, data integration techniques need to be applied to create an initial proposal of a global schema, which is also covered in this section. EasyBDI offers the option to modify the proposed schema in order to correct any possible mistakes.

4.4.1 Local Schema View

The local schema view is a representation of the structure of the underlying data sources, and is built using the different *show* commands available in Presto in order to reveal the contents of each data source, all mapped to a relational data model. This offers a simplified view of the underlying data organization, while providing location transparency and hiding fined grained details of each data source, including the data model and query language. The local schema view is therefore a higher level representation of the local schema generated by Presto. They should not be confused, and throughout this dissertation, local schema is used to refer to the multiple physical data storage systems, and local schema view is the simplified representation created by EasyBDI.

To create the local schema view, the system starts by issuing to Presto the command *Show schema* to reveal all schemas of a particular data source. Note that the concept of schema may differ between different relational databases, as they may organize tables in different ways. For example, in PostgreSQL, *Show schema* would show all schemas created in a given database, but in MySQL and MongoDB it would reveal all databases in the server. *Show tables* returns all table names from a certain schema of a data source. Again, a table might be a different structure depending of the data source, like a collection in MongoDB, but it is mapped to a relational table in Presto. Finally, *Show columns* returns all column names and data type of a given table. However, this last command does not reveal any information regarding constraints on relational database columns. This information is important for later stages, namely to create global schemas and star schemas. The user can add those constraints manually on global schema edition later on, but it would be easier for the user if those constraints were already present in the local schema view.

To get constraint information of each attribute in relational databases, the schema "information_schema" is used, as mentioned in Section 3.4. The interest in this schema lies in the table "table_constraints", containing information of constraints on all columns. However, due to the problem encountered and explained in Section 3.4 regarding this table containing a column whose data type is incompatible, it is not possible to directly query this table using Presto, because the table is not visible to Presto. Instead, a view that converts all columns to a Presto supported data type is created and Presto queries this view instead, thus making it possible to obtain constraint information.

For data sources without constraints, which are non relational databases, constraints are not stored, but users will be able to create constraints (foreign keys or primary keys) in the global schema tuning, later on the configuration process (see Section 5.3.2 for more details).

The local schema view metadata is stored in the SQLite Schema/Metadata Storage module. Figure 4.2 represents the organization of the tables in SQLite used to store this information. Tables exist to store information regarding each data source, table and column. The table *db_ data* stores the information needed to access a data source, namely URL, name and credentials, as well as the model of the data source. This information is obtained through user input. Table *table_ data* stores a table name, the data source it belongs to, and the schema name. The schema name is only stored because Presto needs to know the schema a table belongs to. Finally, table *columns_ data* stores a column's name, data type and the table it belongs to. Two fields are used for constraints: a flag indicating if this is a primary key, and a another field for foreign key constraint, which is empty if a column is not a foreign key, otherwise contains the full name of the column it references in the form "<data source name>.<schema name>.<table name>.<column name>".

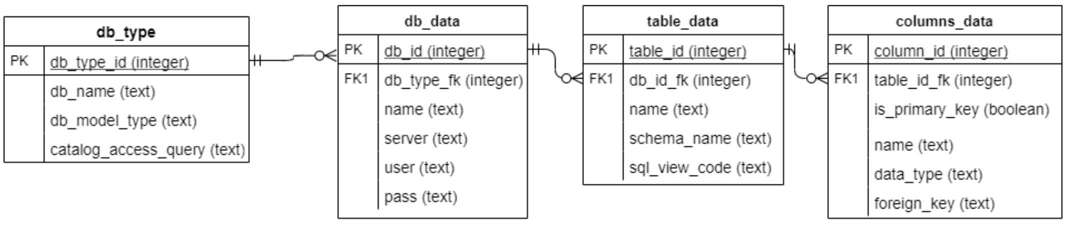


Figure 4.2: Database Diagram of the table organization used in SQLite to store information regarding the local schemas.

Table "table_data" in Figure 4.2 has a column named "sql_view_code" containing an optional user created SQL code that runs on Presto and whose result gives origin to this local table. This feature is fully explained in Section 5.3.2.

The above steps can be summarized in a single process used to build the local schema view, which starts by iterating each data source that is intended to be part of the local schema, and for each one use the *show* command to extract schema, table and column information (or the equivalent concept depending on the data source) and store this information in the tables presented in Figure 4.2. If the data source is a relational database, then constraint information is extracted and this information is added for each attribute.

4.4.2 Creation Of The Global Schema

Creating a global schema from the local schema view involves several steps and is usually referred in the literature as *Data Integration*, as discussed in Section 2.1. The global schema aggregates identical objects found in the local schema into one object, enabling easier analysis of the data residing over the data sources. In this case, the objects on both schemas are tables, because Presto converts all objects into tables regardless of the data source. Schema matching, schema integration and schema mapping are steps needed to build the global schema in order to provide logical data integration. Schema matching is applied to establish correspondences between semantically similar tables found in the local schema, here referred as local tables, in order to group those tables into one table in the global schema. Tables belonging in the global schema are named global tables. A process of *schema integration* is also applied to integrate all similar tables into one global table after the detection of such similarity by schema matching. Mappings are then assigned on the *schema mapping* phase, which allows the creation of queries that convert data from the global schema to the local schema.

Note that an automated way to perform schema matching and schema integration could be skipped, and have the user manually indicating the matches from the local to the global schema. However, manually creating a schema matching from scratch can be an overwhelming task on cases when there is a high number of data sources and tables due to high number of correspondences needed to be manually created by a data source administrator. On the other hand, there is no silver bullet that can be used to create a fully automated and high quality schema matching. Therefore, an automated approach is used, where the system performs schema matching and schema integration algorithms which generate global tables and matches between the global tables and local schemas, which can then be adjusted and complemented by the users.

The process of implementing schema matching and schema integration algorithms can become a complex task, as these fields have been the target of many research throughout the years, with many algorithms available for both processes. As the focus of this dissertation is not only data integration, a simpler approach to both schema matching and schema integration was implemented. Hence, the automatically generated global schema may be only partially correct. Anyway, it would be easier for the user to edit and complete the proposed global schema than creating a global schema from scratch.

Schema Matching

The implemented strategy to perform schema matching is illustrated in Figure 4.3. This figure will be used as a means to explain step by step the implemented algorithm.

The algorithm starts by matching tables with similar names using Levenshtein distance, by iterating each pair of tables (symmetric pairs are not considered, meaning table A and table B are compared, but table B is not compared again with table A). Tables are considered a match if their name similarity is above a defined threshold, currently set as 0.6. In Figure 4.3, step 1 shows 4 tables in the local schema view, and step 2 shows the result of similarity of names between each pair of tables. Only tables "employees" and "employees_info" match, as their name similarity is above 0.6, but table "inventory" has no matches.

Schema matching is also performed between pairs of tables, but only during the process of schema integration of tables and not right after the matching between two local tables. More details regarding schema integration are discussed in the next section.

The similarity between columns is determined by column name, data type similarity and primary key information. A formula is used to obtain a similarity value using all previous similarities, and assuming that both name and data type similarity are equally important to determine column similarity, but primary key information is slightly less important. Percentage of important of each similarity is therefore 40% for both data type and name similarity and 20% for primary key similarity. The formula is shown in Formula 4.1:

$$data\_type\_sim \times 0.4 + name\_sim \times 0.4 + pk\_sim \times 0.2 \tag{4.1}$$

The order in which the similarities are calculated is: data type similarity, name similarity, primary key similarity. Data type similarity is measured in the following way:

- Data types in both columns are the same - similarity = 1.
- Data types are different but belong to the same category (example, double and integer, both numeric) - similarity = 0.8.
- Data types are different and belong to categories where conversion is sometimes possible (example, string and numeric conversions are not always possible) - similarity = 0.4.

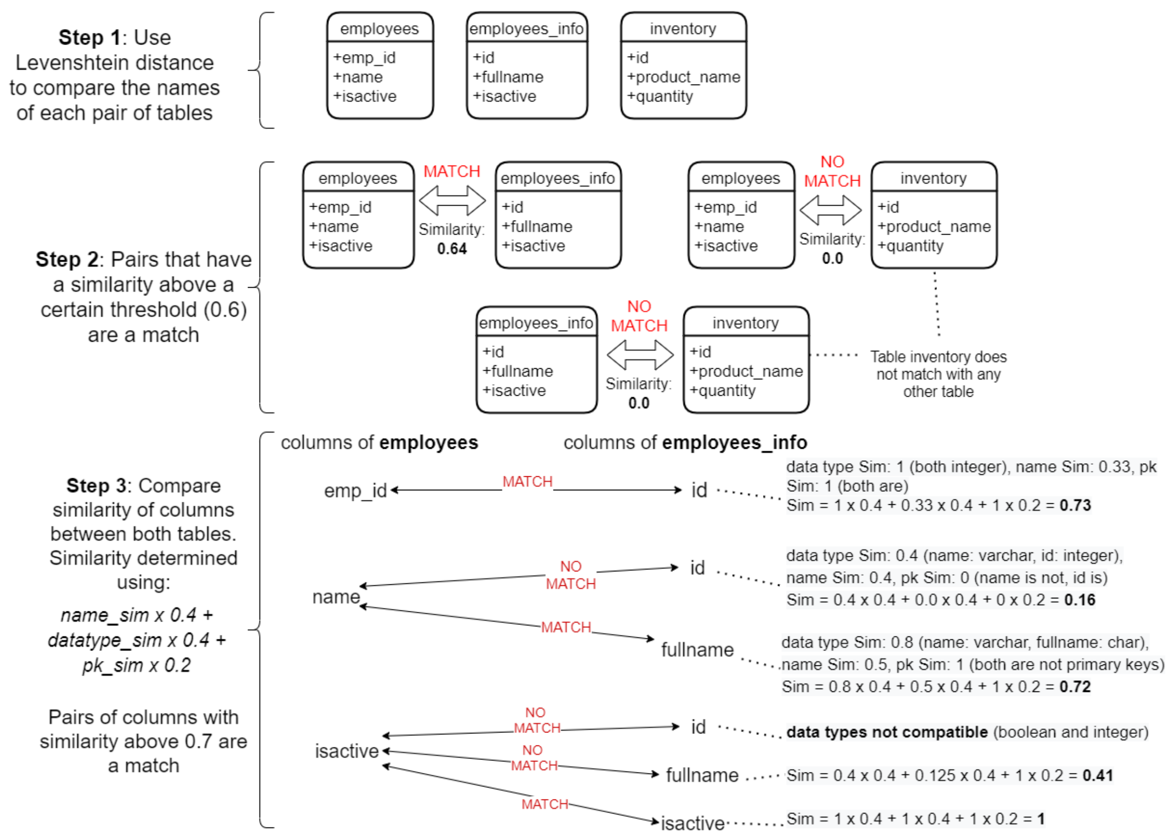

Step 1: Use Levenshtein distance to compare the names of each pair of tables
employees
+emp_id
+name
+isactive
employees_info
+id
+fullname
+isactive
inventory
+id
+product_name
+quantity
Step 2: Pairs that have a similarity above a certain threshold (0.6) are a match
MATCH
Similarity: **0.64**
NO MATCH
Similarity: **0.0**
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**
emp_id
MATCH
id
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**
name
NO MATCH
id
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**
MATCH
fullname
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**
isactive
NO MATCH
id
data types not compatible (boolean and integer)
NO MATCH
fullname
Sim = 0.4 x 0.4 + 0.125 x 0.4 + 1 x 0.2 = **0.41**
MATCH
isactive
Sim = 1 x 0.4 + 1 x 0.4 + 1 x 0.2 = **1**

Figure 4.3: Illustration with an example of the implemented schema match solution.

- Data types are different and belong to categories where conversions are never possible (example, conversions between date and numeric is not possible in Presto) - similarity = 0.0, columns are not a match.

The data types considered here are the ones supported by Presto [2]. For simplicity, only the most common categories of data types supported in Presto were considered, namely the ones show in Table 4.1.

After the calculation of data type similarity, the other metrics are calculated, unless the data types have 0 similarity, in which case the process stops. It is assumed that if the data type between both columns is not compatible, then it is very unlikely that the data in each column has the same semantic meaning, and therefore it does not make sense to calculate name and primary key similarity.

Column name similarity is once again measured using Levenshtein distance. Primary key similarity is equal to 1.0 if both columns are a primary key or if both are not a primary key. If one is a primary key and the other is not, this similarity is 0.

Figure 4.3 portrays on step 3 an example of column matching between the previously matching tables "employees" and "employees_info". Each column of table "employees" is compared for similarity against the columns of table "employees_info". Starting with column "emp_id", similarity is first tested on the first column of the other table, "id". Similarity score is above the defined threshold, 0.7, therefore these columns match and no comparison is

[2] https://prestosql.io/docs/current/language/types.html

Table 4.1: Presto supported data types and their domains used for data type similarity in column schema matching.

Domain	Data types
Numeric	tinyint, smallint, integer, bigint, real, double, decimal
String	char, varchar Json, varbinary
Boolean	boolean
Date and Time	date, time, time with time zone, timestamp, timestamp with time zone, interval year to month, interval day to second

made between column "emp_id" and the remaining columns of the other table. Moving on to the next column of table "employees", "name", is compared with "id" and its similarity score is below 0.7, thus moving to the next column, "full_name", which is a match, therefore the next column of the first table, "isactive" is compared. This column fails to match with "id" due to low similarity score, and also fails to match with "full_name" because their data types (boolean and integer) are not convertible, thus moving to the next column in the second table. In the end "isactive" match with the other column with the same name. Note that in this example, all columns had a match, but cases may exist in which columns have no match. Cases may also exist where tables have no matched columns between them.

The implemented approach can be improved in several ways. For example, Levenshtein distance to test name similarity alone can fail in certain cases (such as "emp" and "employee_info", the score is 0.23, which is low) and an additional technique could be used, such as *n*-gram. However, this approach still does not take into account synonyms, hypernyms and homonyms, which, as mentioned in Section 2.3.1, require a dictionary. Such dictionaries should include abreviation, such as "emp" being equal to "employee". These additions could in theory yield more accurate results. The implemented algorithm provides a good starting point to schema match and can be improved in the future.

Schema Integration

This step integrates the matching tables to form a single global table. To begin, a list of matching tables is created, and each list will form a global table. To form this list, it is detected a "chain" of matches between local tables. It is not expected that one table matches with all the remaining tables in this list. This means that even if table "A" and "D" are not matched, they will still be present on this list because table "A" matches "B", table "D" matches "C," and "B" and "C" match together, for example.

The next step is to perform schema integration. To do this, a binary integration strategy is used to merge all syntactically similar columns into one global column, which is illustrated in Figure 2.5a. The integration process starts by the first table in the list, and the name of the global table is the name of this first table. The first and second tables on the list are "merged" by creating columns on the global table. After two local tables are merged and a new global table is created, this global table and the next local table in the list are "merged". Again, Figure 2.5a shows an example of this, were local tables "A" and "B" are merged in global table "AB" and afterwards this global table is merged with the next local table, "C", and so on.

At each table merge, column schema matching between both tables is performed as described in the previous section. Afterwards, columns must be created for each pair of column matches, and for each column without a match.

For each pair of column matches, a new column is created with name equal to the first table on the list, and datatype equal to both columns 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 a greater diversity of data to be represented without causing any error. For example, double is more generic than integer, and varchar is more generic than integer.

Each global column with no match is "copied" to the global table. As an example, consider once again Figure 2.5a. When table "A" and "B" are merged, the resulting table maintains columns that match between "A" and "B", namely "X", "Y" and "Z". However, column "W" in table "B" has no matches, therefore it is copied to table "AB".

The previous steps are repeated until all tables have been integrated into a single global table. As a remark, note that step 3 in Figure 4.3 that shows column schema matching is considering the easiest scenario, which is the merge of two local table into one global table, mainly because the focus of this figure is on the explanation of schema matching. This figure may give the idea that column matching is performed for each pair of matching tables, but this is not correct, as schema matching of columns is applied for each iteration of the schema integration.

This algorithm has a few limitations:

- If table A and B, both with 4 attributes have only one matching attribute, those attributes are merged in a new one, but the remaining 3 columns from each table that have no matches are also added to the global table, thus creating a global table with 7 attributes which in some cases may result in a table containing attributes with distinct semantic meanings. However, if tables B or A have other column matches with other tables, those columns will also be added as columns in the global table. This could result in various false semantically related columns on the same global table.
- Primary key information is maintained, but foreign key information is not, meaning that global tables will not have foreign keys. The user must add those manually.

Schema Mapping

Each column in a global table holds the information regarding the column or columns in the local tables that correspond to that column. Schema mapping will assign a mapping between each correspondence between columns in the global table and columns in local tables to enable performing queries on the local schema, while converting said results to the global schema.

Schema mappings created in this work are different from those seen in the literature. While many create formulas to specify rules between global and local schemas [10, 44], here, more generic mappings are created and will fall into one of 3 categories depending on the partitioning of the data sources:

- Simple Mapping (or 1-to-1 Mapping) - One global table has only one local tables as a correspondence. Each attribute of the global table will have one correspondence to one attribute of the local table. The global table may only have some of the attributes of the local table.
- Horizontal Mapping - One global table has more than 1 correspondence to local schema tables, which are horizontally partitioned.
- Vertical Mapping - One global table has more than 1 correspondence to local schema tables, which are vertically partitioned.

More details on data partitioning and replication can be found in Section 2.2.1.

Finding the type of mapping involves running algorithms against each global table and their correspondences. The method to determine the mapping between a global table and its corresponding local table(s) is presented in Algorithm 1. First, if each global column has exactly one correspondence to columns belonging to the same local table, then there is a simple mapping (or 1 to 1 mapping), as no partitioning exists in this case. However it is not required that all columns in the local table exist in the global table.

If multiple local tables correspond to one global table, then it is necessary to determine which partitioning exists in the local tables.

To detect horizontal partitioning, the following criteria is used:

- a global table must correspond to more than one local table.
- the number of columns in each local table and in the global table must be equal.
- each of those local columns must have the same name and data type when comparing with the respective global columns (note that it is possible to change a global column's name and data type while correcting the global schema later in the configuration process, as discussed in Section 5.3.2, however the system always considers the original name and data type of the column to determine the mapping type even if the original column's name or data type is changed).

The function *isHorizontal()* in Algorithm 1 checks, for each column, if the previous conditions are met. If they do, horizontal mapping is assigned.

For vertical mapping, one or several tables must have a foreign key referencing a primary key of another table, which should be the original table that was partitioned into other table. This step is implemented by function *isVertical()*, which tries to identify foreign key references in order to determine if there is or not vertical partitioning. First, line 13 obtains the list of local tables containing columns corresponding to the primary key column in global table, and line 14 obtains the local tables containing columns that are foreign keys and primary keys. If there are no tables, than vertical mapping is not possible. Line 18 extracts the column that is primary key but not foreign key(s). Finally, lines 19-26 verify that each of the other primary key columns that also are foreign keys all reference the primary key of the original table. If one of those foreign keys does not reference such column, the function returns false and vertical mapping will not be assigned to the global table.

In the end, if neither simple, vertical or horizontal mapping is detected, then undefined mapping is assigned. Users must correct the global table in this case. Next section will explain user intervention to correct the global schema.

Correcting The Proposed Global Schema

After the automatic generation of the global schema, the user is requested to correct the global schema using the interface to interact with the system. How exactly that interaction is done will be the topic of discussion in Section 5.3.2, as this section covers only what will be done with the changes the user creates using UI. While editing the global schema, the user must also create necessary constraints, namely primary keys and foreign keys in global tables where such constraints may not exist. The user must create the constraints such that it is possible to create a simplified star schema (discussed in the next section), by creating foreign keys and primary keys on the tables that are intended to be the dimensions or the facts table.

When the user confirms the global schema, it is stored in the Schema/Metadata Storage module. Figure 4.4 shows the tables in SQLite used to store the local schema view and also the tables that store the global schema. Table *table_global* stores all global tables and table *column_global* all global columns, along with the reference of the global table it belongs to. The *correspondences* table associates a local column to a global column as well as the mapping type defined in schema mapping stage. All global columns belonging to the same global table should have the same type of mapping in each of their correspondences with local columns. This information will be used to generate queries that gather information from all local columns that match to a specific global column selected by the user in a query, but more details will be given in Section 4.5. Note that no association is made between global tables and local tables, but only between global columns and local columns.

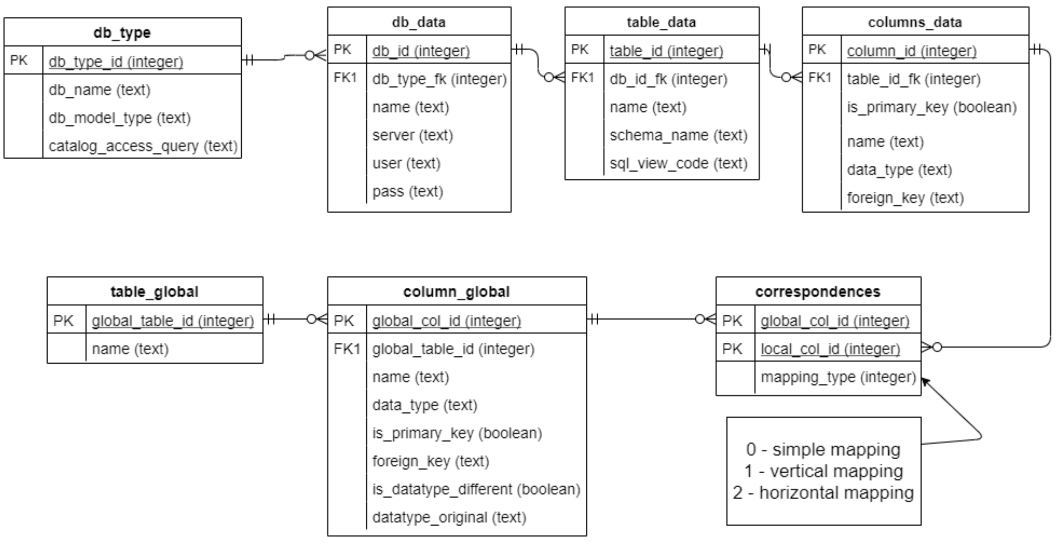


Figure 4.4: Database Diagram of the SQLite Database to store information regarding the global schemas, including correspondences from the local schema to global schema.

Note that table "column_global" in Figure 4.4, that stores global column information contains 2 attributes that may be confusing: "is_datatype_different" and "datatype_original". These attributes are mostly used to identify if the global column had its datatype changed by the user, which is a feature available in EasyBDI and fully explained in Section 5.3.2.

4.4.3 Multidimensional Schema

The previously defined tables on the global schema can be used in one or more star schemas. This mapping consists on identifying which global tables are the dimensions and which is the facts table. In the latter, the attributes that will be used as measures (if any) should also be identified. 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 tables to a star schema, which will be showed and explained in Section 5.3.3. It is important that foreign key constraints are correctly defined between the facts table and the dimensions, a step that is performed on the global schema creation.

Figure 4.5 shows the same tables used to store the local and global schema information seen in Figure 4.4 but also the tables used to store star schemas, seen at the bottom of the figure. Table *multidimensional_ table* contains a global table and a flag which indicates if the reference table is a dimension or a facts table. Each column from each table in the star schema is also stored as a reference in table *multidimensional_ column*, which also contains a flag to indicate if the given column is a measure or not. Of course, measures can only be attributes in the facts table. Both tables *multidimensional_ table* and *multidimensional_ column* belong in a specific star schema, hence they contain a reference to the table *star_ schema*, which contains the name of each created star schema.

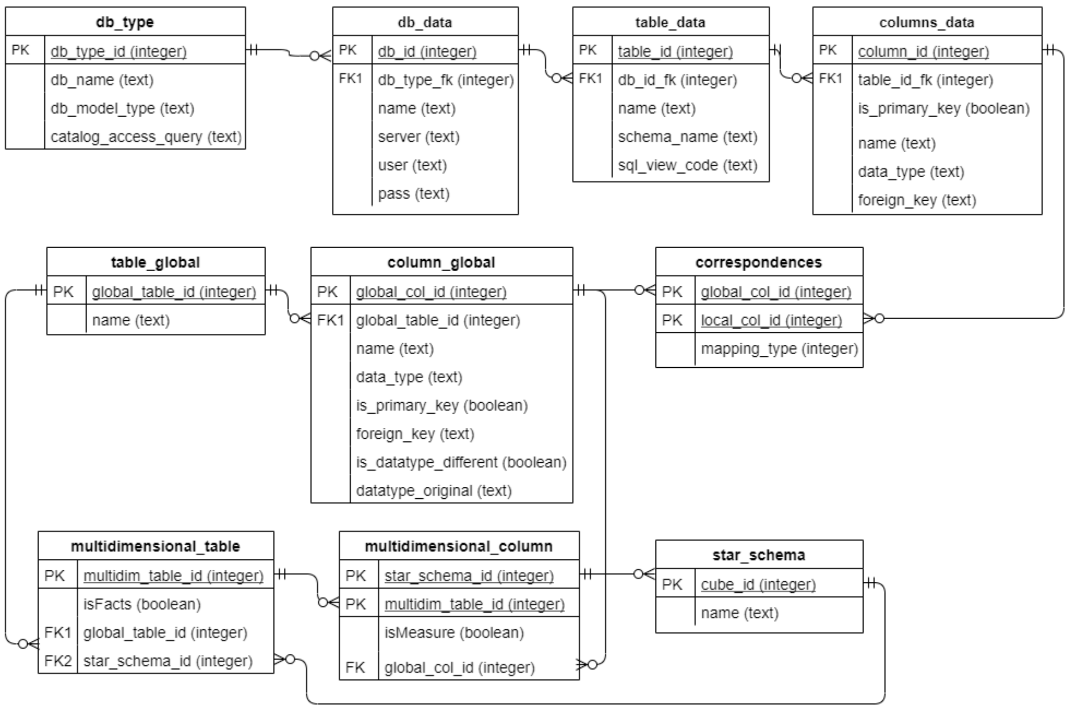


Figure 4.5: Database Diagram of the SQLite Database to store information regarding the star schemas, using tables from the global schema.

Algorithm 1: Detect partition type and mapping

```
Input  : A global table object containing correspondences to local tables
Output: Mapping type
Method:
    $globalColumns \leftarrow globalTable.getGlobalColumnsList()$;
    $localTables \leftarrow globalTable.getlocalTablesList()$;
    if localTables.size() == 1 then
        SimpleMapping;
    else if isVertical(globalTable) then
        VerticalMapping;
    else if isHorizontal(globalColumns, localTables) then
        HorizontalMapping;
    else
        UndefinedMapping;
Function isVertical(globalTable):
    $localTables \leftarrow globalTable.getPrimKeyColumn().getLocalTables()$;
    $fkTables \leftarrow getOnlyPKAndFKTables(localTables)$;
    if fkCols.size() == 0 then
        return false;
    end
    $primKeyOriginalColumn \leftarrow localTables.getPrimKeyNoFK()$;
    for localTable in fkTables do
        for column in localTable.getColumns() do
            if column.isPrimaryKey() and column.isForeignKey() then
                $referencedColumn \leftarrow column.getReferencedColumn()$;
                if referencedColumn.equals(primKeyOriginalColumn) then
                    continue;
                else
                    return false;
            end
        end
    end
    return true;
Function isHorizontal(globalColumns, localTables):
    for localTable in localTables do
        if localTable.getNumberColumns() != globalColumns.size() then
            return false;
        end
        for globalColumn in globalColumns do
            if ! localTable.columnExists(globalColumn.getName(),
              globalColumn.getdata type(), globalColumn.isPrimaryKey()) then
                return false;
            end
        end
        return true;
    end
    return false;
```

4.5 Analytical Queries Using A Global Schema

This section will focus on the solution developed for performing analytical queries using a star schema and visualize the results with the same structure defined in the star schema, while hiding the local schema containing the original data queried, which resides over multiple distributed and possibly heterogenous data sources. It will also deal with how to submit queries through the system, but details on how to fully build queries using the system's interface are available in Section 5.5.

4.5.1 Submitting Queries Through EasyBDI

It is important to define how users will interact with the system to submit queries, knowing that those queries will have to be delivered to Presto in SQL. Additionally, inner queries will have to be added in order to extract data from the local schema. The first approach considered was to have users writing queries in a well known language, such as SQL or MDX, with the latter being more suitable for multidimensional data, because such data requires many aggregations between each dimension table and the facts table. With MDX language, these aggregations are already implied, while in SQL all necessary aggregations would have to be manually written[3] [45]. Furthermore, MDX language was built with a different purpose than SQL, which is to handle multidimensional data [45], and to do so, MDX also allows control of data present in the x-axis and y-axis, unlike SQL language, aimed mostly for only 2 dimensional data [45].

In conclusion, MDX would be a much more suitable language for users to write analytical queries, however it is necessary to convert from MDX to a SQL syntactically and semantically valid query in order for Presto to run it. Research was done on libraries or tools that can make such conversion, but none was found because this conversion is not trivial, as it is necessary to reformat the MDX query to SQL, apply aggregations and other corrections, which, on top of that, should include the inner queries to retrieve data from local schema. Implementing a conversion algorithm can be time consuming. As such, the concept of writing queries on the system was abandoned in favor of creating queries through the user interface, more specifically by dragging elements of the schema to specific areas on the interface and create an SQL query according to the elements in each area. This concept was based on many applications that already allow users to interact with data in a similar manner, such as Microsoft Office Excel's dynamic tables, or Saiku Analytics, which also allows the execution of analytical queries.

These two systems are also used to define which areas should be available and what each one would mean for the query construction. Both applications have a rows and a columns area, meaning that it is possible to control which elements appear in the x-axis through the columns area and the y-axis through the rows area. A filter area also appears in both applications to create filters. An area to select the measures is also needed. Measures indicate quantitative data whose main analytic interests are calculations, and thus only have interest if they appear as a set of values aggregated and grouped by other categories, such as time and location.

Using the previously mentioned tools as an example, it is established that 4 areas for query creation are needed: one for defining rows (y-axis), another for the columns (x-axis), one for filters and a fourth only for measures. Each attribute of the schema placed in any of these areas will generate different queries, therefore it is necessary to explain the logic associated

[3]Example of MDX vs SQL: https://dba.stackexchange.com/questions/138311/good-example-of-mdx-vs-sql-for-analytical-queries/138830#138830

behind query generation for each of those areas, and the query is assembled and the challenges faced.

4.5.2 Creating Queries Capable Of Retrieving Data From The Local Schema To Global Schema

It is necessary to convert the data from the star schema perceived by the user to the local schema where the data resides. The tables in star schemas are essentially global tables with additional mappings to determine if they are dimensions or facts. To show data from the local schema to the global schema, all data from all local tables associated with a global table must be merged and integrated using the mappings previously created. These mappings are used to guide the creation of a query capable of retrieving data from the all local tables associated with a global table, and integrate their data as if a real global table exists. A global table is an entity representation used in EasyBDI, as the data resides in (possibly) various storage systems in the local schema, therefore, a global table is not actually created or instantiated, but it is presented as if it existed.

Because Presto adopts SQL as its query language, one can use nested queries in which the outer query contains the operations requested by the user, while also formatting the data to be coherent with the global schema structure (such as chaning table/column name), and the inner queries containing fetch the data from local tables to the corresponding global table. The inner queries are surrounded by parentheses and are executed before the outer query. The results of the inner query create the local schema entities, which will be used to create a representation of the defined global table on the outer query. It is possible to determine what are the correspondences of each column in the global table and their mappings by consulting the Schema/Metadata Storage module.

Three types of mappings exist, as explained in Section 4.4.2, and the creation of the inner queries depends on the type of mapping. To demonstrate how these global to local schema queries are created, assume that a global table "employees" exists and that a simple query is made by the user that select all elements from this global table. An example of global to local schema generated queries will be demonstrated to all 3 mappings.

Simple mapping queries are demonstrated first. Listing 4.1 shows an example of a query with an inner query fetching data from the correspondences of the global table, which in this case is just one local table, and the outer query will use the result of the inner query and assign an alias whose name is the same as the global table. The global table "employees" exists within the outer query, without actually being instantiated. A local table is referenced using a fully qualified name ("catalog.schema.table") in order for Presto to be able to run it, as mentioned in Section 3.3. Qualified names are shown in example query in Listing 4.1, but in order to simplify the queries, qualified names will be omitted from here on out. Double quotation marks are also used on table names or columns to escape certain characters that may exist, such as spaces. Although tables and columns from relational DBs may not have certain invalid characters, files are much more likely to have and the query will fail with invalid syntax if not escaped.

Listing 4.1: Example of a generated query for simple mapping relationships with full qualified names

```
SELECT * FROM (SELECT * FROM "empdb.store.employees_1")
```

```
AS "employees"
```

When global tables have more than one corresponding local table, it is necessary to retrieve information from all local tables coherently. As explained before, there are two possible mappings that have multiple local tables as correspondences of a global table: horizontal and vertical. The problem is on how to combine data from tables horizontally partitioned or vertically partitioned. For horizontal partitioned tables, data can be combined by applying an *union* operator between all tables. The example query in Listing 4.2 uses nested queries to retrieve data from horizontal partitioned tables that correspond to one global table.

Listing 4.2: Example of a generated query for horizontal mapping relationships

```
SELECT * FROM
(SELECT * FROM "employees_1" UNION SELECT * FROM "employees_2")
AS "employees"
```

For vertical partitioning, all tables can be combined by applying a *join* operation by the primary key columns of each one, as seen in the query in Listing 4.3

Listing 4.3: Example of a generated query for vertical mapping relationships

```
SELECT * FROM
(SELECT * FROM "employees_1" INNER JOIN "employees_2"
ON "employees_1"."emp_id" = "employees_2"."emp_id")
AS "employees"
```

Another example is shown in query in Listing 4.4, where only some columns are selected and aliases are created on each column of the local table, but only when the names of the columns in the local and global schema differ. For example, in query in Listing 4.4, column "name" in global table "employees" has a different name than the corresponding column in the global schema ("full_name"), therefore an alias is added to the local column so that these are recognized in the outer query as having the same name as the name in the global schema. This only works if all columns in the partition have the same name, otherwise this alias will become ambiguous. For example, if in horizontal partitioning, one table is designated as "name" and the other "full_name", assigning aliases with the same name to these columns will make the result in inner query ambiguous, as there are two different columns with the same name. Another strategy would have to be implemented to solve this issue.

There has been an effort to construct the inner queries efficiently, such that unnecessary columns do not appear in the select clause, thus improving performance. Tables from analytical scenarios may have several columns, but if only one is need from that table, it would be naive to simply select all columns from that table on the inner queries. If only one column of a global table has been selected, then the inner queries should only contain that column in the select statement, and this can be seen in the inner query in Listing 4.4. Additionally, the implemented query generation algorithm can understand that column "emp_id", primary key to both vertical partitioned tables, is needed for the join operation, and therefore adds this column to the select clauses of the inner query, even though the corresponding "emp_id" column in the global schema was not selected.

Listing 4.4: Example of a generated query for vertical mapping relationships where only a subset of columns are specified and the name of the columns differ between both schemas

```
SELECT "name", "isactive" FROM
 (SELECT "employees_1"."emp_id","employees_1"."full_name" AS "name",
 "employees_2"."emp_id", "employees_2"."isactive"
FROM "employees_1" INNER JOIN "employees_2"
ON "employees_1"."emp_id" = "employees_2"."emp_id") AS "employees"
```

For local tables partitioned vertically, this could mean that entire tables in the local schema could not be included in the inner queries, thus reducing Presto's operations even further. To demonstrate this, let us revisit query in Listing 4.4 and suppose that column "isactive" in the outer query is not selected. This would mean that an access to local table "employees_2" is unnecessary because no columns in the global schema were selected that map to any column of "employees_2", hence only "employees_1" needs to be selected. Query in Listing 4.5 demonstrates this example.

Listing 4.5: Example of a generated query for vertical mapping relationships where only one local table needs to be accessed instead of both local tables corresponding to the global table to retrieve data

```
SELECT "name" FROM
 (SELECT "employees_1"."emp_id","employees_1"."full_name" AS "name"
FROM "employees_1") AS "employees"
```

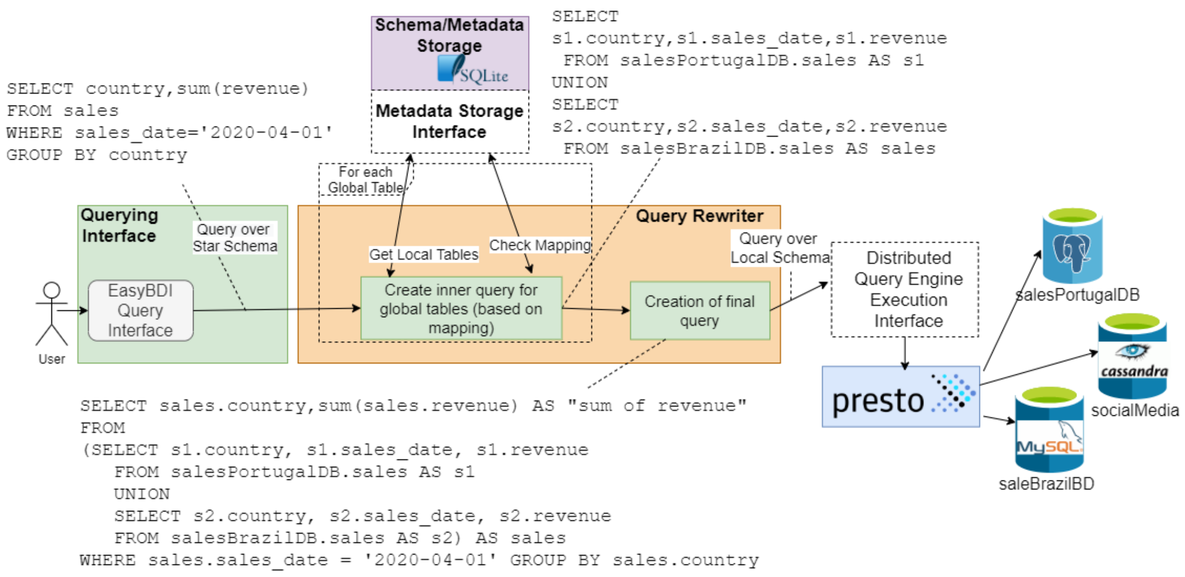
If however, "emp_id" is selected in the global schema, both local tables and a *join* operation are needed. This is demonstrated, although with a different example of table, in Table 4.2. This table further demonstrates this dynamic creation of queries, by showing a few more examples of queries made over the global schema and the generation of a complete query that includes both the global schema queries and the local schema queries in inner queries. Double quotes are omitted for a cleaner look.

4.5.3 Creating And Executing Queries In EasyBDI

The whole process of query execution in EasyBDI is summarized in Figure 4.6, with the identification of the intervening system modules previously explained in Section 4.2 and depicted in Figure 4.1. When a user submits a query using the query UI, the system will create a valid SQL query according to user input. For each global table present in the query, all local table's names, data source location, and mapping are extracted from the Schema/Metadata Storage module and a inner query is generated based on the mapping type, as discussed previously. In the example presented in Figure 4.6 the user creates the query shown on the top left through the interface and an inner query is added for the global table "sales", which has a horizontal mapping assigned for the two corresponding local tables. The inner query selects only the attributes that match the selected attributes in the global table, and performs a union on both local tables in order to return all records, and thus present all data from the global table. An alias is added to the inner query with the name of the global table, and the inner query is added to the original query and sent to Presto for execution. Results are then showed in the interface.

Table 4.2: Example of query generation to extract records from the local schema using inner queries. Vertical partitioning exists in the local schema. Some local tables are not added in the inner queries if their attributes are not necessary.

Example	Global Schema Query	Global Schema With Local Schema Query
1	SELECT id, email FROM employees	SELECT id, email FROM (SELECT id, email FROM employees INNER JOIN employees_details ON employees.id = employees_details.id) AS employees
2	SELECT email, job_description FROM employees	SELECT email, job_description FROM (SELECT email, job_description FROM employees_details INNER JOIN employees_job_info ON employees_details.id = employees_job_info.id) AS employees
3	SELECT email FROM employees	SELECT email FROM (SELECT email FROM employees_details) AS employees



```
SELECT country,sum(revenue)
FROM sales
WHERE sales_date='2020-04-01'
GROUP BY country
```

```
SELECT
s1.country,s1.sales_date,s1.revenue
 FROM salesPortugalDB.sales AS s1
UNION
SELECT
s2.country,s2.sales_date,s2.revenue
 FROM salesBrazilDB.sales AS sales
```

```
SELECT sales.country,sum(sales.revenue) AS "sum of revenue"
FROM
(SELECT s1.country, s1.sales_date, s1.revenue
   FROM salesPortugalDB.sales AS s1
   UNION
   SELECT s2.country, s2.sales_date, s2.revenue
   FROM salesBrazilDB.sales AS s2) AS sales
WHERE sales.sales_date = '2020-04-01' GROUP BY sales.country
```

Figure 4.6: Query transformation process. A query is created in SQL based on user input and nested queries are added to extract data from local tables.

4.6 Generation Of Analytical Queries

This section offers a more in-depth explanation as to what type of queries can be made by the user and how they are generated. The user input is used to generate the outer query, and inner queries are added for each different global table in order to be able to extract information from the local schema, as explained in the previous section. Queries are created by placing attributes in specific areas on the interface. Each area has a different logic and meaning associated towards constructing the end query, and therefore each one will be explained individually in this section. The main operations that can be generated on each query are:

- Aggregate functions.
- Order by operations.
- Filter operations, including addition of logical operators.
- Joins between dimensions and facts tables.
- Data pivoting.

4.6.1 Adding Rows To Analytical Queries

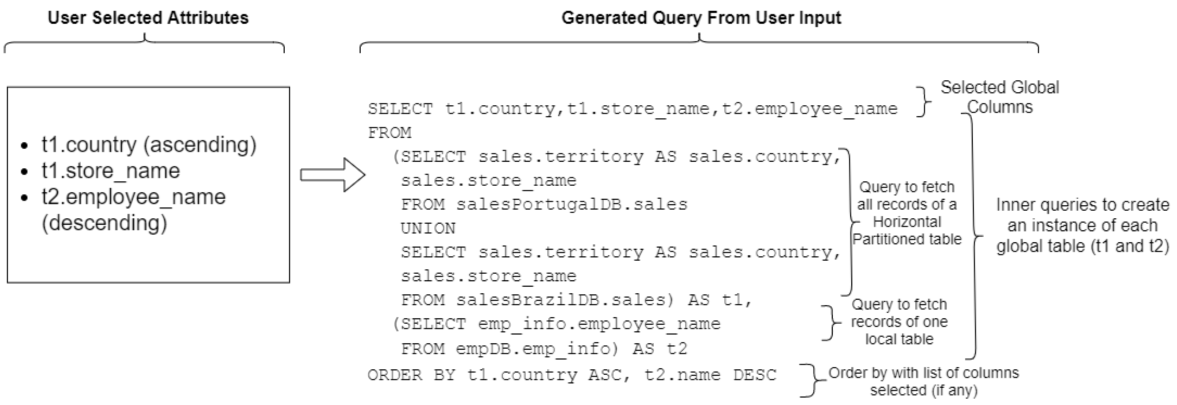
The addition of attributes in the rows area is the equivalent of adding attributes in a *select* clause in a SQL query. The *from* clause must include all different tables belonging to attributes in the *select* clause.

In addition, it is very helpful to be able to sort attributes, and users can do this and select if they want to see the values in ascending or descending order. If an *order by* is added using the interface, then the query generator must add an *order by* clause containing the list of all columns that the user selected to sort, along with the order (descending or ascending).

Figure 4.7 shows an example of a generated query assuming that a user, through the interface, selected three attributes of dimension tables (the rectangle on the left shows the user input), and some of those attributes were also selected to be sorted in the final query. This generates a query as shown on the right, which is then given to Presto. This example selects two attributes from global table "t1" and one from "t2" with two of those attributes being selected to be sorted in a *order by* clause (seen in the last line of the query in Figure 4.7). Note that some quotation marks applied to columns and tables are omitted for simplicity.

In this example, the local tables that correspond to global table t1 have been identified as having an horizontal mapping, hence the inner query will select the columns corresponding to the ones selected by the user from both local tables and perform a *union* operation. The inner query is assigned an alias so that these records are identified as the global table's name, t1. Global table t2 has a simple mapping, therefore only one table is selected in the inner query along with the necessary columns on the inner query for t2. Note that for each inner query, the columns of the local tables also contain aliases, but only if the name of the global column is different from the name of the corresponding local column. In the example query seen in Figure 4.7, the global column "country" contains a correspondence to a local column named "territory", and because both names differ. Since the outer query only deals with the name columns assigned in the global schema it is not possible to select the column "territory" existing in the inner query from the outer query, which uses the name "country". An alias with the global column's name is added to ensure that this is possible, and now the column "territory" is named "country", and therefore accessible to the outer query. However, global column "store_name" has the same name as its corresponding local table, hence no alias needs to be added in the inner query for each corresponding local column.

Aggregate functions are also possible to be added to queries by specifying the attribute and the aggregate function to be used (details on how to do this using the user interface in Section


User Selected Attributes
Generated Query From User Input
• t1.country (ascending)
• t1.store_name
• t2.employee_name (descending)
Selected Global Columns
Query to fetch all records of a Horizontal Partitioned table
Inner queries to create an instance of each global table (t1 and t2)
Query to fetch records of one local table
Order by with list of columns selected (if any)

```
SELECT t1.country,t1.store_name,t2.employee_name
FROM
  (SELECT sales.territory AS sales.country,
   sales.store_name
   FROM salesPortugalDB.sales
   UNION
   SELECT sales.territory AS sales.country,
   sales.store_name
   FROM salesBrazilDB.sales) AS t1,
  (SELECT emp_info.employee_name
   FROM empDB.emp_info) AS t2
ORDER BY t1.country ASC, t2.name DESC
```

Figure 4.7: Illustration of the SQL query generation process using the user selected columns from dimension tables.

5.5.2). However, this generated additional validation that needed to be implemented on the query generation process. For example, if an aggregate function is used in an attribute, and the same query includes one or more attributes without aggregate functions, those attributes need to be added to *group by* clause, else Presto will fail to execute the query. The query generator obeys to this condition, and adds each attribute to a *group by* clause if an aggregate is used in the query.

Aggregated attributes are also given an alias in order to improve the readability of the query output. This alias, is in the form "<aggregate function> of <attribute>". For example, if the attribute 'names' add a *count* aggregate function applied, then that column would have the name "count of name" in the final output. This is also useful to distinguish between columns with the same attribute but different operations on the same query, such as an attribute with *avg* and another one with *sum*, which will be easily distinguishable by it's name given by the added alias. An example of a query containing aggregate functions applied is shown in Figure 4.8.

4.6.2 Adding Measures For Analytical Queries

When a measure from the facts table is selected to be included in the query, the user may be interested in carrying out aggregation operations on that measure, because, in analytical queries, measures are mostly used to describe quantitative aspects and are therefore used in calculations. In this case, users must select which of the supported aggregate functions to use: Sum, Count and Average. Max and Min were not implemented. Multiple measures may be used on the same query and the same measure may appear multiple times as long as it contains distinct operations (such as no aggregate function and again with an aggregate function).

In order to coherently merge records between the dimension tables and the facts table, a condition is used in a *where* clause between the facts and each dimension using the shared attribute between both tables (the primary key on the dimension and the foreign key referencing said attribute on the facts), as seen in the query example in Listing 4.6.

Listing 4.6: Example of a query using the equivalent of a Join operation in the where clause

```
SELECT * FROM facts_table , dimension1 , dimension2
```

```
WHERE facts_table.dim1ID = dimension1.id AND
facts_table.dim2ID = dimension2.id
```

If filters were added by the user, then an *AND* must be added to the where clause.

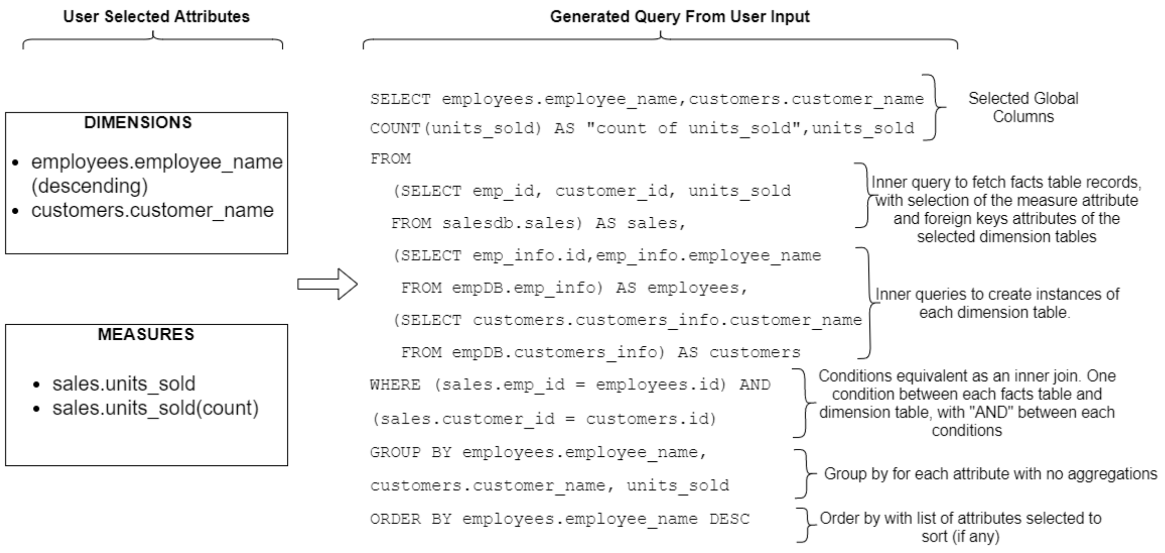
Any aggregate function used in either a dimension attribute or a measure implies adding a *group by* clause containing all other attributes in the select clause that do not have an aggregation operation, if any.

A complete example of a generated query were measures are used is illustrated in Figure 4.8, using as an example a hypothetical analytical query whose main purpose is to demonstrate how these queries are built. This query selects the number of units (of products) each employee and customer sold/bought (count(units_sold)) in each purchase registered and the total amount of units sold (units_sold) between all registered purchases, as well as the name of the client that made the purchase and the name of the employee that registered the purchase. To do this, the user selects the columns regarding employee names in a dimension table, and the column containing customer names in other dimension table and a measure named "units_sold" with a *count* operation and again the same measure but with no operation. For the three global tables in the *from* clause (two are dimension tables and one is the facts table), an inner query is created, once again selecting only the necessary attributes for the outer query in each local table and for the join between the dimensions and facts table, namely, the primary keys/foreign keys (for simplicity, all mappings in this example are simple mappings). The local tables corresponding to "employees" selects only the attributes "employee_name" which was selected in the *select* clause and the primary key "id" needed to join between the facts table. The same goes for the other dimension table "customers" and for the facts table, which selects only the measure and both foreign key attributes referencing the primary keys of the dimensions ("emp_id" and "customer_id"). To perform an equivalent operation as an *inner join* between the facts and each dimensions, a *where* clause is added with a equality comparison between the dimension table's primary key and the respective foreign key in the facts table, resulting on the condition "sales.emp_id = employees.id". The same is done between "sales" and "customers", and the process would be repeated again with another dimension table if more attributes belonging to different dimension tables were specified in the *select* clause. A *group by* clause contains attributes "employee_name" and "customer_name", as well as the measure "units_sold" which had no aggregation function associated.

An alias is also created just as it happens with dimension attributes, as can be seen in Figure 4.8, where a "count" operation is made on the measure "units_sold", and therefore it's alias is "count of units_sold", and the measure with no aggregate function "units_sold" remains with the same name. Note that the same measure can only be added multiple times to a query if it contains distinct operations, such as aggregate operations and one or zero columns with no aggregate function applied to the attribute.

4.6.3 Pivoting Data In Analytical Queries

Certain attributes may be selected by the user to present their values in x-axis of the output, thus forming a new column per value (here we designate this column as a pivoted column). Therefore, the generated query outputs the distinct values of the attributes, which will be converted into column labels in the final query output, and any other values from measure attributes used in the query must be grouped correctly among those columns. SQL contains an operation to do this, called *pivot*, however this operation is not natively supported on Presto, therefore this operation needs to be manually created. Furthermore, one needs


User Selected Attributes
Generated Query From User Input
DIMENSIONS
• employees.employee_name (descending)
• customers.customer_name
MEASURES
• sales.units_sold
• sales.units_sold(count)
Selected Global Columns
Inner query to fetch facts table records, with selection of the measure attribute and foreign keys attributes of the selected dimension tables
Inner queries to create instances of each dimension table.
Conditions equivalent as an inner join. One condition between each facts table and dimension table, with "AND" between each conditions
Group by for each attribute with no aggregations
Order by with list of attributes selected to sort (if any)

```
SELECT employees.employee_name,customers.customer_name
COUNT(units_sold) AS "count of units_sold",units_sold
FROM
  (SELECT emp_id, customer_id, units_sold
  FROM salesdb.sales) AS sales,
  (SELECT emp_info.id,emp_info.employee_name
   FROM empDB.emp_info) AS employees,
  (SELECT customers.customers_info.customer_name
   FROM empDB.customers_info) AS customers
WHERE (sales.emp_id = employees.id) AND
(sales.customer_id = customers.id)
GROUP BY employees.employee_name,
customers.customer_name, units_sold
ORDER BY employees.employee_name DESC
```

Figure 4.8: Illustration of the SQL query generation process using the user selected attributes from dimension tables and measures from the facts table.

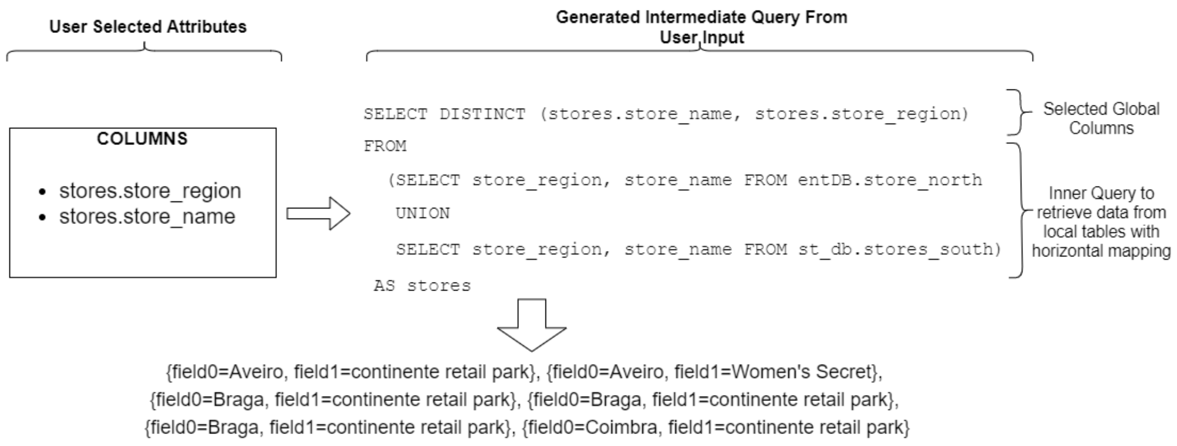
to know in advance all distinct values of the attributes of which to create the columns on the output. The values that should fill the rows of pivoted values should be of measures or aggregate operations applied to measures. For this reason, whenever a column is specified to be pivoted, one, and only one measure must be specified by the user.

First, it is necessary to create and execute a query to retrieve all distinct elements of an attribute. The query is simple, as seen in Figure 4.9, as it only returns the distinct values of the attributes that are needed to produce the final output expected by the user. This query also contains an inner query to retrieve data from the local tables, which in this example, is horizontally partitioned. The output is a single column, with a collection containing fields with the different values of each attribute, and this output contains a combination of all possible values between the selected attributes. For example, the output of the query seen in Figure 4.9 contains all combinations of different values between attributes "store_region" and "store_name", but no combination of values is repeated within the results. These values will be stored in memory for the duration of the generation process of the final query.

The values are then used to create new columns in the output of the user requested query, similar as the *pivot* operator. In the absence of a *pivot* operator in Presto, it is necessary to manually create columns and position the correct rows (the values of the measures) in the corresponding columns. One approach was to use Presto's *map_ agg* [4] which receives a map with key-value pairs, the key being an attribute whose different values should form columns on the output, and the value another attribute whose values of a measure attribute should be grouped with the values of the attribute on the key. However, this strategy resulted in several errors, such as a "key not present", even though such value existed on the records and on the attribute used on the map. Because of these complications, this method was dropped.

Another approach was adopted which consists on using a *case*[5] statement to create new columns and aggregate values in these new columns. These statements contain one or more

[4]https://prestosql.io/docs/current/functions/aggregate.html?highlight=map_agg#map_agg
[5]https://prestosql.io/docs/current/functions/conditional.html?highlight=case


User Selected Attributes
Generated Intermediate Query From User Input
COLUMNS
stores.store_region
stores.store_name
Selected Global Columns
Inner Query to retrieve data from local tables with horizontal mapping

```
SELECT DISTINCT (stores.store_name, stores.store_region)
FROM
  (SELECT store_region, store_name FROM entDB.store_north
   UNION
   SELECT store_region, store_name FROM st_db.stores_south)
 AS stores
```

```
{field0=Aveiro, field1=continente retail park}, {field0=Aveiro, field1=Women's Secret},
{field0=Braga, field1=continente retail park}, {field0=Braga, field1=continente retail park},
{field0=Braga, field1=continente retail park}, {field0=Coimbra, field1=continente retail park}
```

Figure 4.9: Generation of an intermediate query to retrieve all distinct values from attributes, and example of the resulting output whose values will be used on a final query as columns.

conditions, and the statements returns a value depending on the condition that evaluates true, or the value in the 'else' clause if no other condition is true. These statements have the following structure:

```
CASE WHEN value THEN result
    [ WHEN ... ]
    [ ELSE result ]
END AS tableName
```

This statement means that whenever a certain value appears, a specific result must be added to that value's column.

When adding a measure to the query, it is possible to choose between 3 aggregate functions: *count*, *sum* or *average*. It is also possible to not use an aggregate function. This selection will slightly change the case statement. A summary of all the case statements for each of the previous configuration is presented in Table 4.3. Note that it is not possible to add more than one measure to the query (even if the same measure has 2 different aggregate functions).

To correctly name each pivoted value into a user friendly column name, an alias is added with the name of the value of the attribute being pivoted. However, if multiple attributes are being pivoted, the name of each column is the concatenation of each value from each attribute followed by an hyphen between between each value. As an example, if value "Portugal" from attribute "country" and value "Aveiro" from attribute "region" are pivoted into a single column in the output, the column's name will be "Portugal-Aveiro".

Again, in order to avoid SQL syntax error, the following validations are made to aliases:

- If a value is empty, resulting in an alias containing an empty string, then the placeholder "empty" will be used as the column name, because empty strings are invalid column names.

- Enclosing the values composed by digits only using double quotes will make it an acceptable column name.

- Enclosing in double quotes also solves the issue where one or more spaces exist, as well as single quotes which would otherwise be interpreted as the string enclosing character

Table 4.3: Example of case statements applied depending on the aggregate function to pivot data.

Aggregate Function	Case Statement
No Aggregate Function	(CASE WHEN columnName = 'value' THEN 1 ELSE 0 END) AS "value"
Count	SUM(CASE WHEN columnName = 'value' AND measure not null THEN 1 ELSE 0 END) AS "value"
Sum	SUM(CASE WHEN columnName = 'value' THEN measure ELSE 0 END) AS "value"
Avg	AVG(CASE WHEN columnName = 'value' THEN <measure> ELSE 0 END) AS "value"

(such as in "Women's Secrets"). Double quotes are therefore added to all values when adding the alias for each column.

To use the values of the columns obtained from the query in Figure 4.9 in the case statement, more specifically in the condition "columnName = <value>" of the case statement, the values must be treated as literals, otherwise the exception "cannot be resolved" is raised. First, the data type of the attribute will decide if the values should be enclosed with single quotes or not. For numeric data types, single quotes must not be added. For chars, varchars, dates and many other data types, single quotes are necessary. For numeric data types, single quotes must not be added. In this case, spaces are not a problem, but single quotes within the value are, as they will be interpreted as the end of the string and an error with invalid syntax, is returned. However, as with most relational DBMS, such characters can be escaped in Presto by placing another single quote next to each occurrence of the first one (example: "Women''s Secret).

Another issue to consider are empty values, which will again cause a syntax error. Whenever an empty value is detected two single quotes are used, thus avoiding such error.

Group By clauses are added to include other attributes that were not included in aggregate functions (if any). In the example in Figure 4.9, the distinct values of the attributes specified in the "columns" area (that will be pivoted) are not added to the *group by* clause in the generated query, because they are part of an aggregation. However, if the measure does not contain an aggregate function, as seen on the first line of Table 4.3, then all of the pivoted values must appear in the *group by* clause. This poses a new problem, as aliases cannot be added to *group by* clauses because of the order SQL executes each clause. A *group by* clause is executed before the *select* clause, and any aliases created on the *select* clause are not yet defined and cannot therefore be used on clauses processed before the select clause, such as the *where*, *having* and *group by* clauses [46]. The order in which the clauses are executed is shown in Figure 4.10.

Without referencing the alias, the full case statement must be used to identify the column created. But this would imply that each case statement would have to be written again on

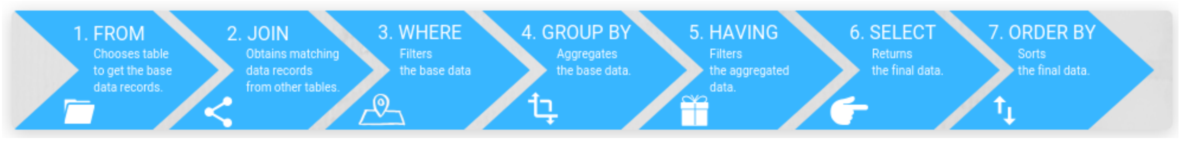
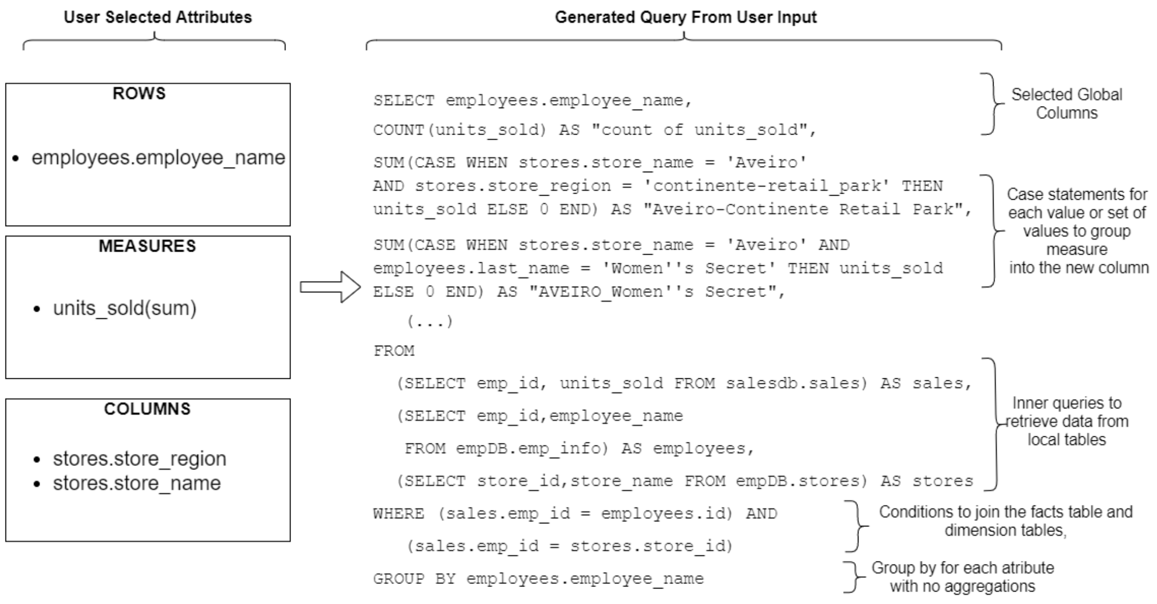

1. FROM
Chooses table to get the base data records.
2. JOIN
Obtains matching data records from other tables.
3. WHERE
Filters the base data
4. GROUP BY
Aggregates the base data.
5. HAVING
Filters the aggregated data.
6. SELECT
Returns the final data.
7. ORDER BY
Sorts the final data.

Figure 4.10: Order of execution of each clause in a SQL query [4] on MYSQL, and also used in Presto. May vary depending on the relational DBMS.

---

the query. The adopted solution, is to use a number representing the order of the created columns in the select statement to refer to these columns, which is accepted by the *group by* clause. As with previous queries, *group by* is only used when aggregate functions are used, otherwise they are not added to the query.

An example of a final query containing measures and attributes transformed to columns is illustrated in Figure 4.11.


User Selected Attributes
Generated Query From User Input
ROWS
• employees.employee_name
MEASURES
• units_sold(sum)
COLUMNS
• stores.store_region
• stores.store_name
Selected Global Columns
Case statements for each value or set of values to group measure into the new column
Inner queries to retrieve data from local tables
Conditions to join the facts table and dimension tables,
Group by for each atribute with no aggregations

```
SELECT employees.employee_name,
COUNT(units_sold) AS "count of units_sold",
SUM(CASE WHEN stores.store_name = 'Aveiro'
AND stores.store_region = 'continente-retail_park' THEN
units_sold ELSE 0 END) AS "Aveiro-Continente Retail Park",
SUM(CASE WHEN stores.store_name = 'Aveiro' AND
employees.last_name = 'Women''s Secret' THEN units_sold
ELSE 0 END) AS "AVEIRO_Women''s Secret",
    (...)
FROM
   (SELECT emp_id, units_sold FROM salesdb.sales) AS sales,
   (SELECT emp_id,employee_name
    FROM empDB.emp_info) AS employees,
   (SELECT store_id,store_name FROM empDB.stores) AS stores
WHERE (sales.emp_id = employees.id) AND
    (sales.emp_id = stores.store_id)
GROUP BY employees.employee_name
```

Figure 4.11: Example of a query containing attributes as columns, and measures grouped on each of those columns.

4.6.4 Creating Filters

Filters are created by dropping attributes to a filters area in the UI. When dropping an attribute, a condition must be created using that attribute. The system's UI related logic ensures that valid conditions are created. Afterwards, whenever a query needs to be created and submitted, the system reads each created condition on the UI and creates the SQL code that will be placed after the *where* clause, or if the query uses conditions to perform joins between the facts and the dimensions (such as the query seen in Figure 4.8), an *AND* is added to the already existing *where* clause followed by the user created filters. Details regarding the interaction with the UI to create filters as well as how the filters are generated with proper SQL syntax from the user input will be explained in Section 5.5.3. Note that there is also an advanced option to manually create the filters, which will be explained in Section 5.5.4. This section will focus on using the SQL generated filters and added to the overall query.

When generating the final query, filters will be used against the global tables created using the inner queries, and are not applied in each inner query, or, in other words, filters are applied over the global schema and not on the local schema.

It is possible to include in the filters attributes that are not present in the selected attributes that will be used as rows or columns in the query output. For example, if only the attributes "month" and "day" from table "time" are selected, it is still possible to filter only by the attribute "hour" in the same table, even if not selected in any other area (rows, columns or aggregations areas). However, this adds another issue that needs to be addressed: inner queries need also to consider attributes in the filters area and include the corresponding attribute of the local schema in their *select* statements. If this is not done, then only the "month" and "day" attributes will be selected in the inner query, and the outer query will not have defined any attribute "hour" from the local schema, as portrayed in Listing 4.7. To solve this, a verification is done for each of the attributes referenced in the user created filters, and attributes are added to the list of attributes to be selected in the inner queries.

Listing 4.7: Example of an incorrect query where one of the attributes used in the filters is not selected within the inner queries, thus generating an SQL error

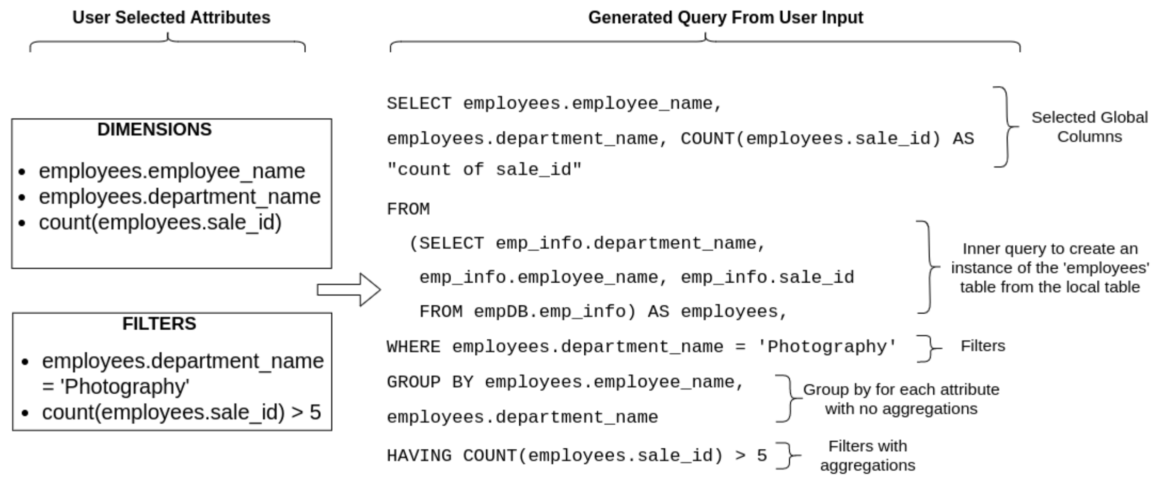
```
SELECT time.day, time.month, FROM
    (SELECT db.time.day, db.time.month FROM db.time) AS time,
WHERE time.hour = time '12:30';
```

It should be noted that current implementation of query generation algorithm does not consider tables containing attributes only present in the filters. It is necessary that all different tables with at least one attribute selected in filters also have at least one attribute in the rows or aggregations area (does not need to be the same attribute).

Another form of filters can be executed on EasyBDI which uses attributes containing aggregate functions. A *where* does not support aggregated attributes, so the *having* clause is used. Any conditions created with aggregated columns must be placed on this clause, but aliases cannot be used on either *where* or *having* clauses because they are not recognized on Presto, unlike many relational DBMS, such as MySQL, which allows the use of aliases on these clauses.

Figure 4.12 shows an example of a query that outputs all employees from the "Photography" department that made more than 5 sales. The attribute "sale_id" has a *count* aggregate function in order to count all sales, and a filter is applied in the *having* clause to this aggregation in order to show values greater than 5. Note that the condition on the *having* clause is explicitly written instead of referring to the alias assigned ("count of sale_id"), as no aliases can be used here (because of the execution order of a SQL query, seen in Figure 4.10.

An exception is made for filters applied to attributes selected to be transformed into columns. In this case, the values from those attributes need to be filtered already in the final query (seen in Figure 4.11), therefore those filters are instead applied to the query that outputs all different values of those attributes in order to filter out those values (query seen in Figure 4.9).


User Selected Attributes
DIMENSIONS
employees.employee_name
employees.department_name
count(employees.sale_id)
FILTERS
employees.department_name = 'Photography'
count(employees.sale_id) > 5
Generated Query From User Input
Selected Global Columns
Inner query to create an instance of the 'employees' table from the local table
Filters
Group by for each attribute with no aggregations
Filters with aggregations

```
SELECT employees.employee_name,
employees.department_name, COUNT(employees.sale_id) AS
"count of sale_id"
FROM
  (SELECT emp_info.department_name,
   emp_info.employee_name, emp_info.sale_id
   FROM empDB.emp_info) AS employees,
WHERE employees.department_name = 'Photography'
GROUP BY employees.employee_name,
employees.department_name
HAVING COUNT(employees.sale_id) > 5
```

Figure 4.12: Query transformation process containing filters with and without aggregations. Filters are applied to the global table's attribute and not to the local schema queries in the inner queries.

4.7 Considerations

This chapter introduced EasyBDI, a system that is able to query heterogenous and distributed data sources, while also providing near real-time querying by directly pulling data from the data sources. These features are mainly achieved by the distributed query engine used, Presto. However, Presto alone does not fill the remaining requirements specified in Section 1.3, as seen in Table 2.1, such as the creation of global schema, star schema, or a user interface that simplifies the overall configuration of the schema and query creation.

This motivated the implementation of a system built on top of Presto, named EasyBDI, that is capable of hiding the internal organization of the data sources by automatically creating a global schema, which can be modified through a user interface. Star schemas can also be created by mapping entities in the global schema to the star schema, and queries are submitted using said star schema and also through a user interface based on drag-and-drop so that users from areas outside of the database system management can also use the system.

EasyBDI's architecture was introduced in this chapter. The top level layer refers to user interface logic. Layers responsible to create and maintain the schemas are present, namely the local, global and star schemas, including a layer specific for data integration techniques used to create a global schema. Any data partitioning is automatically detected using a developed algorithm and hidden by creating a global table that corresponds to the multiple fragmented tables. These schemas are stored in SQLite.

When a query is created using drag-and-drop and submitted through EasyBDI, a SQL query is generated containing all operations needed to present the requested results to the query. Inner queries that are responsible to return data from the local schema are added to the overall query, which is then passed and executed by Presto. Results are presented to the user and organized accordingly with the star schema. No data is materialized, meaning that global tables are never created. Instead, the outer query passed to Presto is also responsible to modify the data according with the star schema structure, such as changing the names of the tables/columns from the results of the local schema. A very in depth guide on all the possible query generation scenarios has been described in this chapter.

This chapter explained the core logic behind EasyBDI. Next chapter will demonstrate how to use the system and present all of its features, such as schema configuration and query execution. Other minor features and implementation aspects not introduced here are also presented, while also explaining how some of those features work whenever necessary.

Chapter 5

EasyBDI Demonstration

This chapter will serve as a walkthrough for the developed system from a user perspective, detailing it's features, how to use its functionalities, how they were implemented and possible limitations.

The system has two main functionalities: creation and configuration of local, global and multidimensional star schema and execution of analytical queries. These contain several details and many other minor features implemented in order to better assist the user. This section introduces the system's UI and how to interact with the system to configure the schemas and to create queries.

5.1 System Users

The main tasks in EasyBDI are the creation of local, global and star schemas, and execution of analytical queries. However, these tasks require different levels of knowledge and it is not expected that a person is capable of executing all of these tasks in the system. These tasks and responsibilities are therefore classified in 3 different types of users:

- Data source Administrator - responsible to connect the platform to each data source, making sure that the platform has permission to access its data.
- Data Administrator - involved in the data source integration process, which requires the definition of a global schema.
- Business Analyst - performs analytical queries through the platform using the global schema.

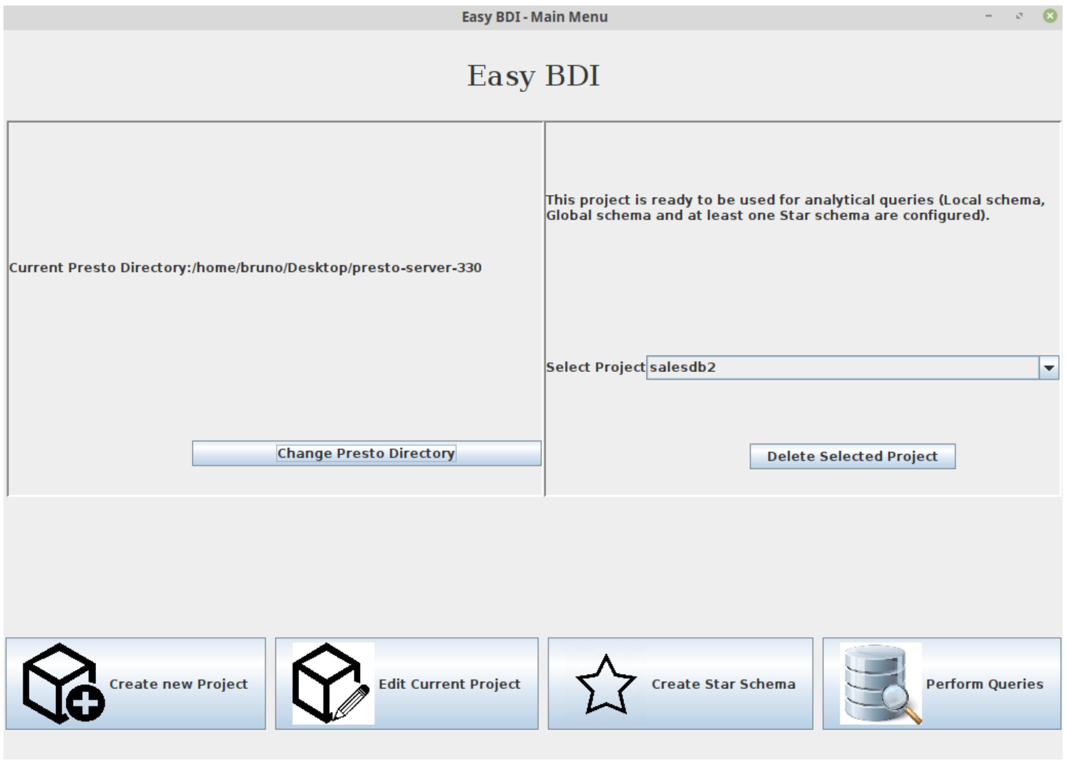
One user can have more than one roll in the system depending on his level of expertise. For this dissertation, the system was implemented without any authentication logic for simplification and demonstration purposes, and assumes the user has full access to the system and can execute every functionality the system has to offer.

5.2 EasyBDI - Main Menu And Basic Concepts

EasyBDI uses the concept of projects that users need to create, each of them containing the local schema view information (with data source information, including connection information to access them, and local tables/columns information), a global schema and one or more star schemas. The creation of different projects allows an easier separation of data and schemas created to be used for specific contexts or use cases. One user can create a project entirely dedicated to analyze the overall sales of the company, while other project can be focused only on inventory management and company expenses. Internally, the Schema/Metadata storage layer creates one SQLite database file for each project, using the schema presented in Figure 4.5.

The initial screen of the system, depicted in Figure 5.1, features a main menu and provides a point of access to either create a new project or to select and use a previously created project to either edit it or perform analytical queries. Users can switch between projects anytime and may even delete projects. Again, to better exemplify the use of this system, a user with all roles is assumed to be using the system.

The system also allows the user to specify a Presto Server directory, however this node is assumed to be the coordinator. The user is responsible to configure Presto and the cluster intended to use with EasyBDI.



This project is ready to be used for analytical queries (Local schema, Global schema and at least one Star schema are configured).

Current Presto Directory:/home/bruno/Desktop/presto-server-330

Select Project salesdb2

Figure 5.1: Main Menu of EasyBDI.

This section will explain all the available features in EasyBDI and how to use them. A few implementation details will also be discussed, sometimes referring to content explained in previous chapters.

5.3 EasyBDI - Creation of New Projects

Wizards are used to provide guided solutions to users for the creation and configuration of new projects. Such configuration consists in adding new data sources, configuring the system's proposed global schema and definition of a multidimensional star schema. Users may quit the configuration project halfway through, but must complete it to perform queries, as the "Perform Queries" button will not be active if no local schema, global schema and/or at least one star schema is created. The selected project on the main menu will present a message indicating if it is ready to be used for analytical queries or if a configuration is incomplete, such as missing local schema or global schema. In this case, users must select the "Edit Project" button to conclude the configuration process, as explained in Section 5.4.

5.3.1 Data Source Configuration

The first step to create a new project is to connect existing data sources to EasyBDI, therefore, a Data source Administrator must fill in information in order for EasyBDI to connect to each data source, such as URL, name, data source model and, if needed, credentials (username and password). The data source administrator is responsible to provide permissions for the user (if applicable) to execute queries from the data source.

Note that different data sources have different concepts of databases and schemas, and therefore may differ on how each server is organized, a topic briefly discussed in Section 4.3. For example, in MySQL it is not necessary to specify a database name, as access is given to all schemas within a single server, while in PostgreSQL one must specify the database name within the server. This means that in some data sources, such as PostgreSQL, users must also supply a database name in order to connect to that database, while in other data sources, such as MySQL, the database name is irrelevant and is only used as a way to identify the data source within the system. To overcome this, the field "data source name" is only visible when necessary. In case the data source requires a database name in its URL, the text field must end with "/<database name>", and this name will also be used as the data source's name within EasyBDI.

Figure 5.2 shows the screen in which users configure data sources. In this image, Cassandra is currently selected, and because a keyspace does not need to be supplied within the connection URL, the field "data source name" is visible.

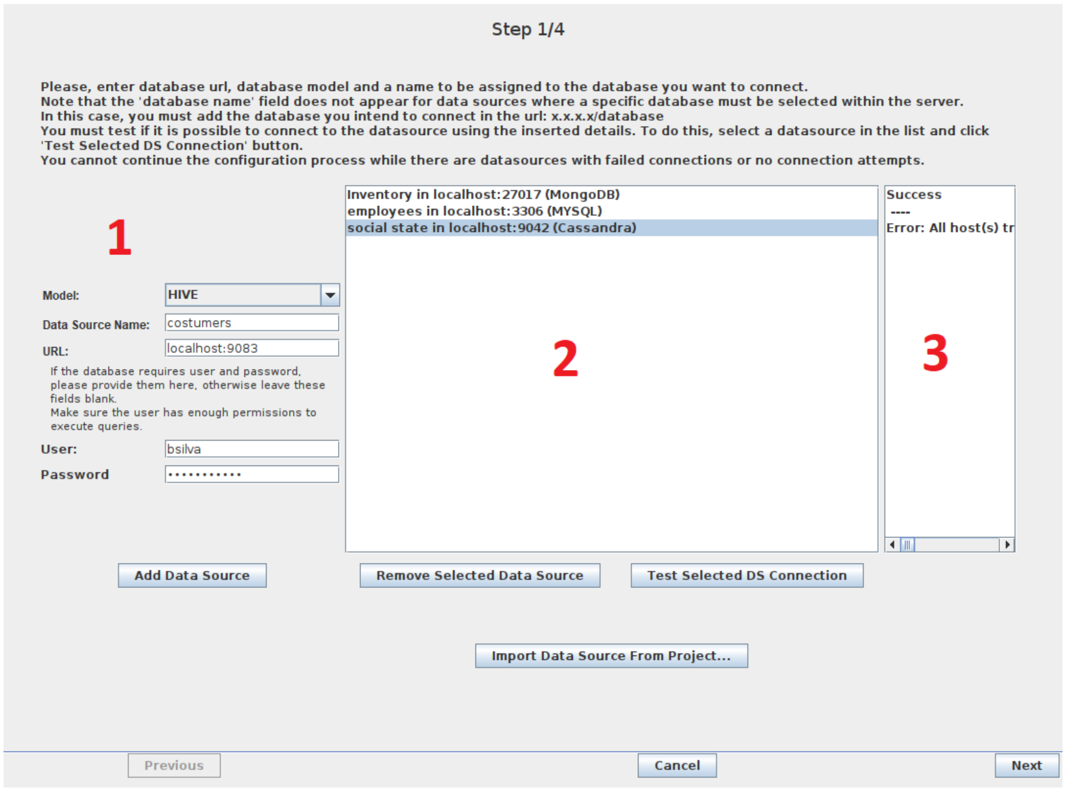
Users can add as many data sources as they want after filling in the required information and clicking the 'Add Data Source' button, which adds to the list marked with the number 2 the data source details. If a specific data source has been used before in another project, it is possible to import its details without entering all information again. This feature can be used by clicking the 'Import Data Source From Project...', which opens a new window with a list of all different data sources from all currently available projects in EasyBDI. The list in Figure 5.2, marked with number 2, contains three data sources.

Before moving on, a connection to each data source must be tested by selecting one data source in the list number 2 and then clicking on 'Test Selected DS Connection' and Presto will temporarily create the appropriate configuration file for the selected data source and restart itself in order to properly use the new properties file created. When restarted, Presto tries to connect to the data source by performing a simple query, *show schemas from <data source>*. If a connection cannot be established the user is alerted with a warning message along with the cause of the error. This error log will be added in the list marked with the number 3 in

Figure 5.2. The user must either edit the details of the data source or delete it, as it is not allowed to continue while there are any data sources with unsuccessful connection attempts or whose connectivity was not tested. For example, the print seen in Figure 5.2 shows that a connection was tested and was successful to MongoDB, but failed on Cassandra, and no connection testing was made for MySQL. Also note that, before moving on, all relational databases must use the scripts supplied in the project's repository [43] to create views that allow the system to retrieve constraint information (primary and foreign keys), as EasyBDI queries those views instead of the tables in the "information_schema" (refer to Section 3.4 for more details regarding this issue). This information will be used to determine the mapping type, as explained in Section 4.4.2.

Once all data sources on the list in area 2 are marked with a "Success" message on area number 3, users can move on to the next step.

At this point, the system retrieves information regarding data organization in schemas and tables (or equivalent depending on the data source) from every data source and stores it in SQLite. In the case of a relational database, information regarding primary key and foreign keys is also extracted if possible, as explained in Section 4.4.1.


Step 1/4

Please, enter database url, database model and a name to be assigned to the database you want to connect.
Note that the 'database name' field does not appear for data sources where a specific database must be selected within the server.
In this case, you must add the database you intend to connect in the url: x.x.x.x/database
You must test if it is possible to connect to the datasource using the inserted details. To do this, select a datasource in the list and click 'Test Selected DS Connection' button.
You cannot continue the configuration process while there are datasources with failed connections or no connection attempts.

Model: HIVE
Data Source Name: costumers
URL: localhost:9083

If the database requires user and password, please provide them here, otherwise leave these fields blank.
Make sure the user has enough permissions to execute queries.

User: bsilva
Password: •••••••••••

Inventory in localhost:27017 (MongoDB)
employees in localhost:3306 (MYSQL)
social state in localhost:9042 (Cassandra)

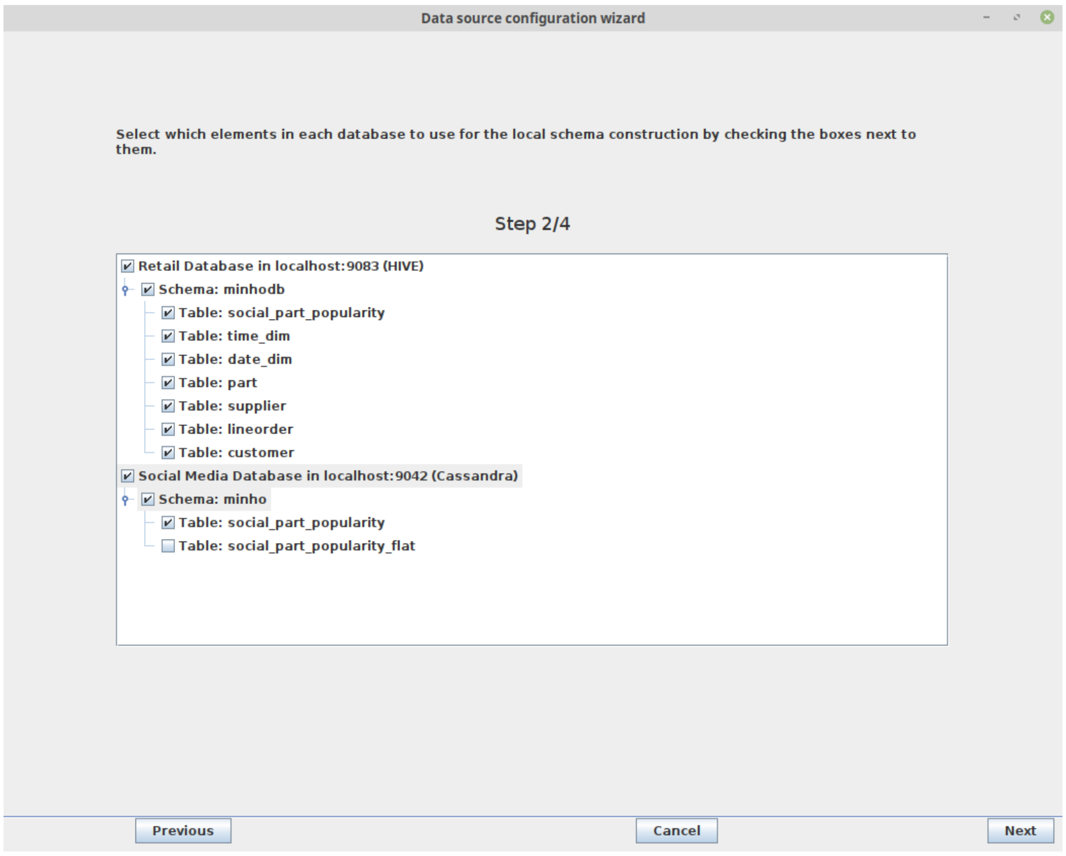
Success

Error: All host(s) tr

1
2
3
Add Data Source
Remove Selected Data Source
Test Selected DS Connection
Import Data Source From Project...
Previous
Cancel
Next

Figure 5.2: Project Creation Wizard - Adding new data sources and testing the connections.

A filtering window is available right after the *data source configuration window* that allows users to filter out any tables and schemas they do not wish to include in the project. Figure 5.3 shows the schemas and table selection interface. The organization is a hierarchy, starting with the data source, then its schemas and then the tables in the schema. Checking or unchecking an higher element in the hierarchy will also check or uncheck its childs.


Data source configuration wizard

Select which elements in each database to use for the local schema construction by checking the boxes next to them.

Step 2/4

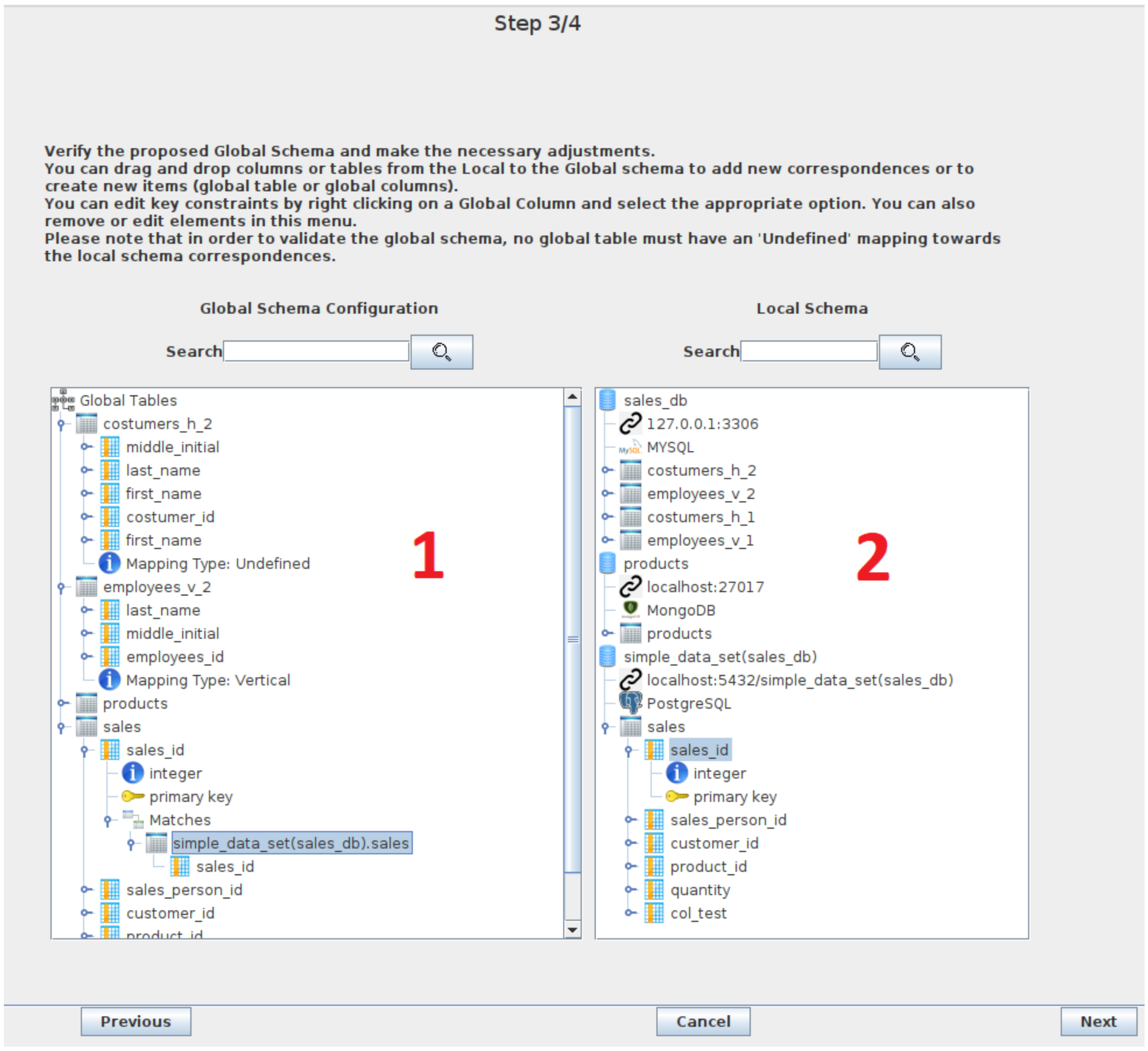
- ☑ Retail Database in localhost:9083 (HIVE)
  - ☑ Schema: minhodb
    - ☑ Table: social_part_popularity
    - ☑ Table: time_dim
    - ☑ Table: date_dim
    - ☑ Table: part
    - ☑ Table: supplier
    - ☑ Table: lineorder
    - ☑ Table: customer
- ☑ Social Media Database in localhost:9042 (Cassandra)
  - ☑ Schema: minho
    - ☑ Table: social_part_popularity
    - ☐ Table: social_part_popularity_flat

Figure 5.3: Project Creation Wizard - Selecting schemas and tables in a Local Schema.

5.3.2 Global Schema Configuration

In the global schema editing wizard, a Data Administrator is responsible for verifying the global schemas created automatically by the system (details regarding global schema generation are in Section 4.4) and editing the schemas in order to create a consistent global schema suited for the analytical needs of business analysts. A global schema can be edited by adding/removing mappings between local and global entities, deleting global tables or columns, creating new tables (but creating new columns is not allowed) and establishing primary key and foreign key constraints, an important configuration needed for the star schema creation.

Figure 5.4 shows the global schema editing wizard. Area 2 presents the local schema view previously created, and area 1 presents the global schema, both represented as an hierarchy using Java Swing's JTree class, which was extended and customized to allow the addition of custom images to facilitate reading and interpreting the different nodes of the hierarchy, such as databases, tables, columns and other elements in the nodes of the JTree. It is possible to expand nodes of the tree to see more information regarding each element. For example, expanding a data source node in the local schema tree will show its URL and model, while expanding a table will show its columns, and expanding columns will show its data type and constraint information (primary and foreign key information).



Verify the proposed Global Schema and make the necessary adjustments.
You can drag and drop columns or tables from the Local to the Global schema to add new correspondences or to create new items (global table or global columns).
You can edit key constraints by right clicking on a Global Column and select the appropriate option. You can also remove or edit elements in this menu.
Please note that in order to validate the global schema, no global table must have an 'Undefined' mapping towards the local schema correspondences.

Figure 5.4: Global Schema Creation UI.

Creating Correspondences Between Schemas

A column node located in the global schema contains a child node named 'Matches' with information regarding the correspondences of the local schema, as well as the mapping type. Column "sales_id" in area 1 in Figure 5.4 is expanded and shows the "Matches" node for this column that only has one correspondence to column "sales_id" in table "sales" from the local schema.

It is possible to drag columns from the local schema to a global column node (or the 'Matches' node beneath a global column) to establish a correspondence between the two schemas. As users edit the correspondences on each global column of a global table, the mapping type associated to that table is updated, which can be either horizontal, vertical or simple.

If needed, one can quickly create a global table from a local table by dragging a local table from the local schema tree and dropping it in the global schema tree. The new global table has the same name and all columns have correspondences to the corresponding local column, making this a simple mapping (recall mapping types at Section 4.4.2). A global column can also be created by dragging one local column and dropping it in a global table and a global column will be created with same name, datatype and primary key information, along with a correspondence to the local column.

Editing Elements And Validating Constraint Information In Global Schema

The generated global schema can be edited in several ways, by selecting one element (such as a table or column) and right click to open a menu that presents several options, depending on the selected element. For a table, it is possible to change its name, or delete it. For a column, options such as add primary key or add foreign key are also shown (or remove primary/foreign key, if the respective key already exists), as depicted in Figure 5.5.

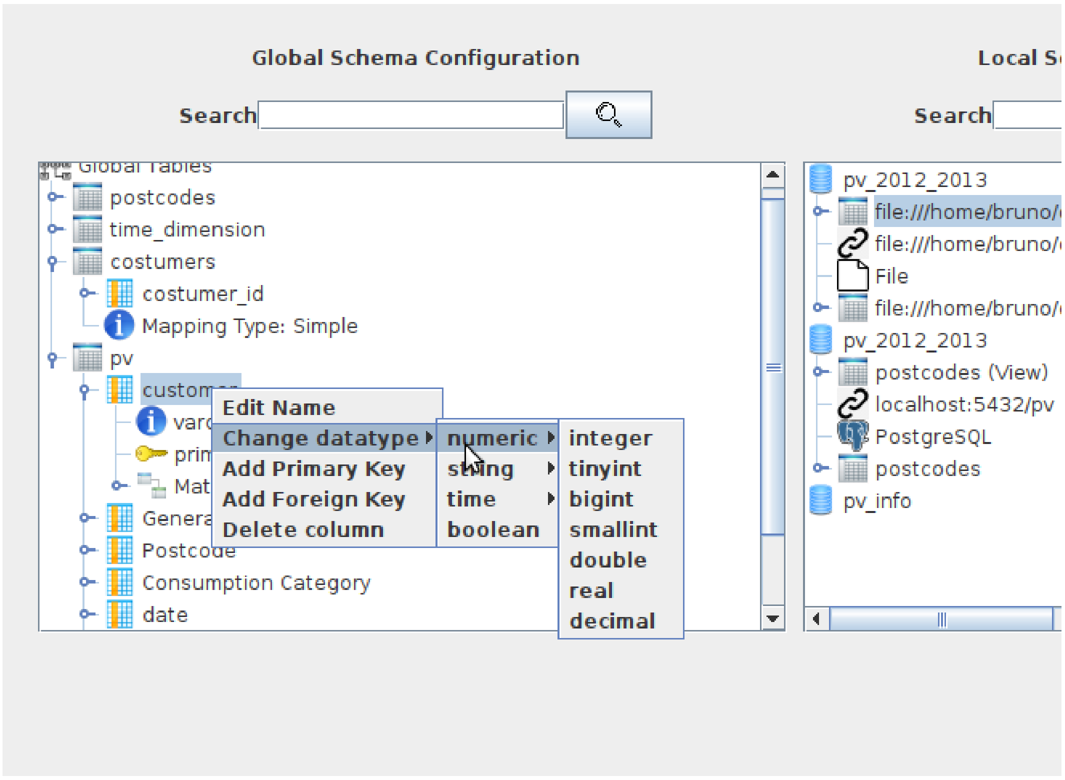


Figure 5.5: Global Schema pop up menu.

Foreign key and primary key information in the global schema should also be verified and corrected as this is important information for the system. One can add a primary key by right-clicking on a column in the global schema using the menu seen in Figure 5.5 and selecting "Add Primary Key", or add a foreign key by selecting "Add Foreign Key" in the same menu. When adding a foreign key reference, a small window will appear prompting users to select the table and column in the global schema to be referenced by the foreign key.

Searching Schema Elements

If the global or local schemas contain several elements, it is possible to search for a specific element, by typing in the corresponding search box on the top of the local or global schema. Only elements that contain a substring of the entered string will appear, until the user clicks the 'Erase search button' that will appear when a search is executed. If the user search in the local schema for "sales", then if there is only one table named 'sales_info', only that table and it's columns will be visible.

Changing Data Type

Some columns might have a data type that is undesirable or incompatible with the global or star schemas being created. Other scenario is when the data source is a raw file, as all columns are assigned the varchar data type (as mentioned in Section 3.5). To overcome this, EasyBDI contains an option that allows users to change the data type of a column in a global table, provided the original data type is compatible with the new data type. For example, date cannot be converted to a numeric data type in Presto, because the *cast* operator does not allow it. Users are also responsible to ensure that the data of a column whose data type was changed is compatible, such as changing a varchar to integer, as the user must ensure that all columns contain only digits in order to be converted to integer.

To change the data type of one column, select the global column and right click to open a menu, and select "Change Data type". When hovering on top of this option, 3 types of data types categories will be shown (string, numeric, time) and boolean data type is also present. Hovering on top of a data type category shows its data types, and selecting one will change the column to that data type, unless the original data type and the selected data types are incompatible, which will show an error message and the operation is canceled.

The information that a column table had its original data type changed is also stored in the Schema/Metadata storage, specifically in table "column_global", which stores information regarding each local table, seen in Figure 4.4. The column "is_datatype_different" is a boolean signaling "true" if user changed the global column's data type and false otherwise. Column "datatype_original" indicates the original data type this column had when it was created and column "data_type" stores the current data type of the column. These last tables have the same value if the data type was not modified.

In terms of query generation, whenever a column from a global table whose data type was changed is used in a query, a "CAST(<column> as <data type>)" must be used each time this attribute appears in the query, regardless of its location in the query. An example of a query made in EasyBDI containing attributes whose data types were changed can be seen in Figure 6.10, where the data type of the attribute "GG" changed from varchar to double. The resulting generated query can be seen in Listing 6.12

Adding Views Using SQL Code

It is possible to add a view over a local table in order to modify its organization. This feature could be useful when using a dataset whose structure is not compatible with multi-dimensional schemas or with the analytical purposes intended. The case study described in Section 6.2 shows an example where a data source had data organized in such a way that made it difficult to create relevant analytical queries. In these cases, an expert user can create code using SQL and the result of that query will be used as a virtual table, like a view in a relational database. Column information of a virtual table is obtained by analyzing the columns in the result set of the query, as well as data type, however, constraint information is not known.

A local table view is created by selecting a table in the local schema area of the global schema creation screen, and right-clicking on the table to select the "Create view for table using SQL...", which will open a window in which users can place SQL code and then run the code directly in Presto. If the code runs without any error, the output will be used to create a local table view, containing the same table name as the original table with the suffix "(view)". This virtual table is stored in the local schema, along with the information that this is virtual, table by saving in the SQL code in the column "sql_view_code" from table "table_data" Schema/Metadata storage, seen in Figure 4.4.

Case study 2 will show an example of a local table view creation, illustrated in Figure 6.8.

With this feature, users can change the structure of an object in the local schema without changing the actual object. Some examples of possible changes would be, for example, removing columns, or adding new columns (by performing joins for example), pivoting or unpivoting data, or creating formulas to change the values of columns.

However, this approach may increase the execution time of analytical queries that use virtual files or other objects created in this way, because the code that generates the virtual object may be complex and operate over an object with a lot of data, and so analytical queries may take even longer to execute because the virtual object needs to be created first and only then the analytical query is executed. Even for OLAP scenarios where quick response time are not as important as OLTP, queries that take too long may be counterproductive. It is up to the user to analyze the query execution time and the capabilities of the Presto cluster being used, as complex queries may struggle on lower end clusters.

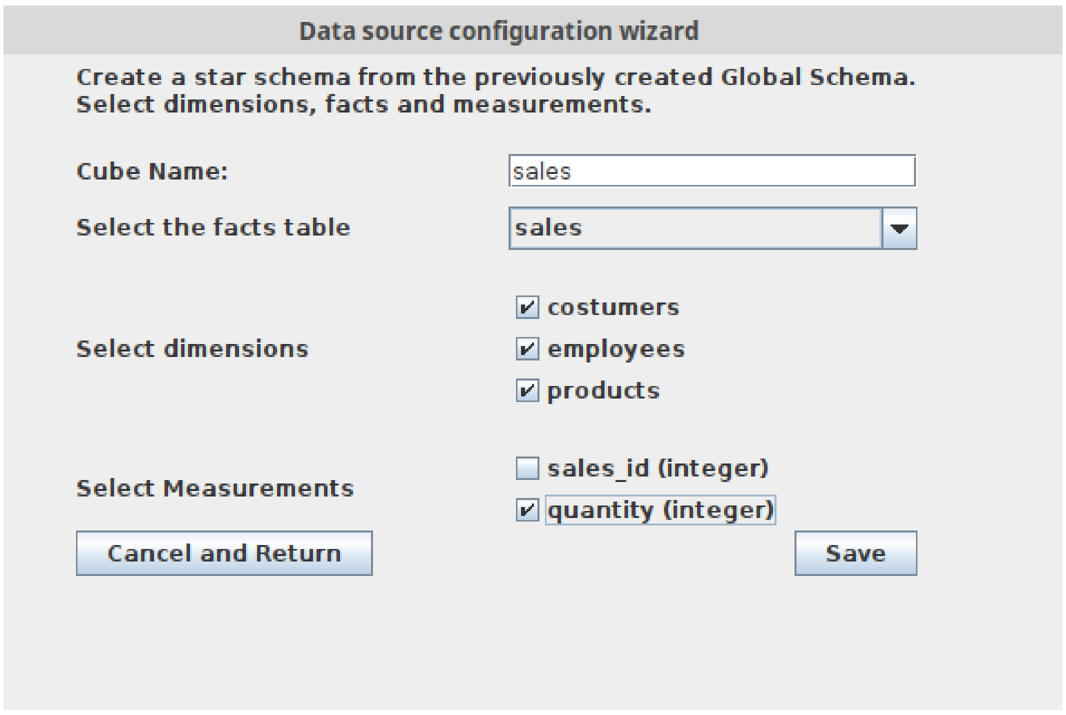
Validate Mappings And Finishing The Global Schema

To conclude the global schema creation process, every global table must have a defined mapping type. The mapping type of each global table is seen in the last node on the list of child nodes of each global table, as seen in Figure 5.4. If one or more tables have an "undefined" mapping, the system will not allow the user to continue to the next step, and correspondences must be readjusted such that the system can detect one of the three mapping types. The list of correspondences of each column of the global table is below the node "Matches". Figure 5.4 shows a local table (highlighted in blue) and its columns that are correspondences to the global column "sales_id".

The system is able to automatically detect vertical or horizontal partitioning between tables in the local schema (even if the partitioned tables are stored in distinct data sources), or no partitioning at all. In order to efficiently detect these mappings, the algorithms use strict rules. For example, horizontal mapping requires that all partitioned tables share the same column names. However, the user may wish to define a horizontal mapping using tables whose columns have slightly different names for example. In order to allow these types of scenarios, the system offers the option to override the detected mapping type and set a mapping according to user's choice. It is still required that minimum criteria are met, such as the correspondences between local and global schema being of compatible data types, the same number of column in all partitioned tables in the local schema in case of horizontal mapping, and a foreign keys referencing a primary key for a vertical mapping.

5.3.3 Star Schema Configuration

To finish a project configuration, at least one star schema (also designated as multidimensional cube) must be available to perform queries. Therefore, the user must create one by selecting global tables as dimensions and a fact table while also specifying which attributes are measures. The interface which allows the creation of a star schema from a global schema is shown in Figure 5.6. The star schema must be named and a global table must first be selected as the facts table. Afterwards every other global table can be chosen as a dimension table by checking it. Note that multiple star schemas can be created from a single global schema. The system will offer suggestions of measures to the user, as they tend to be numeric attributes that are not foreign keys. Selection of at least one dimension and measure attribute is mandatory to proceed. Clicking on the "Save" button will create the star schema and store both the global and star schema to the SQLite, thus concluding the project creation, which is now ready to be used for analytical queries.


Data source configuration wizard

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

Cube Name: sales

Select the facts table: sales

Select dimensions:
- ✓ costumers
- ✓ employees
- ✓ products

Select Measurements:
- ☐ sales_id (integer)
- ✓ quantity (integer)

Figure 5.6: Project Creation Wizard - Creation of a Star Schema from a Global Schema.

5.4 EasyBDI - Edit Projects

Any previously created project can be modified in order to add new data sources, change global schema or add a star schema. An unfinished project that does not have a finished local, global and star schema must have its configurations concluded by selecting this option. When editing a project, the wizard interfaces and configuration processes are the same as the ones explained in Section 5.3, except that previous configurations are shown to the user instead of starting over. The difference on each of the wizard interfaces when editing a project is detailed as follows:

- In the data source configuration interface, any data source previously added is automatically added to the list in area 2 of Figure 5.2 and cannot be removed, and only new data sources can be added. Old data sources do not need to perform a connection test and are already marked as ready.
- In the schema/table selection, any schemas and tables of new data sources are also present, but any previous schema or table that was used for the global schema are greyed out and cannot be unchecked.
- If a previous global schema was created in the project, no automatic creation of global schema is performed, even if new data sources, schemas or tables were added to the local schema. User must edit (or not) the global schema in the project.
- If the global schema is changed, then any star schema previously created may not be correct, as references to some global tables may no longer exist. Therefore, any existing star schema mapping to the old global schema is deleted and a new ones must be created.

5.5 EasyBDI - Execution Of Analytical Queries

To execute analytical queries a project must first be selected from the main menu, but at least one star schema must be available in that project, otherwise it will not be possible to make analytical queries.

When clicking the "Execute queries" button a new interface is shown as seen in Figure 5.7, which contains the star schema and a set of areas that will be used to create and submit queries. The option identified as 1 allows to switch between the different star schemas created. The areas in the middle (2 - 10) allow to create and visualize queries. The tree area identified as 2 presents a top node with the facts table and its measures. The other columns from the facts table are omitted. Another node contains the dimensions, and each dimension has child nodes with its columns, each with information regarding its data type by expanding a certain column's child nodes.

To create queries, columns and measures must be dragged to the areas 3 to 8. Each area has a specific role on building the final query. The logic behind the generation of the SQL query Presto executes is explained in detail in Section 4.6. Each area contains a small question mark, and when the mouse hovers it, a tooltip with a brief explanation of what the area will do for the query and what should be dragged is presented in order to assist the user.

Columns and measures from the star schema in area 2 must be dragged to specific areas. For example, columns can only be placed on areas 3, 4, 5, 6 and 7. Area 8, is used to create filters using aggregations, and only elements present in area 5 can be dragged to this area. More details will be discussed later.

Button "Execute Query" in area 10 starts the query generation process which is then sent to Presto for execution. Results are displayed in area 9. Button "Clear All Fields" in area 10 can be used to quickly remove any elements in the interface and start a new query creation from scratch.

5.5.1 Adding Rows Or Columns

When dragging an attribute to area 3 or 4, the attribute dragged will show on the respective area. All attributes in each area are grouped by tables, as seen in Figure 5.8. Measures are

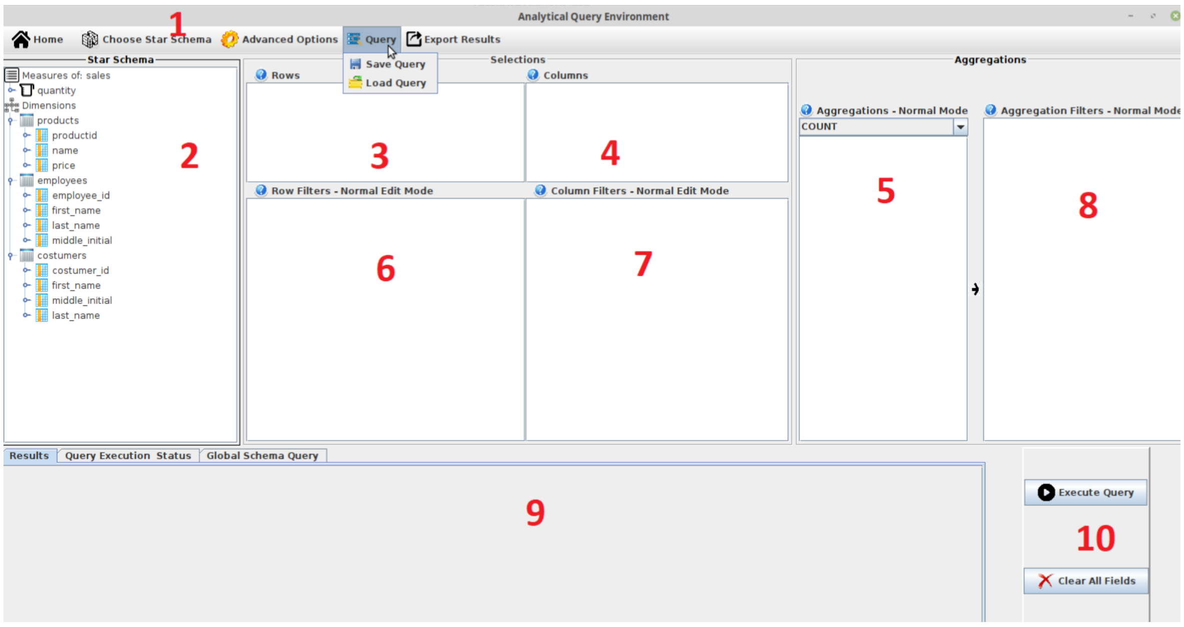


Figure 5.7: User Interface that enables the creation, execution and visualization of the results of analytical queries.

not allowed in these two areas, and can only be placed in area 5.

Area 3 is used to create rows in a query result, and every column placed here will be added to the *select* statement. Attributes placed in area 4 will be used to create columns on the query output, therefore, each distinct value will be pivoted. On area 3 (named rows), a menu will appear next to the cursor by selecting one attribute and then right-clicking. This pop up menu can be seen in Figure 5.8, and it allows the removal of attributes or the addition (or removal) of a sorting order for the selected attribute.

Attributes placed in area 4 are used to create columns on the query output, therefore, each distinct value will be pivoted. This area also has a similar right click menu on a selected attribute, but it is only possible to remove the attribute. Due to performance constraints, a maximum of 3 attributes must be placed on this area, in order to avoid heavy queries, as the process used to pivot values (explained in Section 4.6.3) can become expensive.

5.5.2 Adding Rows With Aggregations

The area 5 is used to define the aggregate functions using dimension attributes and measures. This is the only area where measures can be placed.

When an attribute is placed here, the aggregate function currently selected in the selection box above area 5 will be associated with the dropped attribute. Figure 5.9 exemplifies this, with attribute "price" with aggregate function "count" and measure "sales" with aggregate "sum". Measures dropped here always appear before columns from dimensions. The same measure may be added multiple times but with different aggregate operations, so, for example, measure "price" could be added again but should contain a different aggregate function, otherwise an error message will inform the user of this.

A right click menu is also present in this area, and contains the options to delete and add sorting options, as well as other options, as seen in Figure 5.9. It is possible to change the

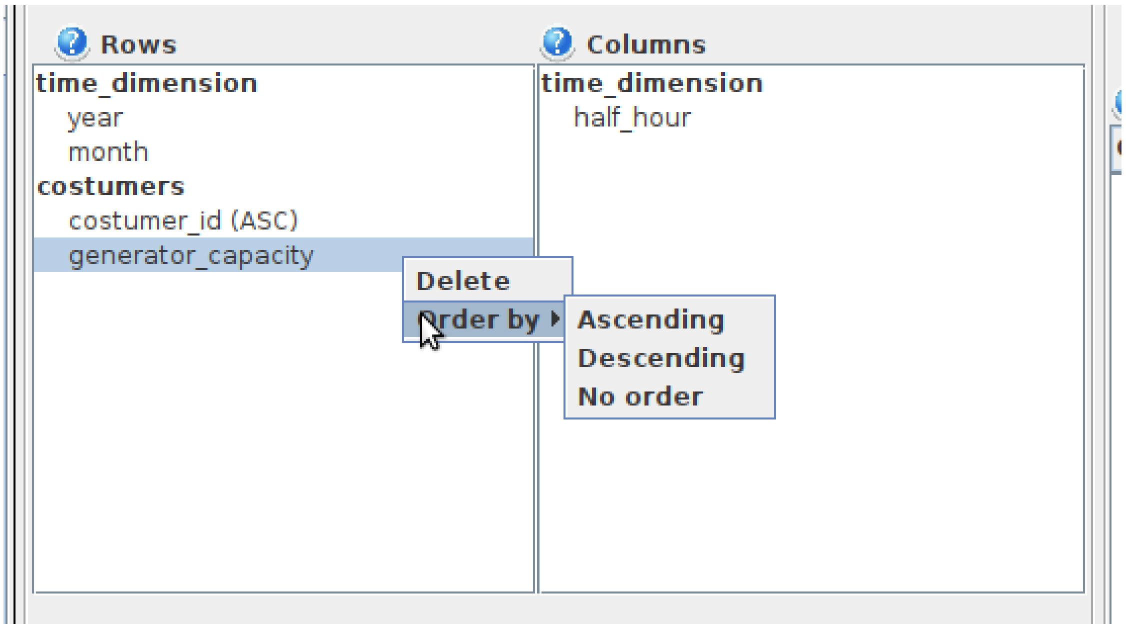


Figure 5.8: Example of attributes placed in the column and rows area of EasyBDI query interface.

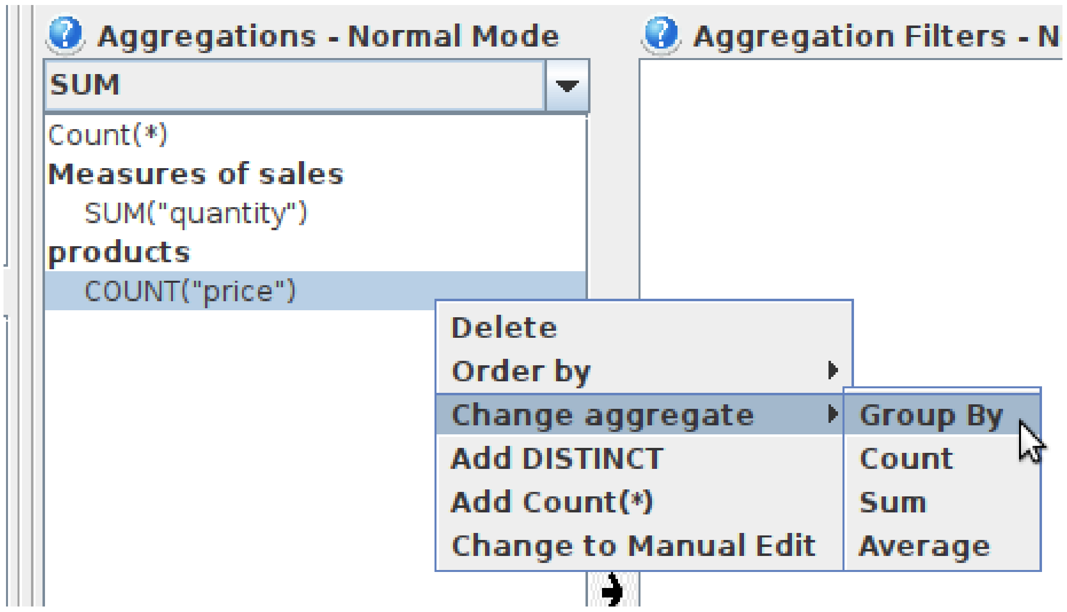


Figure 5.9: Example of attributes placed in the aggregations area of EasyBDI query interface.

---

aggregate function of the selected attribute, add a *count (*)* to the query (seen in Figure 5.9), or add a *distinct* key word in the aggregate function in order to count only distinct elements.

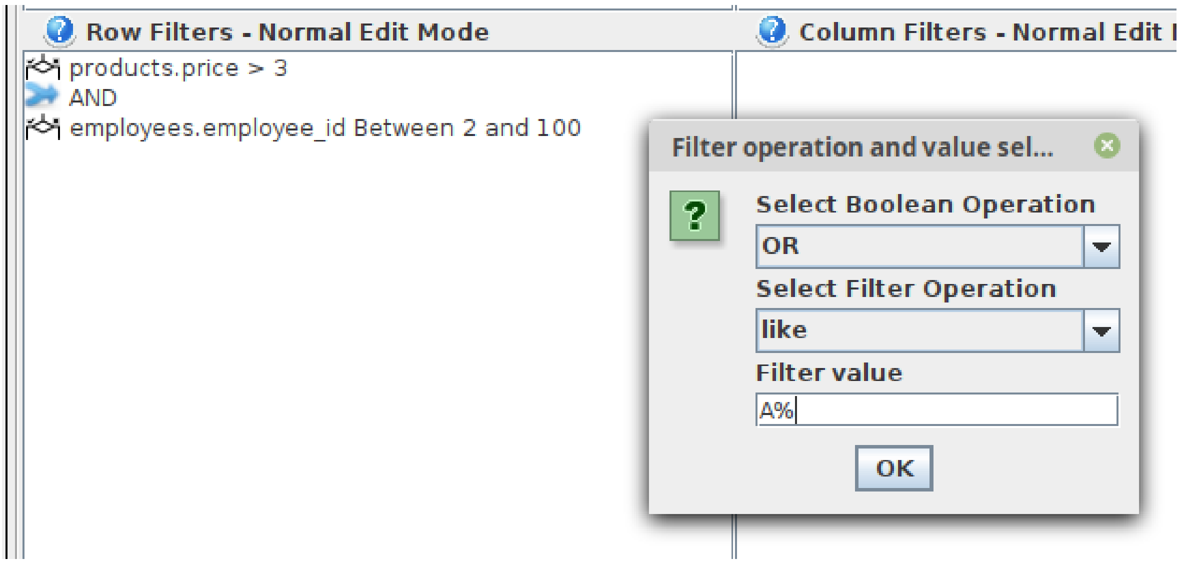
5.5.3 Adding Filters

Areas 6 and 7 allow the creation of filters over rows and columns, respectively. Whenever an attribute is dragged to one of these areas, a prompt will appear asking the user to specify an operator and the value to create a condition using the dragged attribute. This prompt can be seen in Figure 5.10. The system allows only the use of compatible operators. For example, string-like data types can use operators such as *like*, *between*, *!=* and $=$ but not $>$ or $<$ and their variants, while numeric-like data types are compatible with most comparison operators, with a few exceptions such as the *like* operator. For the *between* and the *not between* operators, users must add an *AND* between each value, otherwise the system will alert them to do it.

No single quotes need to be added for string-like or time-like data types, the system will add them in the final query, unless the user already did. Similarly, for time-like data types, users do not need to add the data type when adding a time or a date (such as time '12:30:00') because the system will also add the data type.

If another attribute is dragged and dropped in the filters area while at least one condition is already present, then a boolean operator must be added between the conditions, as depicted in Figure 5.10. The operators that can be added between conditions are *AND* and *OR*.

After a condition is created correctly, it is displayed in area 6, 7 or 8, depending on where it was created. These areas are JTrees and each condition is a node, and between conditions a node containing a boolean operation also exists. Figure 5.10 shows for example a first condition filtering for prices bigger than 3, a node with an *AND* operator and a third node with a second condition showing employees with id between a certain range. The figure also shows an example of a filled form for the creation of a new condition and a selection of a boolean operator. When confirmed, a new node will be added with the selected boolean operator (in this example, the operator *OR* is used) and another node with the condition created.


Row Filters - Normal Edit Mode

products.price > 3
AND
employees.employee_id Between 2 and 100

Column Filters - Normal Edit

Filter operation and value sel...

Select Boolean Operation: OR

Select Filter Operation: like

Filter value: A%

Figure 5.10: User Interface showing the addition of a filter.

To create nested conditions, an attribute must be dropped on top of an existing condition in the filters area, and then fill out information in the prompt form seen in Figure 5.10 and after confirmation the new condition will be displayed in the area as a child node of the parent node condition. An example of a query creation using EasyBDI containing nested conditions can be seen in Figure A.2.

All of the mentioned features and instructions apply to all filter areas, but each one has a different purpose. Area 6, rows filters creates filters applied to the rows on the query results. In other words, this means that conditions here created will be used in the *where* clause, as discussed in Section 4.6.4. Also mentioned in the referenced section, is that any tables from attributes used in the rows filters (area 6) must be present in the rows (area 3), or aggregations (area 5), otherwise the query cannot be executed and an error message will be shown detailing this issue.

Conditions created in area 8 are used to create aggregation filters, hence only elements containing aggregation functions can be dragged to this area, and this is why the user is restricted to using elements from area 5 (where aggregations are created) and not from the star schema in area 2. Conditions here created are used in a *having* clause, also discussed in Section 4.6.4.

Finally, conditions created in area 7 are used against the attributes in area 4, whose values will be pivoted, therefore these filters are used in a query that will fetch all distinct values to form columns in the final query, as discussed in Section 4.6.3 and seen in query depicted in Figure 4.9. Any attribute placed in the filter columns (area 7) must be present in the columns area (area 4).

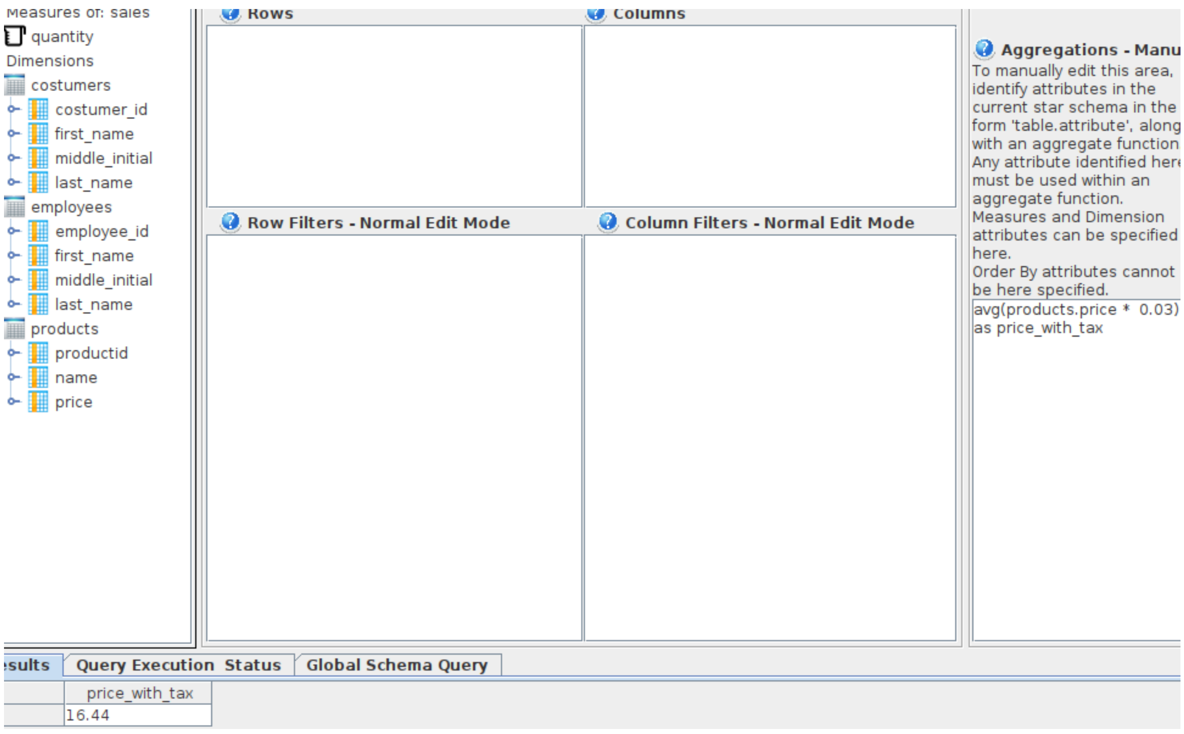
For the query generation, each node is iterated in order to generate an SQL filter. Each node contains additional information, transparent to the user, that is used for query generation, such as if the attribute needs to be casted to a user defined data type (see Section 5.3.2 for details). If a node has childs, then parenthesis need to be opened, then each child node converted to SQL query and when all child nodes are traversed, parenthesis are closed, thus creating a valid SQL nested condition.

5.5.4 Manually Editing Filters/Aggregations

Some of the areas in Figure 5.7 offer the option to manual type part of the SQL query. This feature can offer more options to expert users and overcome most limitations that creating queries using an user interface may cause. The areas that allow this option is the aggregations area (5) and all filters area (6,7 and 8). This mode can be accessed using the right click menu on each of these areas and selecting the "Change to Manual Edit". When using this mode, users must obey certain rules in order to ensure that the SQL query is generated correctly. This allows to write conditions and apply operations that cannot be specified using the UI. It can also be used by users that do not feel comfortable creating complex filters with several nested conditions.

In area 8, filters aggregations, it is mandatory that any manually written SQL code written contains attributes with aggregate functions, and no attribute without any aggregate must be added. For example, it is allowed to add two or more attributes inside the same aggregate function. Sortings cannot be done in this mode. When the manual mode is activated for this area, a small text is displayed warning the user of these restrictions. Figure 5.11 depicts a query containing an aggregate function written using manual mode.

The code written by the user in any of these areas is integrated into the final query, but before, all text is parsed and validated. While parsing, any mentioned attributes and their respective tables are extracted from the text by name because the logic used to generate



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. Order By attributes cannot be here specified.

```
avg(products.price * 0.03)
as price_with_tax
```

	price_with_tax
	16.44

Figure 5.11: User Interface showing a query with an aggregate function created using manual mode.

queries requires the object containing the attribute's metadata in order to correctly generate the inner queries that retrieve data from the local schema. In order to detect elements in the user written code, any attribute referenced must be written with a qualified name using 'table.column'. Users are also responsible to escape any needed elements in the written query, as EasyBDI will not "auto-escape" any elements from manual queries that need to be escaped. See, for example, query in Listing 6.4 (generated from the interface configuration in Figure 6.5), which contains in the *select* clause an aggregate operation created using manual mode but none of its elements are escaped because the user did not add any escape characters while writing the code, while the other attribute present in the clause was placed in the rows area, and is therefore escaped automatically by EasyBDI.

5.5.5 Processing And Showing Results

Results of queries are shown in a table at zone 9 of Figure 5.7. When a query is submitted by the user using the submit button in area 10, a valid SQL query will be generated according with the elements on the interface (as described in Section 4.5.1) and then sent to Presto in order to execute the query. When the query finishes and returns the results, these must be processed and placed in the table at zone 9. To populate the results table, the names of each column on the output are first placed in the first row of the table, the header, followed by all records which are iterated row by row and placed in the corresponding column. The first column of the results table indicates the line (excluding the top row).

When the submitted query contains more than one attribute whose values were pivoted, the resulting columns are grouped and more layers are created in the table header in other to display the results in a more intuitive and clean way, as opposed to the query output returned by Presto where such column names are the concatenation of the different values that were used to create the column, as explained in Section 4.6.3. To be able to customize the header of a table in java an external package is used, named *JBroTable* [1], which offers a ready to use implementation of the java standard JTable with more functionalities, including the addition of more than 1 row to the header and grouping of multiple columns on the header with the same name into one spanning more than 1 columns in the table. Figure A.1 shows a query containing 3 pivoted attributes and the results in EasyBDI.

The process of query execution may vary in time depending on the complexity of the submitted query. While this is happening on the background,it is important to inform the user regarding the status of the query, as it may fail or take longer than expected, depending on the complexity of the query and the capacity of the cluster where Presto is installed. For this effect, a loading screen animation is shown from the moment the "execute query" button is clicked, simulating system activity, along with a message describing the current step of the query processing: generation of the SQL query, execution of the query by Presto, or results handling to insert on the table. The loading animation stops only when either an error occurs or when all the results are received and placed on the user interface. Examples of possible errors that might happen are:

- **Unable to connect with Presto** - The query was sent to Presto, but Presto connection does not exist, probably because it is not running or the location of the Presto Coordinator node was changed.
- **Presto returns no results** - This might happen for several reasons, either because one or more of the data sources could not be reached or because the query is too heavy and takes a long time. A query returning 0 rows is not considered an error. If a Presto worker node exceeds memory, it will crash and stop. If that node is also the coordinator, the system will be waiting indefinitely for the query, as Presto does not return any error when memory exceeds. To prevent this, each query has a maximum time of 3 minutes on Presto to execute, and after that is canceled.
- **Presto returns an SQL exception** - This happens when the SQL query contains a syntax or semantic error.

The first 2 categories of errors are not directly dependent on the system, but some of the errors in the last item may be directly or indirectly caused by the system, namely on incorrect SQL query generation due to syntactic or semantic errors. Efforts were made to reduce the occurrence of these errors as much as possible.

5.6 EasyBDI - Other Features

EasyBDI presents several features that can aid and improve the users tasks.

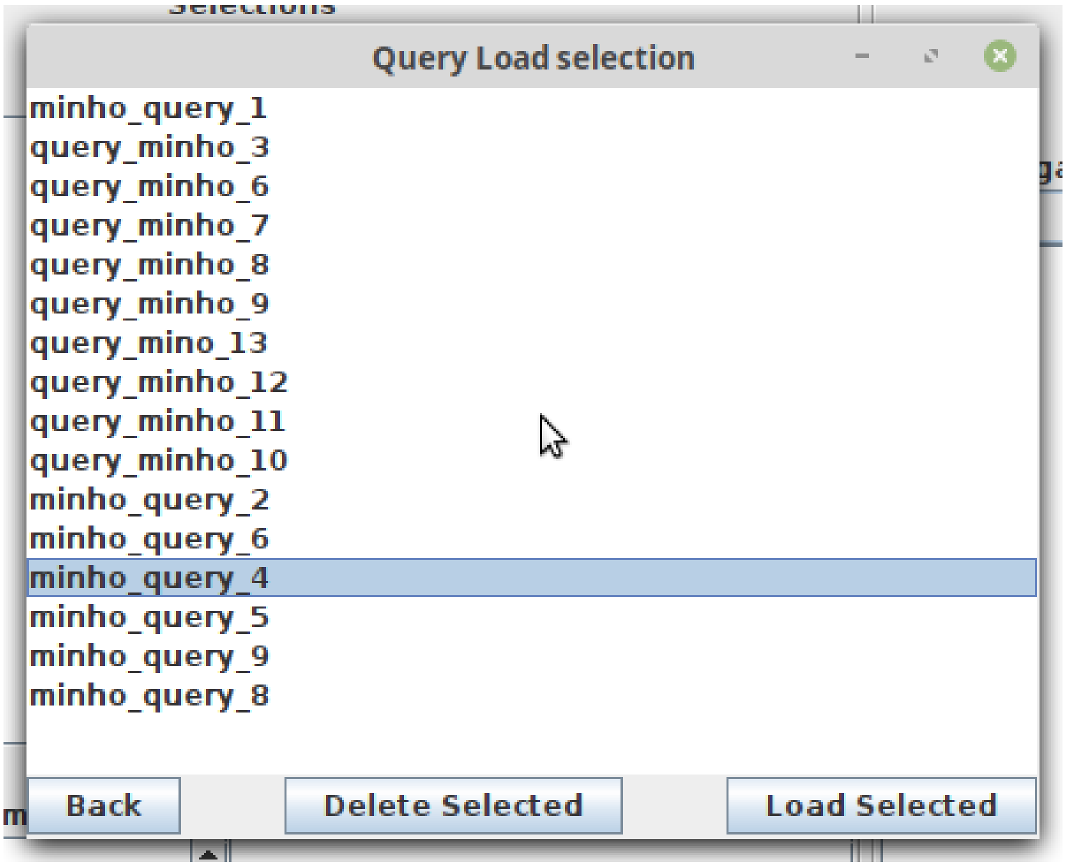
5.6.1 Saving And Loading Queries

Saving a query allows to quickly submit a previously used query into the system instead of having to redo it via the interface, which is very helpful for queries that need to be run

[1]https://github.com/Qualtagh/JBroTable

frequently. Internally, the state of the different areas in the interface (rows area, columns, filters and aggregations) are saved to the SQLite module storage when selecting the "save query" option inside the "Query" menu at the top. A name must be given to the query before it can be saved. Saved queries are associated to each project and to each star schema. Therefore, it is possible to have 2 queries with the same name in the same project but belonging to different star schemas.

To load a query previously saved, select the "Load Query" option inside the "Query" menu at the top, and select one of the queries previously saved from the list depicted in Figure 5.12. It is also possible to delete a previously saved query.

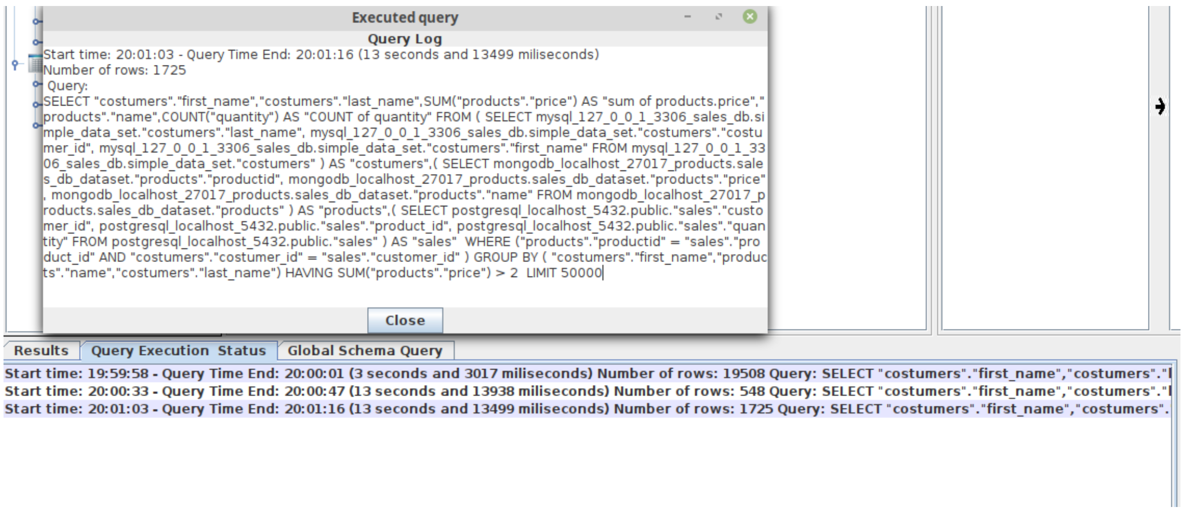


minho_query_1
query_minho_3
query_minho_6
query_minho_7
query_minho_8
query_minho_9
query_mino_13
query_minho_12
query_minho_11
query_minho_10
minho_query_2
minho_query_6
minho_query_4
minho_query_5
minho_query_9
minho_query_8

Figure 5.12: User Interface of a list of previously saved queries in EasyBDI.

5.6.2 Query Logs

Each query submitted through the system, is logged and added to a list of logs that can be consulted by selecting the "SQL Query" tab at the top of the query results panel. These logs contain the SQL query sent to Presto, the time the query was submitted, query status (failed or successful) and the elapsed time since submission until results were received from Presto, processed and shown in the interface. Figure 5.13 shows as an example a 3 queries in the list of executed queries. By double-clicking on a query in the list, a window opens showing the full query, as seen in the same figure. In this window, the SQL code can be copied and used somewhere else. It is also possible to export one or all logs in the list to a text file. Saving logs of queries is a similar feature seen in many programs that interact with DBMS, such as MySQL Workbench or PGAdmin and allows users to quickly analyze the last performed action on the system and their status.



```
Start time: 20:01:03 - Query Time End: 20:01:16 (13 seconds and 13499 miliseconds)
Number of rows: 1725
 Query:
SELECT "costumers"."first_name","costumers"."last_name",SUM("products"."price") AS "sum of products.price","
products"."name",COUNT("quantity") AS "COUNT of quantity" FROM ( SELECT mysql_127_0_0_1_3306_sales_db.si
mple_data_set."costumers"."last_name", mysql_127_0_0_1_3306_sales_db.simple_data_set."costumers"."costu
mer_id", mysql_127_0_0_1_3306_sales_db.simple_data_set."costumers"."first_name" FROM mysql_127_0_0_1_33
06_sales_db.simple_data_set."costumers" ) AS "costumers",( SELECT mongodb_localhost_27017_products.sale
s_db_dataset."products"."productid", mongodb_localhost_27017_products.sales_db_dataset."products"."price"
, mongodb_localhost_27017_products.sales_db_dataset."products"."name" FROM mongodb_localhost_27017_p
roducts.sales_db_dataset."products" ) AS "products",( SELECT postgresql_localhost_5432.public."sales"."custo
mer_id", postgresql_localhost_5432.public."sales"."product_id", postgresql_localhost_5432.public."sales"."quan
tity" FROM postgresql_localhost_5432.public."sales" ) AS "sales"  WHERE ("products"."productid" = "sales"."pro
duct_id" AND "costumers"."costumer_id" = "sales"."customer_id" ) GROUP BY ( "costumers"."first_name","produc
ts"."name","costumers"."last_name") HAVING SUM("products"."price") > 2  LIMIT 50000
```

```
Start time: 19:59:58 - Query Time End: 20:00:01 (3 seconds and 3017 miliseconds) Number of rows: 19508 Query: SELECT "costumers"."first_name","costumers"."l
Start time: 20:00:33 - Query Time End: 20:00:47 (13 seconds and 13938 miliseconds) Number of rows: 548 Query: SELECT "costumers"."first_name","costumers"."l
Start time: 20:01:03 - Query Time End: 20:01:16 (13 seconds and 13499 miliseconds) Number of rows: 1725 Query: SELECT "costumers"."first_name","costumers".
```

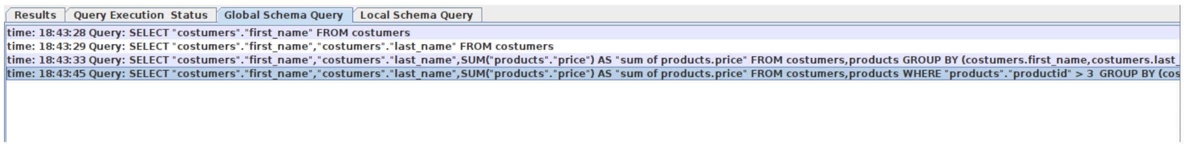
Figure 5.13: Executed query Log list, with a window showing the complete log of an executed query. Each time a query is executed, the SQL code is added to the list along with the star and end time, as well as its duration and status.

Experienced users may like to know what is the query that will be generated from Presto before it is executed as the elements are being configured in the interface. For this purpose, two other logs exist, one that shows the complete generated SQL query by the system for the local schema (with the inner queries for the local schema data retrieval), depicted in Figure 5.14b and another one that shows only the query over the global schema, without inner queries, depicted in Figure 5.14a. By default, only the global schema query is enabled, but this can be changed in the "Advanced Options" menu at the bottom, which enables or disables global schema query logs and local schema query logs.

Each time a user drops an element in either area, the logs are updated. Any SQL code written using the manual edition mode does not count towards the queries displayed in these logs. For queries containing attribute pivots, the global schema log will present these columns using a "pivot (column name)" format, and for the local schema logs, the values of each column are obtained as specified in Section 4.6.3. As with the executed query log list, it is also possible to open a window with the complete log by double-clicking on a log in either list.

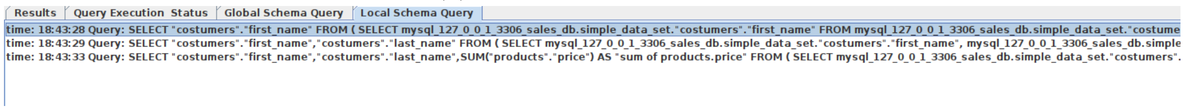
5.6.3 Export Results

It is possible to export query results to a csv file, by selecting the "export to csv" option below the "Export Results" menu at the top. If a query result containing multiple headers, then all header will be merged in a single column, and the name of each column will be the concatenation of the names of each row in the header. For example, if the query shown in Figure A.1 was exported to a csv file, the name of the first column in the csv file would be "a-O'Leary-Michael".



```
time: 18:43:28 Query: SELECT "costumers"."first_name" FROM costumers
time: 18:43:29 Query: SELECT "costumers"."first_name","costumers"."last_name" FROM costumers
time: 18:43:33 Query: SELECT "costumers"."first_name","costumers"."last_name",SUM("products"."price") AS "sum of products.price" FROM costumers,products GROUP BY (costumers.first_name,costumers.last_
time: 18:43:45 Query: SELECT "costumers"."first_name","costumers"."last_name",SUM("products"."price") AS "sum of products.price" FROM costumers,products WHERE "products"."productid" > 3  GROUP BY (cos
```

(a) Global Query Log List.



```
time: 18:43:28 Query: SELECT "costumers"."first_name" FROM ( SELECT mysql_127_0_0_1_3306_sales_db.simple_data_set."costumers"."first_name" FROM mysql_127_0_0_1_3306_sales_db.simple_data_set."costume
time: 18:43:29 Query: SELECT "costumers"."first_name","costumers"."last_name" FROM ( SELECT mysql_127_0_0_1_3306_sales_db.simple_data_set."costumers"."first_name", mysql_127_0_0_1_3306_sales_db.simple
time: 18:43:33 Query: SELECT "costumers"."first_name","costumers"."last_name",SUM("products"."price") AS "sum of products.price" FROM ( SELECT mysql_127_0_0_1_3306_sales_db.simple_data_set."costumers".
```

(b) Local Query Log List.

Figure 5.14: Query Log lists. Each time an element is dragged and dropped, the generated SQL query for the elements in the interface is added to the lists.

5.6.4 Protection Against SQL Injection

By allowing the user to manually write part of the SQL query, a security risk is introduced, namely, SQL injection attacks. These can be performed by more experienced users that inject SQL code with malicious intentions. For example, if the user uses the manual edition mode to create filters, he can write in the text box and submit a query similar to the one seen in Listing 5.1, which will execute the first query, but also a second one that deletes a complete table from the data source. All queries are run over Presto, and some data source connectors may have limited support for record or schema modification, but it is still possible to alter or delete tables or records using Presto.

Listing 5.1: Example of an SQL injection attack. This query will run in batch and therefore delete the table 'suppliers'

```
SELECT * FROM users WHERE user_id = 3; DROP TABLE suppliers;
```

Another option to protect against SQL injection is to use stored procedures [47], but that is not possible using PrestoSQL. Validating user input and escaping user written inputs such that no unwanted query is executed is also a viable option [47].

In EasyBDI, some mechanisms were implemented to prevent most forms of SQL injection attacks if the user uses any of the manual edition fields. All user writen text is analyzed for certain SQL commands that should not be present, such as *drop*, *where*, *having*, *delete* or *insert*. It also checks if the expression *"x = x"*, commonly used for many SQL injection attacks, is present. If any of these words or expressions are detected, the system does not submit any SQL query to Presto and shows a warning message asking the user to remove said words or expressions.

The implemented mechanisms were implemented as an example and to demonstrate how to prevent these attacks on EasyBDI. Further security mechanisms can be implemented in the future.

5.7 Considerations

EasyBDI is a system containing several features that allow non-expert users to perform analytical queries. This chapter presents a detailed description on how to use the system, while also explaining their purpose and the adopted solution that allows such features to work. An guide explaining how to create a project, from data source connection, global schema editing and star schema creation has been presented. The user interface for query creation was also presented and explained, detailing every area, menu, and tool available to create varied queries. Finally, additional features that may prove useful to potencial business analysts using this system are also presented and explained, such as a feature that allows to save and load queries, or export query results in csv file.

Overall, EasyBDI is a very complete system, presenting a variety of functionalities useful for any business expert or non-expert analyst.

Chapter 6

Evaluation and tests

To demonstrate EasyBDI's usefulness and versatility, this section presents two case studies. Appropriated data sources are created and populated with data from the datasets in both case studies, and then queried using EasyBDI. It is intended to test how expressive EasyBDI is by performing a wide range of distinct analytical queries. Using an interface to create queries takes away some of the expressiveness present on the SQL language to create queries, and so limitations may arise on the type of queries that can be created through EasyBDI. Some of the limitations found were reviewed and a solution was implemented in order to eliminate or at least reduce them.

The configuration used for these tests include PrestoSQL, version 330, configured to behave as both a coordinator and a worker on a single node configuration. The machine in which the tests were executed has 8GB of RAM, Intel Core i7-8550U at 1.80GHz with 4 cores processor, 250 GB SSD, running a Linux Mint 19.1 Cinnamon Operative System.

6.1 Case Study 1: Batch And Streaming OLAP

This case study uses the SSB+ system [48], which includes a dataset intended to be used as an OLAP benchmark for decision support systems. OLAP Benchmarks intend to evaluate the performance of decision support systems by creating a business oriented data model and generation of data to populate data sources, and a set of queries oriented towards business and a high degree of complexity [49].

The SSB+ benchmark [48] is an extension of the SSB benchmark in O'Neil et.al. [50], which in turn is an adaptation of the star schema presented by the TPC-H benchmark [1]. The SSB+ data model contains two star schemas, one for batch (OLAP) and the other for streaming OLAP. The batch OLAP contains facts and dimension tables used in the context of retail data. The streaming OLAP is based on social media data, which represents the popularity of the retail store (and of its sales and deliveries) on social media. Figure 6.1 shows both star schemas, with the facts tables being highlighted in rectangles with thicker edges.

Tables belonging to the batch OLAP star schema are located in Hive, while the facts data for the streaming OLAP is stored in Cassandra (a NoSQL database that uses a wide-column store model), and conceptual relationships exist between data stored in both systems, as represented also in Figure 6.1. Tables in Hive have a variable number of records, namely 2

[1]http://www.tpc.org/tpch/

000 (supplier and date), 30 000 (customer), 200 000 (part) and 6 001 215 (lineorder, the facts table).

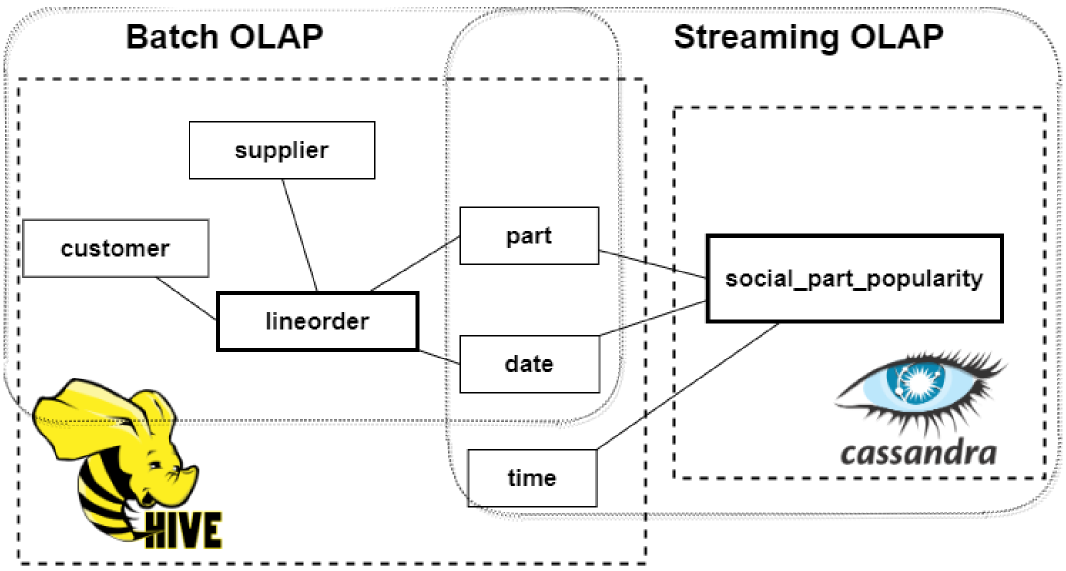


Figure 6.1: Case study: batch and streaming OLAP from SSB+ - overview of local schemas.

To recreate SSB+ benchmark environment, both Cassandra 3.11.7 and Hive 3.1.2 were installed on the same local machine, and the code available in the SSB+ project's Github repository [2] was used to create the data models, generate data and use it to populate the data sources. However, some of the steps required to add data to the facts table from the streaming OLAP star schema located in Cassandra involved the use of Kafka to produce data and Spark streaming to consume the generated data and store it in Cassandra. To simplify, a Kafka consumer was used to consume the data generated by the producer, and the data is inserted into the Cassandra facts table directly through a JDBC interface.

The benchmark developed in the work of Costa et.al. [48] also contains a flat version of both facts tables, meaning that attributes from dimension tables are merged on the facts table, creating one single table. These tables are mainly used for performance comparison purposes between the star schemas, one of the paper's contributions. On this evaluation, the focus is not on performance evaluation or comparison, but to test and demonstrate the use of EasyBDI on different use cases, so the flat tables are not considered. The amount of generated data for this evaluation is also not very relevant, however small data is preferred, because the cluster being used to for this experiment is modest.

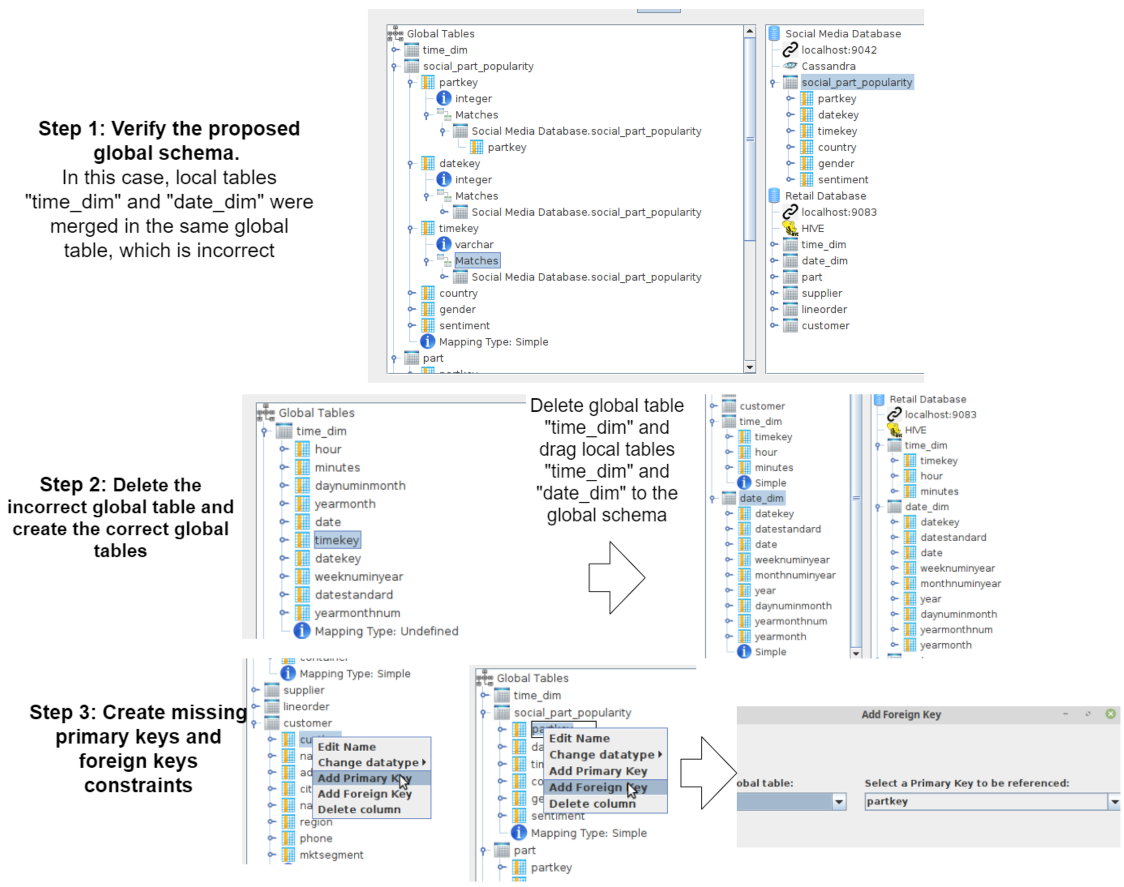
6.1.1 Project Configuration In EasyBDI

With the data sources populated with data, it is only a matter of connecting them to EasyBDI and create the global schemas and both star schemas, by configuring a new project. These steps are the same as the ones described in Section 5.3. Both Cassandra and Hive connection information are added, with Cassandra named as "Social Media Database" and Hive named as "Retail Database". After successful test connection to both data sources, the

[2]https://github.com/epilif1017a/bigdatabenchmarks

schema containing the tables relevant for this project are selected and the system proposes a global schema.

The proposed global schema was not entirely adjusted because the schema matching algorithm defined tables "time_dim" and "date_dim" as similar, which resulted in both local tables being merged in one global table. To solve this, the resulting global table was deleted, and correct global tables were created by dragging the original tables from the local schema to the global schema, as depicted in the second step in the global schema configuration process in Figure 6.2. All other tables were correct, and a simple mapping is assigned between the global and the local tables.

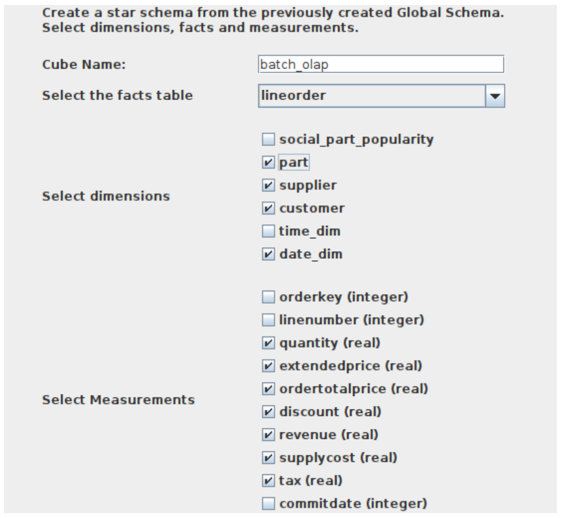


Step 1: Verify the proposed global schema.
In this case, local tables "time_dim" and "date_dim" were merged in the same global table, which is incorrect
Step 2: Delete the incorrect global table and create the correct global tables
Delete global table "time_dim" and drag local tables "time_dim" and "date_dim" to the global schema
Step 3: Create missing primary keys and foreign keys constraints
Global Tables
Social Media Database
localhost:9042
Cassandra
Retail Database
localhost:9083
HIVE
Mapping Type: Simple
Mapping Type: Undefined
Edit Name
Change datatype
Add Primary Key
Add Foreign Key
Delete column
Add Foreign Key
Select a Primary Key to be referenced:
partkey

Figure 6.2: Global Schema correction process in EasyBDI for the SSB+ dataset.

Primary key constraints were also specified in the global schema, because these constraints were not specified in the original data sources. Foreign key references were also created in the global schema, an important configuration needed to create both star schemas. This process can be seen in step 3 of Figure 6.2.

To conclude, the two star schemas need to be defined by selecting the facts, dimensions and measures from the elements in the global schema. When creating a new project in EasyBDI, only one star schema can be created in the configuration wizard, but after the project is created, more star schemas can be added to the project by selecting the option "Create Star Schema" from the main menu. Figure 6.3a shows the creation of the batch OLAP star schema, and Figure 6.3b shows the creation of the stream OLAP star schema.



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

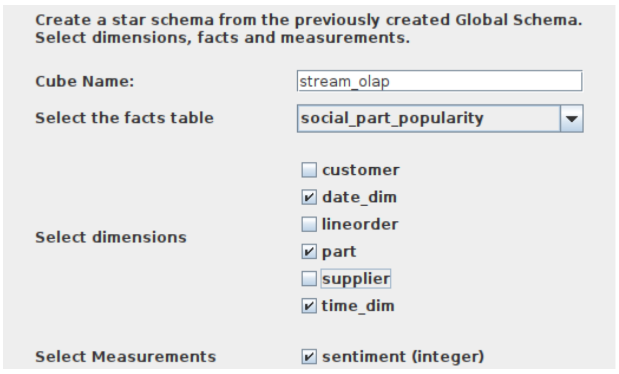
Cube Name: batch_olap

Select the facts table: lineorder

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

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

(a) Batch OLAP star schema mapping.



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

Cube Name: stream_olap

Select the facts table: social_part_popularity

Select dimensions:
- [ ] customer
- [x] date_dim
- [ ] lineorder
- [x] part
- [ ] supplier
- [x] time_dim

Select Measurements:
- [x] sentiment (integer)

(b) Stream OLAP star schema mapping.

Figure 6.3: Mapping of the Batch and Stream OLAP star schemas using EasyBDI's interface.

This concludes the configuration of this project, and queries can now be submitted using the created star schemas.

6.1.2 Queries Over The Created Star Schema Using SSB+ Dataset

The scripts found on the aforementioned Github repository from Costa et.al. [48] also contain SQL queries used for performance benchmarking. These queries were created in 2 sets: a set for the batch OLAP, and the other for the streaming OLAP. The first set contains 2 versions of each query, one for Hive, and the other for Presto. The objective is to translate the SQL Presto-ready queries to a configuration on EasyBDI's interface, such that the query generated created by EasyBDI is similar, if not identical, to the original query.

Earlier implementations of EasyBDI, were unable to reproduce some queries that contained filters or more complex aggregation operations, which led to the creation of a manual editing mode for filters and aggregations, as explained in Section 5.5.4. This addition made it possible to replicate and successfully execute on EasyBDI a higher number of queries from the ones available in the project's scripts than before the implementation of this feature.

Batch OLAP Queries

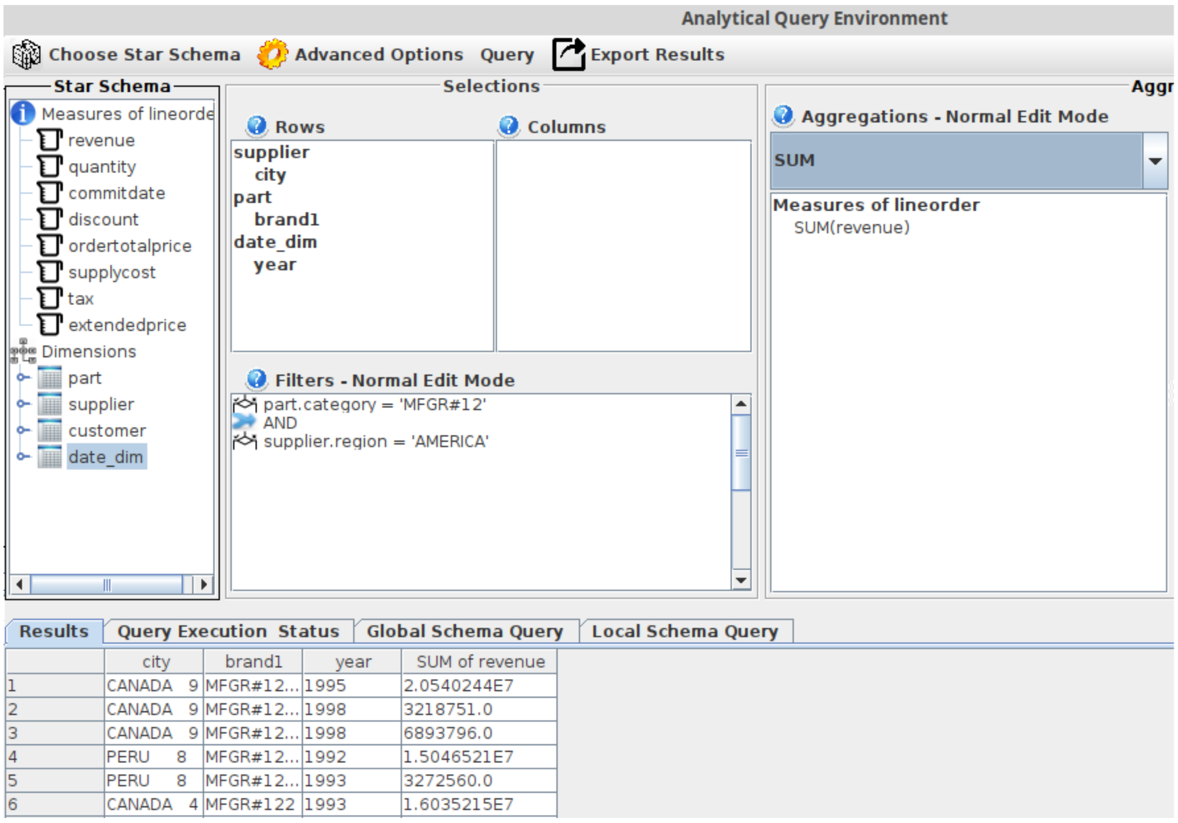
The set of queries from the batch OLAP star schema has 13 queries available. Some queries are very similar in structure and contain only small variations. For simplicity, only 4 queries are shown on EasyBDI, with 2 queries here presented as an example (queries number 2 and 4) and other 2 shown on the appendix (queries 9 and 13), as seen in Figures A.2 and A.3. For each query, it is presented a configuration via EasyBDI's interface, and the final SQL generated in order to better demonstrate and exemplify the system.

Starting with query 4, seen in Listing 6.1, it can be observed that this query contains a wide diversity of operations that are interesting to create in EasyBDI, namely joins, filters, aggregations, and order by and group by operations.

The filters and aggregations in this query can be specified using only EasyBDI's interface, as seen in Figure 6.4. Again, Listing 6.2 shows the query generated by EasyBDI.

Listing 6.1: SSB+ benchmark - Query 4 from the set of batch OLAP queries

```sql
SELECT SUM(lo.revenue), d.year, p.brand1
FROM lineorder lo JOIN date_dim d ON lo.orderdate = d.datekey
JOIN part p ON lo.partkey = p.partkey
JOIN supplier s ON lo.suppkey = s.suppkey
WHERE p.category = 'MFGR#12' and s.region = 'AMERICA'
GROUP BY d.year, p.brand1
ORDER BY d.year, p.brand1;
```



Rows:
supplier
  city
part
  brand1
date_dim
  year

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

Aggregations: SUM

Measures of lineorder
  SUM(revenue)

	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

Figure 6.4: Query 4 from the set of batch OLAP queries in SSB+ bechmark created using EasyBDI.

Listing 6.2: EasyBDI generated query: query 4 from the set of batch OLAP queries in SSB+

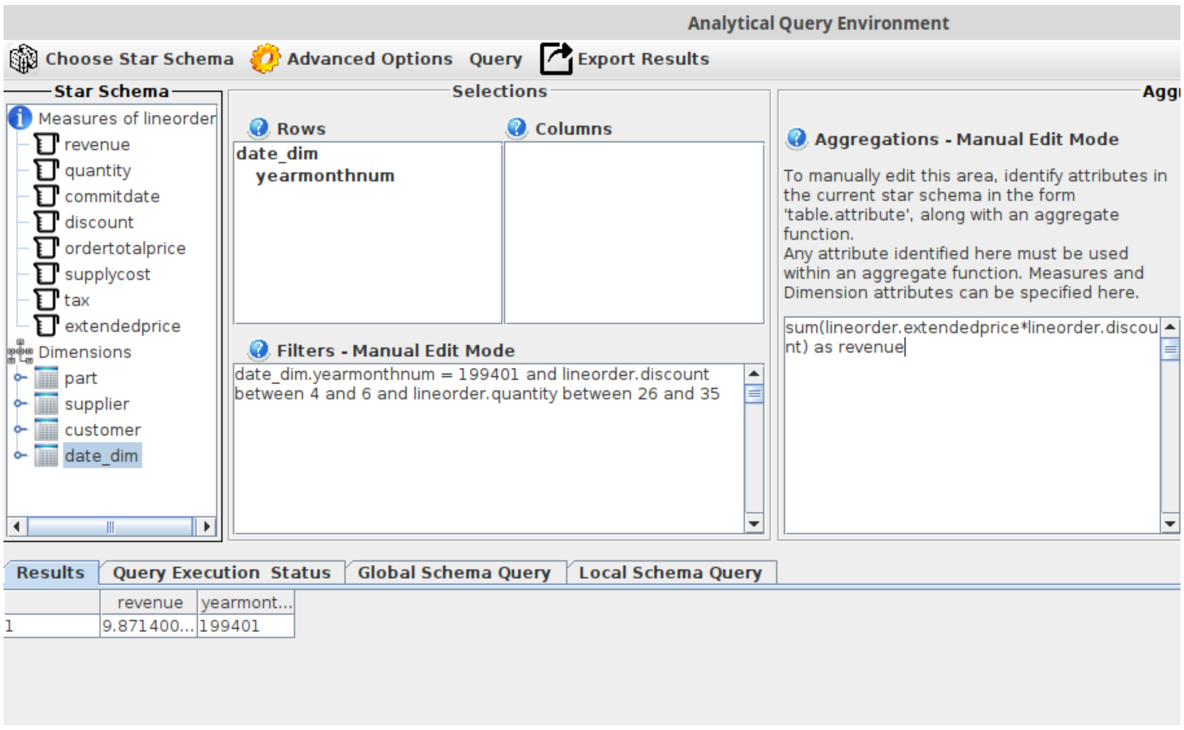
```
SELECT "supplier"."city","part"."brand1","date_dim"."year",
SUM("revenue") AS "SUM␣of␣revenue"
FROM
   (SELECT hive.minhodb."supplier"."city",
    hive.minhodb."supplier"."suppkey",
    hive.minhodb."supplier"."region"
   FROM hive.minhodb."supplier") AS "supplier",
    (SELECT hive.minhodb."part"."brand1",
    hive.minhodb."part"."partkey",hive.minhodb."part"."category"
   FROM hive.minhodb."part") AS "part",
    (SELECT hive.minhodb."date_dim"."year",
    hive.minhodb."date_dim"."datekey"
   FROM hive.minhodb."date_dim") AS "date_dim",
    (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"."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")
LIMIT 50000;
```

Query number 2, shown in Listing 6.3, is composed by an aggregation operation which multiplies two different attributes, a join operation between the facts and dimension tables, and a filter.

When creating query 6.3 on EasyBDI using drag-and-drop, it was found that it was not possible to express the aggregation presented in this query, unlike the previous query. This was a major impediment that initially prevented the creation of queries with similar operations in EasyBDI, because only simple aggregations were possible to be created using drag-and-drop. With the inclusion of manual edition of aggregations and filters, this query could successfully be replicated in EasyBDI. Figure 6.5 shows a print of EasyBDI's interface with a configuration that generates the query seen in Listing 6.4, which is fairly similar to the original query in 6.3, except that inner queries are added and that joins are made using *where* conditions. Also note that the attribute "yearmonthnum" from table "date_dim" is in *select* clause of the query generated by EasyBDI, while the original query does not add this attribute in the *select* clause. The reason for this is that all tables whose attributes are present in the row filters area must have at least one attribute selected on either the rows or aggregation areas, as mentioned in Section 5.5.3. In this case, not adding any other attribute from table "date_dim" in any of these areas, while an attribute of this table is used in the filters, would result in EasyBDI not being able to generate a correct query. The original query also adds an alias to the aggregation operation. Adding aliases to aggregated attributes is possible but only when using manual edition mode, otherwise, as mentioned in Section 4.6.1, aggregated attributes will have an alias assigned to them in the form "<aggregation> of <attribute name>".

Listing 6.3: SSB+ benchmark - Query 2 from the set of batch OLAP queries

```
SELECT sum(lo.extendedprice*lo.discount) as revenue
FROM lineorder lo
JOIN date_dim d ON lo.orderdate = d.datekey
WHERE d.yearmonthnum = 199401 AND lo.discount BETWEEN 4 AND 6
  AND lo.quantity BETWEEN 26 AND 35
```



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

Figure 6.5: Query 2 from the set of batch OLAP queries of SSB+ bechmark created using EasyBDI.

Listing 6.4: EasyBDI query generated: query 2 from the set of batch OLAP queries in SSB+ benchmark

```
SELECT sum(lineorder.extendedprice*lineorder.discount)
AS revenue, "date_dim"."yearmonthnum"
FROM
   (SELECT hive.minhodb."date_dim"."yearmonthnum",
   hive.minhodb."date_dim"."datekey"
  FROM hive.minhodb."date_dim") AS "date_dim",
   (SELECT hive.minhodb."lineorder"."discount",
   hive.minhodb."lineorder"."orderdate",
   hive.minhodb."lineorder"."extendedprice",
   hive.minhodb."lineorder"."quantity"
  FROM hive.minhodb."lineorder") AS "lineorder"
WHERE ("date_dim"."datekey" = "lineorder"."orderdate" )
   AND ( date_dim.yearmonthnum = 199401
   AND lineorder.discount BETWEEN 4 and 6 and lineorder.quantity
   BETWEEN 26 and 35)
GROUP BY ( "date_dim"."yearmonthnum")
LIMIT 50000;
```

Two more examples of queries from the set of queries in the batch OLAP star schema can be seen in Figures A.2 and A.3, although similar to the ones shown here.

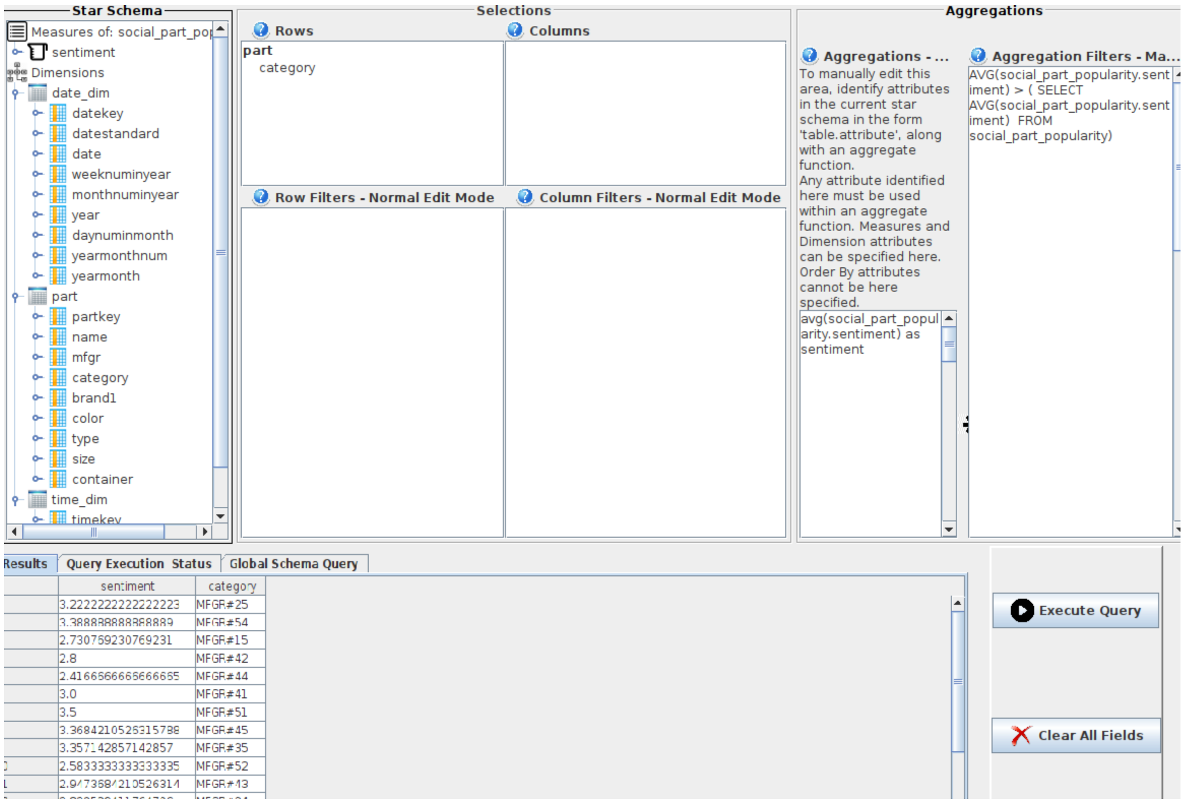
Stream OLAP Queries

SSB+ offers 3 distinct queries for the stream OLAP star schema. The third query, seen in Listing 6.5, includes a nested query in the *having* clause, which cannot be created in EasyBDI. However, the nested query exists only in the *having* clause, meaning that EasyBDI's manual mode can be used in the aggregation filters area to write the inner query directly. Figure 6.6 depicts EasyBDI interface where query 6.5 is recreated by placing the needed attributes in the rows area and by manually writing the average aggregation operation (which could also be created using normal mode in this example) and the *having* clause containing a nested query. With this, EasyBDI generates the query 6.6.

This query was possible to be executed on EasyBDI because the nested query in the *having* clause contains attributes belonging to tables that are also present in the outer query. For example attribute "sentiment" was added on the "aggregations" area, and was therefore added to the generated outer query. If the nested query contained elements not present in any of the selection areas of EasyBDI (rows, columns or aggregations), it would not be possible for EasyBDI to generate a valid query. For example, if by using manual edition mode it was included an attribute referencing a table or column not selected in any of the other areas, than the generated subqueries that are responsible to fetch data from the local schema will not include that table, and the query will fail. Therefore, the use of nested queries in EasyBDI is possible, but limited, and works only if all the attributes present in said nested queries have been included in the selection zones of EasyBDI.

Listing 6.5: SSB+ benchmark - Query 3 from the set of streaming OLAP queries

```
SELECT p.category , gender , AVG(sentiment) sent
FROM social_part_popularity AS spp
JOIN hive.$4.part AS p ON spp.partkey = p.partkey
GROUP BY p.category , gender
HAVING AVG(sentiment) >
 (SELECT AVG(sentiment) FROM social_part_popularity);
```



Rows: part, category

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. Order By attributes cannot be here specified.

avg(social_part_popularity.sentiment) as sentiment

AVG(social_part_popularity.sentiment) > (SELECT AVG(social_part_popularity.sentiment) FROM social_part_popularity)

	sentiment	category
	3.2222222222222223	MFGR#25
	3.3888888888888889	MFGR#54
	2.730769230769231	MFGR#15
	2.8	MFGR#42
	2.4166666666666665	MFGR#44
	3.0	MFGR#41
	3.5	MFGR#51
	3.3684210526315788	MFGR#45
	3.357142857142857	MFGR#35
	2.5833333333333335	MFGR#52
	2.9473684210526314	MFGR#13

Figure 6.6: Query 3 from the set of stream OLAP queries in SSB+ bechmark created using EasyBDI.

Listing 6.6: EasyBDI query generated: query 3 from the set of streaming OLAP in SSB+

```
SELECT avg(social_part_popularity.sentiment) as sentiment,
"part"."category" FROM
   (SELECT hive.minhodb."part"."category",
   hive.minhodb."part"."partkey"
  FROM hive.minhodb."part") AS "part",
   (SELECT cassandra.minho."social_part_popularity"."partkey",
   cassandra.minho."social_part_popularity"."sentiment"
  FROM cassandra.minho."social_part_popularity" )
   AS "social_part_popularity"
WHERE ("part"."partkey" = "social_part_popularity"."partkey")
GROUP BY ("part"."category")
HAVING AVG(social_part_popularity.sentiment) >
 (SELECT AVG(social_part_popularity.sentiment) FROM social_part_popularity)
LIMIT 50000;
```

Query 1, depicted in Listing 6.7, shows a creation of a sub query named "sentByCountry" and then using it in another query, which, as explained, cannot be done in EasyBDI. The same query also contains a *union all* operation, which cannot be added by users in EasyBDI, therefore, this query cannot be fully expressed. However, the contents of the sub query named "sentByCountry" can be created in EasyBDI, but it is not possible to re-use this query in other outer queries.

Note that it would be possible to create a virtual table using the "sentByCountry" query, as explained in Section 5.3.2, and then use said table in the star schema to make queries against this table.

Listing 6.7: SSB+ benchmark - Query 1 from the set of streaming OLAP queries in SSB+

```
WITH sentByCountry AS
     (SELECT country, AVG(sentiment) as sent
    FROM social_part_popularity GROUP BY country)

SELECT * FROM
 (SELECT * FROM sentByCountry ORDER BY sent DESC LIMIT 2)
UNION ALL (SELECT * FROM sentByCountry ORDER BY sent ASC LIMIT 2);
```

Finally, query number 2, seen in Listing 6.8, contains a set of case statements that cannot be manually created using EasyBDI, which makes it impossible to express this query.

Listing 6.8: SSB+ benchmark - Query 2 from the set of streaming OLAP queries in SSB+

```
SELECT
  CASE
    WHEN t.hour >= 0 AND t.hour <= 6 THEN 'Dawn'
    WHEN t.hour > 6 AND t.hour <= 12 THEN 'Morning'
    WHEN t.hour > 12 AND t.hour <= 18 THEN 'Afternoon'
    WHEN t.hour > 18 AND t.hour <= 23 THEN 'Night'
     else NULL END AS dayPeriod,
  p.category,
  COUNT(sentiment) as count
FROM social_part_popularity AS spp
JOIN hive.$4.part AS p ON spp.partkey = p.partkey
JOIN hive.$4.time_dim AS t ON spp.timekey = t.timekey
WHERE (country='Portugal' OR country='Spain') AND gender='Female'
GROUP BY t.hour, p.category;
```

6.2 Case Study 2: Ausgrid Dataset

Ausgrid, an Australian electricity company which owns and operates the distribution grid in Sydney and nearby areas in New South Wales, has made available a dataset[3] containing information of photovoltaic (pv) panel production and consumption of 300 randomly selected customers from 1 July 2010 to 30 June 2013 [51]. The pv panels installed in residences in Sydney have measuring equipment to record consumption and production of energy every half hour for each customer. The types of pv panel measuring data are production (or gross generation, GG), consumption (or general consumption, GC) and controllable load (CL), associated

[3]https://www.ausgrid.com.au/Industry/Our-Research/Data-to-share/Solar-home-electricity-data

with water heating and only available for some customers [51]. Customer information is very limited to protect their anonymity, hence only id and postal code number is known.

The work presented in [51] cleans the Ausgrid dataset by removing data anomalies, such as hour changes, and other irregularities in the data, and reshapes the original dataset structure. This work provides a Github repository[4] where the results are presented, such as Python Notebooks showing the process of dataset cleaning and reshape, as well as examples of analysis over the cleaned dataset.

6.2.1 Dataset Configuration

The cleaned dataset was not used with EasyBDI because it was not made available by the authors, so instead, the original Ausgrid datasets are used and analytical queries will be made over this dataset. However, unlike the datasets in case study 1, a well organized star schema cannot be created with the current file structure. To fix this, the dataset constituted by the 3 files needs to be separated and organized into dimensions and facts table.

The proposed star schema to be created in EasyBDI can be seen in Figure 6.7. This implies the creation of dimensions in databases, while the facts table is constituted by the 3 files from all years. This star schema creates 3 dimensions:

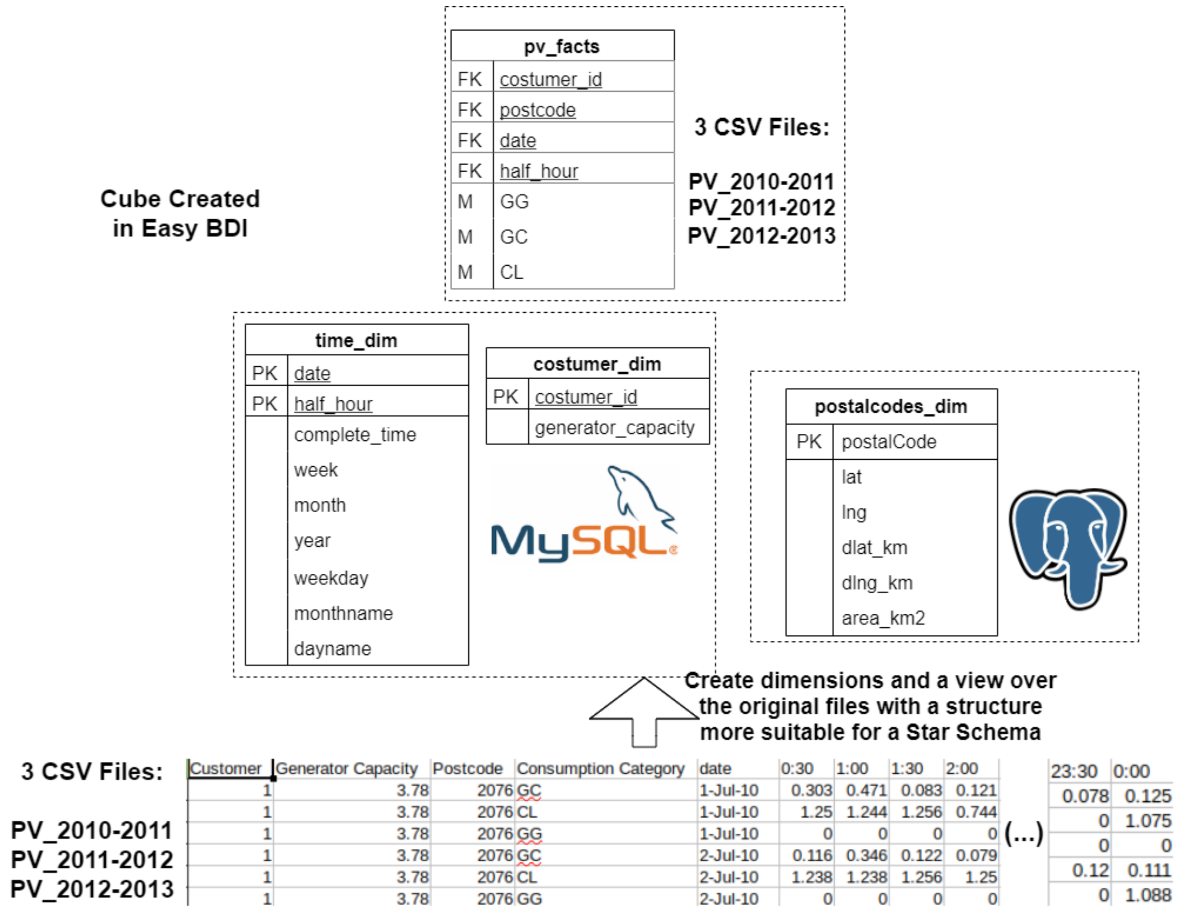
- customers, containing id (which is the only available information of a customer) and the generator capacity associated with their photovoltaic panel, which is a valor associated with each customer, and is included here.
- postalcode, complemented with additional data such as area and coordinates of the postal code area, that can be found on the repository associated to the work seen in [51].
- time, from 01/07/2010 to 31/06/2013, containing several time granularities, from year, month, week, day and hour. A script[5] was used to generate a time dimension table.

Scripts that create each of the dimensions and populate them with data are available in the project's repository, inside the "pv_dataset" directory.

The table that mapped as the facts table was created using data from the original files, however, the current structure of the files, as seen in Figure 6.7, needs to be modified because it is not suitable for a star schema. For example, each pv panel generation data for each half hour is set as several columns, which makes queries more complicated and is not compatible with the created time dimension table, so it was preferable that the hour data was set as rows and the values of electricity generation data to be a separated column. In SQL, this is the equivalent an unpivoting operation for the hour columns. Additionally, the different types of electricity generation data categories (GG, GC and CL) are set as rows, and these fields are the main analytical data of most interest from which aggregations would be performed, so it would make sense for this values to be treated as measures, which means that these 3 categories need to be pivoted. Finally, column "Generator Capacity" is not necessary in the facts table, because it is already associated with a customer id in the customers dimension table.

[4]https://github.com/pierre-haessig/ausgrid-solar-data

[5]Time Dimension Script Adapted from: https://www.joyofdata.de/blog/setting-up-a-time-dimension-table-in-mysql/



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			

Figure 6.7: Case study: pv dataset - Overview of local schema and star schema organization.

The proposed changes to the structure of the original 3 files, seen in the "pv_facts" table of Figure 6.7 are intended to be applied by creating a table view, without altering the original files, using EasyBDI's feature that allows creating virtual tables using the results of SQL queries, as detailed in Section 5.3.2. The query that uses the necessary operations to create an output similar to the table in Figure 6.7 is very extensive because it contains an unpivot operation[6] to place the hours in rows instead of columns by performing a *UNION ALL* operation for each hour. With 48 hours (30 minutes interval), 48 unions are needed. Pivot operations are also present for the 3 categories of electricity generation by using the case statements. The final query can be seen in Listing 6.9, although the 48 are omitted for simplification. This query must be executed for each source file.

Listing 6.9: SQL query to create a table with energy data. Contains a pivot and an unpivot operation.

```
SELECT customer, postcode, date, half_hour,
max(case when "consumption_category" = 'GG' then value end) as GG,
max(case when "consumption_category" = 'GC' then value end) as GC,
max(case when "consumption_category" = 'CL' then value end) as CL
```

[6]SQL Unpivot Code Adapted from https://www.informit.com/articles/article.aspx?p=2755707&seqNum=4

```
FROM (select customer,  Postcode, "Consumption␣Category", "date",
 '0:00' as half_hour, "0:00" AS value from
 flex.csv."file:///path/2012-2013_solar_home_electricity.csv"
UNION ALL
 select customer,  Postcode, "Consumption␣Category", "date",
 '0:30' AS half_hour, "0:30" AS value FROM
 flex.csv."file:///path/2012-2013_solar_home_electricity.csv"
UNION ALL
 (...)
UNION ALL
 select customer,  Postcode, "Consumption␣Category", "date",
 '23:30' as half_hour, "23:30" AS value FROM
 flex.csv."file:///path/2012-2013_solar_home_electricity.csv"
GROUP BY customer,  postcode, date, half_hour;
```

The query in Listing 6.9's output can be used in one of 2 ways:

1. save the output as a new file and connect that file directly to EasyBDI
2. connect the original files to EasyBDI, and execute query in Listing 6.9 each time that access to the original files is needed and use the output of that query instead of the contents of the original files

There are pros and cons between both approaches. In the first one, the query would run only once outside of EasyBDI, using Presto and saving the output as files which would then be used to create the table in EasyBDI. In the second approach, the SQL code needs to be executed each time the virtual file needs to be accessed, which is a similar concept as views, present in most relational databases, but this could increase the overall execution time of the overall analytical query. This also required the implementation of a new feature in EasyBDI that can create a view of objects in the local schema and use that view for the global schema configuration instead of the contents in the data sites.

In the end, it was decided that solution 2 would be a good feature to be included in EasyBDI, as it covers scenarios where users would like to connect data sources, such as files, that contain a structure that is incomplete or incompatible with the overall star schema. And so, this is what inspired the implementation of the "views using SQL code" feature, as described in Section 5.3.2.

To create the virtual facts table, EasyBDI needs to execute a query containing the results of query in Listing 6.9. There are 3 files, so query in Listing 6.9 needs to be executed three times in order to create a virtual table for each file. In this case, the 3 virtual tables will be horizontally partitioned, and each one will map to one global table. When access to the 3 virtual tables is needed, the executed local schema query will need to execute query in Listing 6.9 3 times, one for each file, and combine their outputs with a *union all* operator. However, this operation is so complex and results in various stages being created by Presto, 1 for each *union* and one for each aggregation, resulting in 48 + 3 = 51 stages per file! A single query containing all 3 files will execute this code for each one, creating around 150 stages, and Presto's default stage limit is 100, as previously mentioned in 3.3. This can be changed to other value, such as 200, in one of Presto's configuration files by adding "query.max-stage-count=200". For this reason, the original dataset was splitted, and instead of a file with 300 customers (containing more than 250 000 lines), only 25 customers (about 23 000 lines) are used because the Presto cluster used for this experience constraints the amount of data that can be used in such complex queries in files with such high volume of data.

6.2.2 Project Configuration In EasyBDI

With all data sources created and populated with data, the next step is to connect them to EasyBDI in order to create the global and star schemas (recall an explanation of the full project creation process in Section 5.3). First step is to create a new project by introducing data source connection information, and selecting the relevant tables and schemas of each one. The proposed global schema suggested by EasyBDI for this project (recall data integration techniques used in EasyBDI in Section 4.4.2) created global tables for all dimensions and established a simple mapping between the correspondent local schema tables, however the global table that maps to the 3 data files (and should have a horizontal mapping with the 3 files in the local schema), contains incorrect mappings. In addition, the structure of this table is not the one desired for this star schema and contains the organization of the original files (seen in Figure 6.7), as previously explained. This is where the feature that allows the creation of virtual local tables can be used. To help illustrate the global schema configuration process for this project, Figure 6.8 shows images for each step of the configuration performed in EasyBDI.

The first step is to delete the proposed global table considered as incorrect, then create a view in the local schema using the SQL code in Listing 6.9. To do this, the local table that needs to be altered is selected, right clicked to open a pop up menu, and select the option that allows the insertion of code to create a view. As seen in the second row of images of Figure 6.8, a window will appear, prompting the user to type or paste SQL code, which is then tested, and if no problems occur, the output of the query is used to create a new table in the local schema and it will appear below the "original" table in the JTree. This is repeated 2 more times for the other local tables that will be used to create the facts table (one local table per file). Afterwards, one of the created virtual tables is dragged and dropped to the global schema, and columns from the other two tables are dragged and dropped in order to create valid correspondences. In the end, each column of the global table should have 3 correspondences to the three created virtual tables, creating a horizontal mapping. Data types of each column of the global table are changed because any data originated from files are set to varchar by deafault, so for example, the attribute related to hours is set to "varchar" instead of "time" and needs to be changed. This change to the datatypes could also be performed on the SQL query that was used to create the local tables, by casting each column to the correct data type, however for demonstrations purposes it was chosen no to do so. The final step is to validate all primary keys and set foreign keys in the facts table.

6.2.3 Queries Over The Created Star Schema

Queries are now executed using the previously created star schema. The objective is to create queries that will reveal useful information that can lead to important conclusions on the overall dataset. The Github repository in [51] has notebooks that evaluate and extract information from the cleaned dataset. Some of those queries will be translated to EasyBDI in order to create a query that provides similar results, whilst also evaluating the system's capability and possible limitations when creating queries.

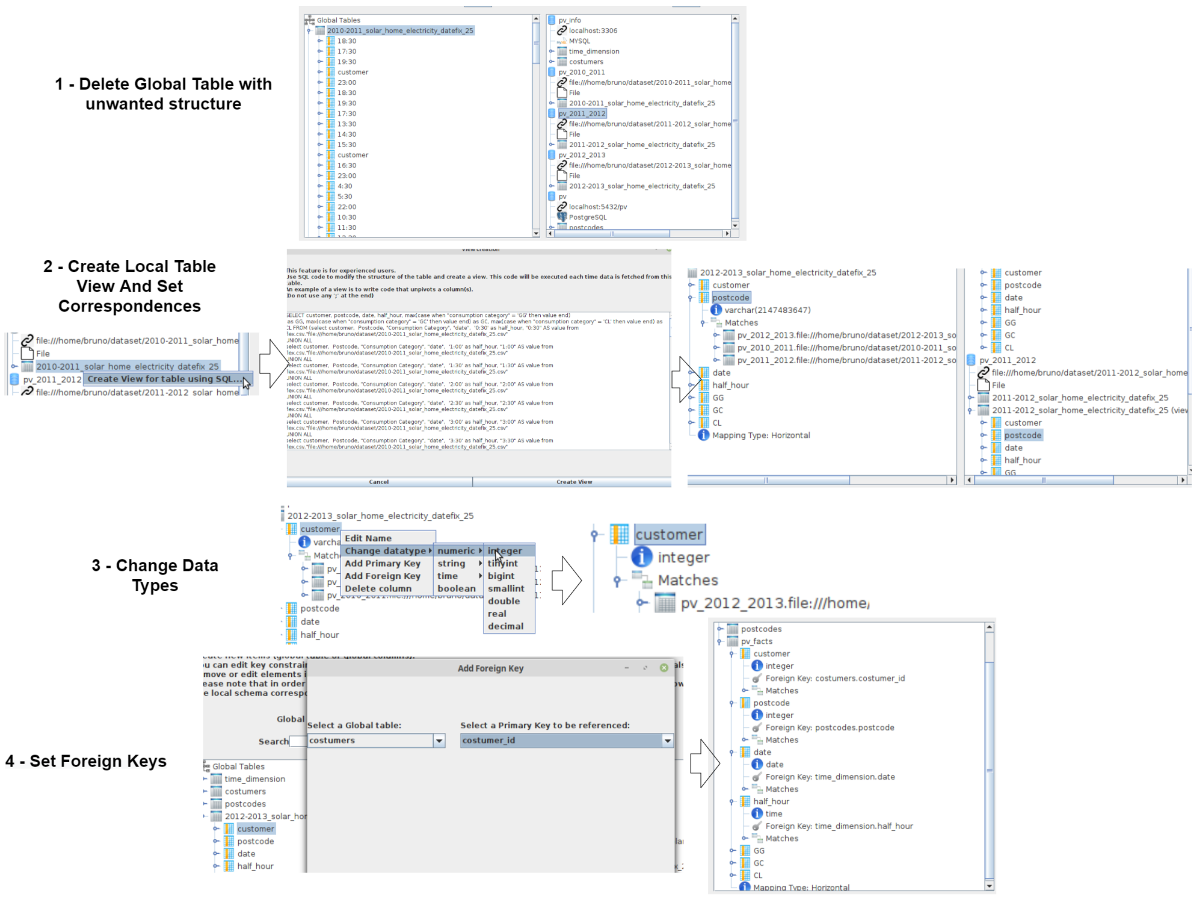
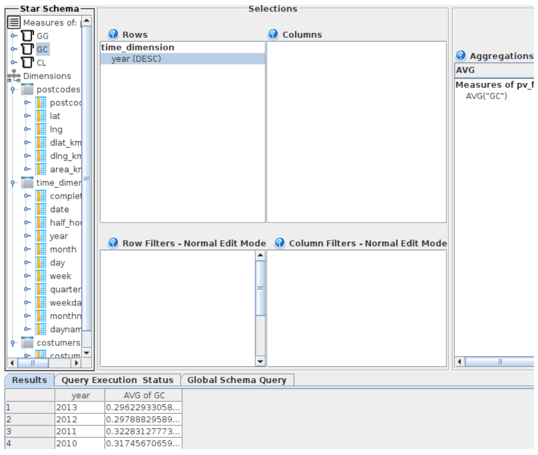

1 - Delete Global Table with unwanted structure
2 - Create Local Table View And Set Correspondences
3 - Change Data Types
4 - Set Foreign Keys
Create View for table using SQL...
Cancel
Create View
Edit Name
Change datatype
Add Primary Key
Add Foreign Key
Delete column
numeric
string
time
boolean
integer
tinyint
bigint
smallint
double
real
decimal
customer
integer
Matches
pv_2012_2013.file:///home/
Add Foreign Key
Select a Global table:
costumers
Select a Primary Key to be referenced:
costumer_id
Foreign Key: costumers.costumer_id
Foreign Key: postcodes.postcode
Foreign Key: time_dimension.date
Foreign Key: time_dimension.half_hour
Mapping Type: Horizontal

Figure 6.8: Steps needed to create the Global Schema in EasyBDI using the PV dataset.

For example, one of the most important information that can be extracted from this dataset is the evolution in production and consumption of energy over the years. Figure 6.9 depicts the creation of 2 queries in EasyBDI: the query in Figure 6.9a shows the average consumption of energy for all years, and query in Figure 6.9b shows the production of energy by client, while also showing only two years.

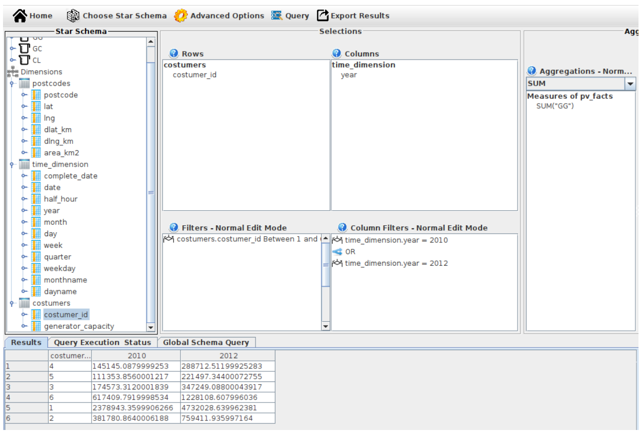

Rows: time_dimension, year (DESC)

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) Consumption average of all years.


Rows: costumers, costumer_id

Columns: time_dimension, year

Aggregations - Norm...: SUM

Measures of pv_facts: SUM("GG")

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

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

	costumer...	2010	2012
1	4	145145.0879999253	288712.51199925283
2	5	111353.8560001217	221497.3440072755
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) Total Production of energy by clients in years 2010 and 2012.

Figure 6.9: Evaluation of energy consumption and production over some or all years.

Listing 6.10: Easy BDI Generated Query from the configuration seen in figure 6.9a.

```
SELECT "time_dimension"."year",
AVG(CAST("GG" AS double)) AS "AVG_of_GG" FROM
   (SELECT mysql.pv_schema."time_dimension"."half_hour",
   mysql.pv_schema."time_dimension"."year",
   mysql.pv_schema."time_dimension"."date"
  FROM mysql.pv_schema."time_dimension" ) AS "time_dimension",
SELECT "hour", "half_hour", "GG" FROM
   (<user code for file1> UNION <user code for file2>
   UNION <user code for file3>) AS "pv_facts"
WHERE ("time_dimension"."half_hour" =
CAST("pv_facts"."half_hour" AS time) AND
"time_dimension"."date" = CAST("pv_facts"."date" AS date))
GROUP BY ("time_dimension"."year")
ORDER BY "time_dimension"."year" DESC
LIMIT 50000;
```

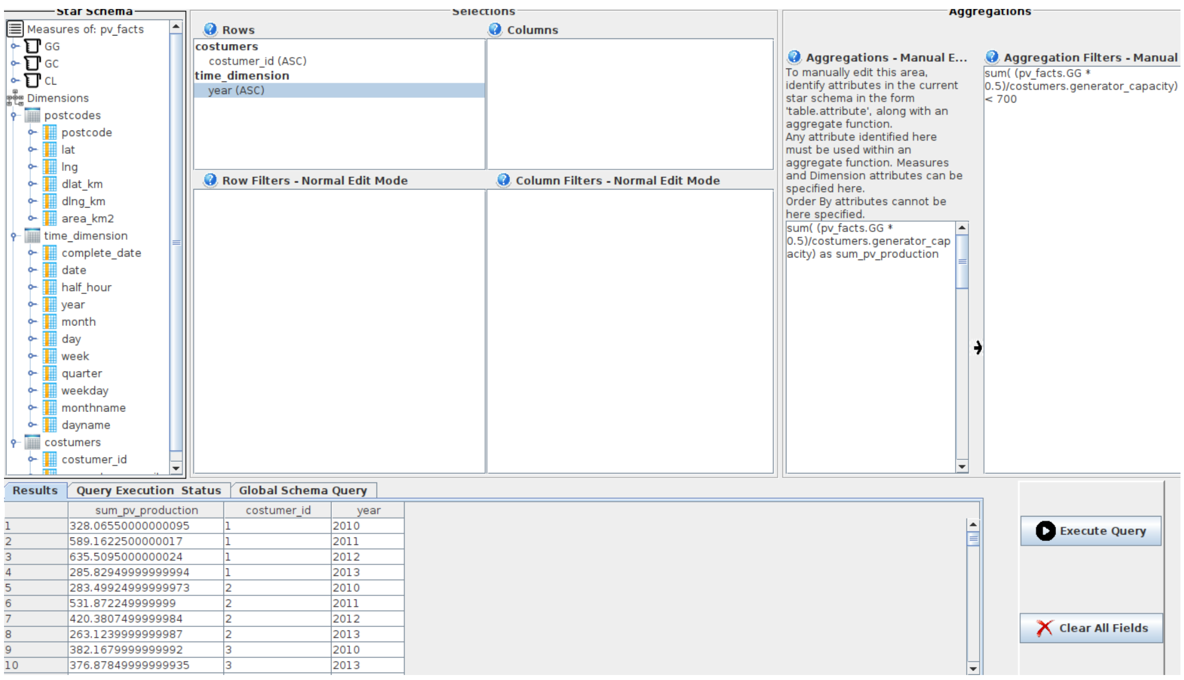
Listing 6.11: Easy BDI Generated Query from the configuration seen in figure 6.9b.

```
SELECT "costumers"."costumer_id",
SUM(CASE WHEN time_dimension.year = 2010 THEN CAST ("GG" AS double)
ELSE 0 END) AS "2010", SUM(CASE WHEN time_dimension.year = 2012
THEN CAST ("GG" AS double) ELSE 0 END) AS "2012" FROM
   (SELECT mysql.pv_schema."costumers"."costumer_id"
   FROM mysql.pv_schema."costumers") AS "costumers",
   (SELECT mysql.pv_schema."time_dimension"."half_hour",
   mysql.pv_schema."time_dimension"."year",
   mysql.pv_schema."time_dimension"."date"
   FROM mysql.pv_schema."time_dimension" ) AS "time_dimension",
SELECT "hour", "half_hour", "customer", "GG" FROM
   (<user code for file1> UNION <user code for file2>
   UNION <user code for file3>) AS pv_facts
  WHERE ("costumers"."costumer_id"=CAST("pv"."customer" AS integer))
  AND ("costumers"."costumer_id" Between 1 and 6 )
  GROUP BY ("costumers"."costumer_id")
  LIMIT 50000;
```

More specific information can be presented by changing the time being displayed from year to a specific date, while also presenting the values of production or consumption of energy every half hour. Figure A.4 depicts a query in EasyBDI that shows information regarding energy production, but now showing more detailed information by date, which is done by placing the date attribute in the rows area and again in the filters area to select one specific date (if needed). Production information is also showed by hours in the created query, but EasyBDI offers more than one option to display information, and so the hours can be displayed in 2 different ways: Figure A.4a shows the hours attribute in columns, while Figure A.4b presents the hours as rows.

More complex queries can be created as demonstrated previously. One of the notebooks found in the repository of the work in [51] presents an analysis to determine if a panel is failing or having bad or blank production (related to maintenance or record issues) by analyzing its production over a certain period of time. According to the notebook, this can be done through the sum of half the production value (because it is in kWh, and each entry is only half an hour) divided by the generator capacity (in kWp) of the panels the customer has because some panels can produce more energy than others.

This query was translated and executed in EasyBDI, however the notebook presents conclusions only for the period between July 2011 and June 2012, because only one file was used, whereas EasyBDI can query a larger period (between July 2010 to June 2013) as all 3 files are being analyzed simultaneously. Figure 6.10 shows the query and its results in EasyBDI, by performing a sum aggregation and applying the previously mentioned formula (which required the user of manual mode) and grouping these results by customer and year. However, the notebook concludes that 4 customers had pvs with issues, but those customers are not present in the data files used because the data in each file was reduced. The resulting query from Figure 6.10 can be seen in Listing 6.12.



Rows:

costumers
  costumer_id (ASC)
time_dimension
  year (ASC)

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.
Order By attributes cannot be here specified.

```
sum( (pv_facts.GG *
0.5)/costumers.generator_cap
acity) as sum_pv_production
```

```
sum( (pv_facts.GG *
0.5)/costumers.generator_capacity)
< 700
```

	sum_pv_production	costumer_id	year
1	328.06550000000095	1	2010
2	589.1622500000017	1	2011
3	635.5095000000024	1	2012
4	285.82949999999994	1	2013
5	283.49924999999973	2	2010
6	531.872249999999	2	2011
7	420.3807499999984	2	2012
8	263.1239999999987	2	2013
9	382.1679999999992	3	2010
10	376.87849999999935	3	2013

Figure 6.10: Query to analyze the performance of pv panels of each customer by year.

Listing 6.12: Easy BDI Generated Query from the configuration seen in figure 6.10. This query contains attributes whose original data types were changed, and manual aggregations and filters. Facts table "pv_facts" is also composed by 3 horizontal partitioned virtual tables, each one built using a user specified query.

```
SELECT
sum((CAST("pv_facts"."GG"AS double)*0.5)
/costumers.generator_capacity)
as sum_pv_production,"costumers"."costumer_id",
"time_dimension"."year"
FROM
    (SELECT mysql.pv_schema."costumers"."costumer_id",
    mysql.pv_schema."costumers"."generator_capacity"
   FROM mysql.pv_schema."costumers" ) AS "costumers",
    (SELECT mysql.pv_schema."time_dimension"."year",
    mysql.pv_schema."time_dimension"."date",
    mysql.pv_schema."time_dimension"."half_hour"
   FROM mysql.pv_schema."time_dimension" ) AS "time_dimension",
SELECT "hour", "half_hour", "customer", "GG" FROM
    (<user code for file1> UNION <user code for file2>
   UNION <user code for file3>) AS "pv_facts"
WHERE("time_dimension"."half_hour" =
CAST("pv_facts"."half_hour" AS time) AND
"costumers"."costumer_id" = CAST("pv_facts"."customer" AS integer)
AND "time_dimension"."date" = CAST("pv_facts"."date" AS date))
GROUP BY ("time_dimension"."year","costumers"."costumer_id")
HAVING sum((CAST("pv_facts"."GG"AS double)*0.5)
/costumers.generator_capacity)<700
ORDER BY "costumers"."costumer_id" ASC,
"time_dimension"."year" ASC
LIMIT 50000;
```

The analyzed time period can easily be changed by using the time dimension and selecting a smaller time period for more specific analyzes, such as month instead of years.

This experiment helped to improve the system by discovering and fixing some problems related to manual mode, namely:

- Originally, EasyBDI didn't validate attributes' data types in the code inserted by users when in manual edition. Hence, query execution would probably fail if the code has references to an attribute whose data type changed and data type casting operations are not used. To prevent such type of errors, the system could ask users to manually add the cast operation themselves in these cases, but this would not be ideal. The solution implemented is to add to the user written code any necessary cast operators whenever an attribute whose data type was changed is referenced.

- If decimal numbers are present when parsing SQL code in manual mode to find columns and tables (as explained in Section 5.5.4), the decimal point is confused with the qualified name of a column, which resulted in the system assuming that "0.5" was a table named 0 and a column named 5. This was later fixed and it is ensured that decimal points surrounded by digits only are ignored.

6.3 Performance And Volume of Queried Data

The focus of this work is mainly on developing a solution in order to satisfy the objectives and requirements in Section 1.3. Performance never was a topic of much focus in the development of this work. However, this section will present a simple overview of EasyBDI's performance.

Performance of query execution time from the moment the user submits the queries from the system until results are displayed on the screen is measured by the time it takes EasyBDI to generate the SQL query, send it to Presto, the time it takes for Presto to execute the query, and finally the time it takes the system to process the query results and display them on the table in the UI. Table 6.1 demonstrates examples of the performance of EasyBDI in some of the queries shown in this chapter. As expected, the query generation process made by EasyBDI generates practically no overhead to the overall performance of the system. Query in Listing 6.11 takes more time than the remaining queries because it needs to first execute a first query to retrieve the values to be pivoted. The remaining time is mainly spent by the query execution time handled by Presto. The first two queries take less time because they retrieve data from Hive, which should contain mechanisms to efficiently retrieve data, as opposed with the remaining queries that retrieve data from raw files which can be slower. Additionally, the last 3 queries also include complex operations to reshape the data's original structure. The time it takes for EasyBDI to process the results and display them in the UI is not considered as it is very short and generates no overhead.

Query	Data Sources	Query Generation Time (ms)	Query Execution Time (s)
Listing 6.2	Hive	4	5.967
Listing 6.4	Hive	3	6.896
Listing 6.10	raw files, PostgreSQL, MySQL	3	67.872
Listing 6.11	raw files, PostgreSQL, MySQL	1496	31.453
Listing 6.12	raw files, PostgreSQL, MySQL	11	53.512

Table 6.1: Query Generation Times And Query Execution Times By Query And Their Complexity.

The main reasons that Presto may take longer to execute queries, are the cluster environment. Recall that all queries here were executed on a single machine. More nodes could improve the performance of the query execution time. Query execution time may very. For example, query in Listing 6.12 took 87.56 seconds on another execution. Additionally, the system only displays the data to the user after every result is received from Presto, which is not an ideal solution. The JDBC API delivers the results to EasyBDI periodically as they are ready if the volume of data is big. EasyBDI should display the results to the user as they are ready, and not once it receives all results, which may lead to longer times before the user can see any results.

6.4 Considerations

This section serves 3 purposes: to exemplify how to use EasyBDI, to demonstrate what the system can do, and what are its limitations when creating queries. This was quite useful throughout the development, as it allowed not only to fix errors, but also to improve and add features in order to reduce some of EasyBDI's limitations.

The first case study analyzed provides a dataset and a set of model queries, mostly used for performance benchmarking. Each of these queries were created in EasyBDI in order to determine which ones were possible to execute, and which were not. Initial tests revealed that some queries could not be totally specified based on drag-and-drop operations. To mitigate this, manual edition mode was created (explained in Section 5.5.4) in order to provide more flexibility to the user when writing aggregations and filters, increasing the number of complex operations that can be specified on queries created on EasyBDI, thus making it possible to recreate a greater number of model queries from the ones available in this case study's dataset. However, this is not a generic solution and does not allow the same expressiveness as an SQL query. Other limitations include other SQL operations, such as *case* statements and *union all*, as well as the creation of nested queries which can only be done in the filters using manual mode. These limitations do not allows some of the model queries to be recreated.

The second case study presents a more realistic use case for EasyBDI, related to photovoltaic energy production and consumption data. Useful information can be extracted from this dataset through analytical queries, such as total production of energy per client. This show cases the ability of EasyBDI to incorporate data whose original organization may not be compatible or unwanted for a star schema. In this case, the original organization of the data from this dataset was not compatible with the creation of a star schema, but EasyBDI can solve this by creating a view via a SQL query, and then using this view to create the star schema and perform queries.

In conclusion, both case studies show that EasyBDI can integrate data and suggest a global schema, regardless of the data sources used. This global schema may not always be correct, but it can be corrected using an intuitive user interface, and star schemas can be created from the global schema. A reorganization of a table's structure can be done if necessary and then used as a part of these schemas. Performing queries using an user interface can also allow for less experienced users to perform analytical queries, as the system automatically generates a valid SQL query, with joins whenever necessary and other requested user operations, thus abstracting from specific SQL language details. These tests also allowed an experimentation with a larger diversity of queries, allowing to detect and correct some of EasyBDI's limitations and fix errors that appeared only in specific scenarios.

EasyBDI is a tool that can be used for analytical scenarios, and provides many features that increase its versatility.

Chapter 7

Conclusions and Future Work

In modern information systems, data is scattered in several heterogenous data sources, which creates difficulties when trying to query and integrate them. Some techniques and architectures exist to overcome this challenge, and several systems have been proposed with different strategies, which bring advantages and disadvantages.

Another issue that rises with the appearance of new topics, such as IoT, data streams, and e-Commerce, is that traditional data warehouses do not contemplate recent data into analytical queries, which may by critical for these types of decision making environments. To solve this, data warehouse architectures that enable near real-time querying have been suggested. Some of the solutions include a separated cache with a view of the recent data. These DWs also materialize most or all of the data, which is a complex and long process, and if the the structure of the star schema needs to be changed, then the long process of data extracting, integrating and loading must be repeated all over again.

The previous issues presented can be associated as being a part of a larger common issue: support for analytical queries over heterogenous data sources, while retaining the most recent data. To the the best of our knowledge, no tool capable of performing all of these tasks simultaneously has been suggested, and in fact, table 2.1 evidences this conclusion.

A system named EasyBDI was implemented, capable of solving all of these issues. It can perform OLAP queries while also interactively querying data stored in heterogeneous and possibly distributed data sources, regardless of their model. It uses Presto as the distributed query engine that supports big data and heterogeneous data sources. The fact that Presto interactively queries the data sources, means that the most recent data residing in the data sources are used, which offers an improvement over existing near real-time DW architectures, that despite offering access to recent data, do not provide access to the current data. Furthermore, no data is materialized, which means that the process of extracting and loading data does not take place, enabling for a much faster change if the star schema if needed.

EasyBDI is mainly constituted by modules that use Presto. These modules were added to fulfill the requirements that Presto alone could not, such as the creation of a unified (logical) schema providing an overall view of the data residing in data sources. Data Integration techniques were implemented to automatically create this schema. Other implemented features in EasyBDI include the creation of star schemas, and interactive user interfaces in order to guide users in the configuration process. This configuration consists in connecting data sources, correcting the automatic global schema proposed, and mapping entities from this schema to a star schema, thus creating dimensions and facts tables. However, as with traditional DWs, if the schema on any of the data sources is modified, the overall star schema needs to be reconfigured, meaning that EasyBDI does not provide any mechanisms to safeguard against schema evolution. Even so, EasyBDI allows for an easier way to edit star schemas when compared with traditional DW by using the wizard-like user interface, thus offering a solution to recover from schema evolution. Performing intensive operations directly on the data sources can also lead to performance degradation.

On EasyBDI, queries can be created by dragging elements from the star schema to certain areas, each one representing a different output in the final query, such as rows, columns and filters. This allows for an easier validation of the user actions in order to ensure that the user only generates valid SQL queries from the drag-and-drop elements. If the user were to write SQL code, validation would be harder, and this would require more work from the user, as most SQL OLAP queries require multiple joins, which are automatically generated with the implemented approach. This creates some limitations on the queries that can be created. For example, the use of nested queries is very limited, and there is a wide variety of SQL commands, making it very hard to implement all of these commands in a first implementation of EasyBDI.

In conclusion, this work fulfills the proposed objectives defined in Section 1.3, and can also fulfill every single requirement presented in Table 2.1, which no other of the reviewed systems could. This work showcases a system that solves modern problems using well known techniques from the literature, with a focus on real-time analytical querying that non expert users can perform.

7.1 Contributions

The main contributions of this work are:

- A system, named EasyBDI, capable of executing real-time analytical queries over distributed and heterogenous data sources, using logical data integration to create an abstraction of the complexity of dealing with multiple schemas.

- EasyBDI can also be classified as a user-friendly system, as both experts and non-experts can perform analytical queries through a user interface based on drag-and-drop. Users can configure the data sources to be used, adjust the proposed global schema, and create a star schema, all through a wizard-like user interface, thus promoting an easier and more interactive configuration process.

- EasyBDI generates SQL queries from user input, which are sent to Presto in order to query the local schema. Users use high level abstractions (star schemas over global abstract entities), but the generated SQL queries include references to the physical structures and data sources where data resides. These queries receive the results from the local schema and present them accordingly with the user perceived star schema. Data partitioning is also transparent to the end user, as EasyBDI generates queries that fetch partitioned data across multiple data sources.

- EasyBDI offers several helpful features. It enables querying on raw files, such as csv, txt and others (a characteristic not seen in many systems), which can be useful for querying data without loading it into a database, for example. It also enables the creation and management of different projects, which allows users to better organize and separate data sources relating to different contexts. Other utilities are also present, such as exporting results and saving queries (very useful if certain queries need to be executed periodically).

- For the development of EasyBDI, an algorithm that applies data integration techniques to automatically generate a global schema was developed.
- This dissertation also presents an overview of multiple solutions that have been suggested to solve both problems presented (near real-time analytical querying and distributed and heterogenous querying) while also classifying and comparing them using the requirements specified in Section 1.3.

7.2 Future Work

Many features of EasyBDI can be improved, and many other useful features could be added. The future work can be split in 3 categories: EasyBDI features and user experience, data integration techniques and Presto integration.

EasyBDI Features and User Experience: - In this work, many useful features were implemented and several validations were also implemented to make the system robust, useful and usable (despite the version of the system here presented is a proof of concept), but it is not bulletproof. Many other features could be added in order to better assist users while creating queries or editing the global schema.

The following improvements or changes are proposed:

- The current user interface of EasyBDI could be submitted into a usability test in order to determine how intuitive and how user friendly it is, and from that feedback enhance the overall User Experience (UX) of EasyBDI. An intuitive interface is very important for the creation of global schemas and to execute queries, but even with informational tool tips, the UX could be improved in order to better assist the user.
- Add more support for datatype changes and better handling of unsupported data types in the global schema edition. For example, a query could be run on Presto to test if the cast is possible, such as string to numeric castings, as some strings cannot be converted to numeric. Currently it is assumed that the user defined data type change is possible and will not generate errors.
- Redesign the way how matches between global to local schemas are presented.
- Further work could be done to provide more OLAP functionalities, namely, the organization of data in hierarchies and the use of OLAP operators, such as slice, dice and drill down.
- Work could be done to improve the performance of the system, namely on query execution time. This could potentially change Presto's implementation, or generate queries in a more efficient way.

Data Integration Improvements: - The database integration techniques could be improved by employing more robust algorithms and techniques. Their performance could be evaluated in order to generate more accurate global schemas.

A solution could also be developed to deal with schema evolution, meaning that, changes on the underlying local schema would not generate any errors on the global schema created (such as missing tables or modified columns). In these events, the system could alert the user that the local schema has changed or automatically adjust the global schema.

Presto Integration Improvements: - As of now, the management of each Presto node cannot be controlled within EasyBDI, as users simply specify the location of Presto Coordinator. Catalogues are generated by the system using user configuration, but in order for the catalogues to be used, the system uses a Process Builder to restart Presto which is not a robust solution. Presto Admin [1] is a command line tool that provides commands to install or uninstall Presto nodes and manage their status, as well as to add or remove catalogues. This tool could be further explored, namely to test how easy it is to perform these tasks, its limitations and the possibility of integrating this tool into EasyBDI. If possible, the system could take user input and pass it to Presto admin to better control the cluster and the catalogues.

[1]https://prestosql.github.io/presto-admin/docs/current/index.html

Bibliography

[1] (2019) An Introduction to Database Sharding. [Online]. Available: https://www.techsupportpk.com/2019/02/what-is-database-sharding.html

[2] E. Rahm and P. Bernstein, “A survey of approaches to automatic schema matching.” *VLDB J.*, vol. 10, pp. 334–350, 12 2001, doi: 10.1007/s007780100057.

[3] R. Tan, R. Chirkova, V. Gadepally, and T. G. Mattson, “Enabling query processing across heterogeneous data models: A survey,” *Proceedings - 2017 IEEE International Conference on Big Data, Big Data 2017*, vol. 2018-Janua, pp. 3211–3220, 2017, doi: 10.1109/BigData.2017.8258302.

[4] (2019) SQL Order of Operations. [Online]. Available: https://learnsql.com/blog/sql-order-of-operations/

[5] P. P. Khine and Z. Wang, “A review of polyglot persistence in the big data world,” *Information (Switzerland)*, vol. 10, no. 4, p. 141, 2019, doi: 10.3390/info10040141.

[6] M. Y. Santos, C. Andrade, C. Costa, B. A. Martinho, E. Costa, J. Galvão, and F. V. Lima, “Evaluating SQL-on-Hadoop for big data warehousing on not-so-good hardware,” in *ACM International Conference Proceeding Series*, vol. Part F1294, 2017, pp. 242–252. [Online]. Available: http://dx.doi.org/10.1145/3105831.3105842

[7] J. Duggan, A. J. Elmore, M. Stonebraker, M. Balazinska, B. Howe, J. Kepner, S. Madden, D. Maier, T. Mattson, and S. Zdonik, “The bigdawg polystore system,” *SIGMOD Rec.*, vol. 44, no. 2, p. 11–16, Aug. 2015, doi: 10.1145/2814710.2814713. [Online]. Available: https://doi.org/10.1145/2814710.2814713

[8] H. Ramadhan, F. I. Indikawati, J. Kwon, and B. Koo, “MusQ: A Multi-Store Query System for IoT Data Using a Datalog-Like Language,” *IEEE Access*, vol. 8, pp. 58 032–58 056, 2020, doi: 10.1109/ACCESS.2020.2982472.

[9] B. Garg and K. Kaur, “Integration of heterogeneous databases,” in *Conference Proceeding - 2015 International Conference on Advances in Computer Engineering and Applications, ICACEA 2015.* Institute of Electrical and Electronics Engineers Inc., jul 2015, pp. 1033–1038, doi: 10.1109/ICACEA.2015.7164859.

[10] M. T. Özsu and P. Valduriez, *Principles of Distributed Database Systems*, 4th ed. Springer International Publishing, 2020, doi: 10.1007/978-3-030-26253-2.

[11] B. Kolev, P. Valduriez, C. Bondiombouy, R. Jiménez-Peris, R. Pau, and J. Pereira, “CloudMdsQL: querying heterogeneous cloud data stores with a common language,” *Distributed and Parallel Databases*, vol. 34, no. 4, pp. 463–503, dec 2016, doi: 10.1007/s10619-015-7185-y.

[12] 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,” *Proceedings - International Conference on Data Engineering*, vol. 2019-April, pp. 1802–1813, 2019, doi: 10.1109/ICDE.2019.00196.

[13] Apache Drill - Architecture. [Online]. Available: https://drill.apache.org/architecture/

[14] Apache Drill - Architecture Introduction. [Online]. Available: https://drill.apache.org/docs/architecture-introduction/

[15] R. J. Santos and J. Bernardino, “Real-time data warehouse loading methodology,” in *ACM International Conference Proceeding Series*, vol. 299, 2008, pp. 49–58, doi: 10.1145/1451940.1451949.

[16] J. Zuters, “Near Real-time Data Warehousing with Multi-stage Trickle & Flip,” in *Perspectives in Business Informatics Research*, J. Grabis, , and M. Kirikova, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 73—-82, doi: 10.1007/978-3-642-24511-4_6.

[17] A. Cuzzocrea, N. Ferreira, and P. Furtado, “A rewrite/merge approach for supporting real-time data warehousing via lightweight data integration,” *Journal of Supercomputing*, vol. 76, no. 5, pp. 3898–3922, may 2020, doi: 10.1007/s11227-018-2707-9.

[18] B. Silva, J. Moreira, and R. Costa, “Easy BDI: Near Real-Time Data Analytics over Heterogeneous Data Sources - To Appear,” in *International Conference on Extending Database Technology*, 2021.

[19] R. Wrembel and C. Koncilia, *Data Warehouses And Olap: Concepts, Architectures And Solutions.* Hershey, PA, USA: IRM Press, 2006, doi: 10.4018/978-1-59904-364-7.

[20] M. T. Özsu and P. Valduriez, *Principles of distributed database systems, third edition*, 3rd ed. Pearson Education, Inc, 2011, doi: 10.1007/978-1-4419-8834-8.

[21] M. Chen and R. Hofestädt, *Approaches in Integrative Bioinformatics: Towards the Virtual Cell*, 2014, doi: 10.1007/978-3-642-41281-3.

[22] C. Y. Ko, “Three Approaches to a Multidatabase System,” *Proceedings of Philippine Computing Scirnce Congress (PCSC) 2000*, pp. 116–121, 2000.

[23] A. P. Sheth and J. A. Larson, “Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases,” *ACM Computing Surveys (CSUR)*, vol. 22, no. 3, pp. 183–236, jan 1990, doi: 10.1145/96602.96604.

[24] A. Adamou and M. d’Aquin, “Relaxing global-as-view in mediated data integration from linked data,” in *Proceedings of The International Workshop on Semantic Big Data*, ser. SBD ’20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3391274.3393635

[25] M. Ptiček and B. Vrdoljak, “Big data and new data warehousing approaches,” in *Proceedings of the 2017 International Conference on Cloud and Big Data Computing*, ser. ICCBDC 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 6–10. [Online]. Available: https://doi.org/10.1145/3141128.3141139

[26] (2017) Data partitioning: Vertical partitioning, horizontal partitioning, and hybrid partitioning. [Online]. Available: http://cloudgirl.tech/data-partitioning-vertical-horizontal-hybrid-partitioning/

[27] C. Batini, M. Lenzerini, and S. B. Navathe, “A Comparative Analysis of Methodologies for Database Schema Integration,” *ACM Computing Surveys (CSUR)*, vol. 18, no. 4, pp. 323–364, 1986, doi: 10.1145/27633.27634.

[28] A. A., A. Nordin, M. Alzeber, and A. Zaid, “A Survey of Schema Matching Research using Database Schemas and Instances,” *International Journal of Advanced Computer Science and Applications*, vol. 8, no. 10, 2017, doi: 10.14569/ijacsa.2017.081014.

[29] Z. Jin, C. Baik, M. Cafarella, and 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: Association for Computing Machinery, Inc, jun 2018, pp. 1–4. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3209900.3209902

[30] R. J. Miller, L. M. Haas, and M. A. Hernández, "Schema mapping as query discovery," in *Proceedings of the 26th International Conference on Very Large Data Bases, VLDB'00*, 2000, pp. 77–88.

[31] Y. Katsis and Y. Papakonstantinou, *View-Based Data Integration.* New York, NY: Springer New York, 2018, pp. 4452–4461. [Online]. Available: https://doi.org/10.1007/978-1-4614-8265-9_1072

[32] A. Bonifati, G. Mecca, P. Papotti, and Y. Velegrakis, "Discovery and Correctness of Schema Mapping Transformations," in *Schema Matching and Mapping.* Springer Berlin Heidelberg, 2011, pp. 111–147, doi: 10.1007/978-3-642-16518-4_5.

[33] B. ten Cate, P. G. Kolaitis, and W.-C. Tan, "Schema Mappings and Data Examples," in *Proceedings of the 16th International Conference on Extending Database Technology*, ser. EDBT '13. New York, NY, USA: Association for Computing Machinery, 2013, pp. 777–780. [Online]. Available: https://doi.org/10.1145/2452376.2452479

[34] Federated Database system. [Online]. Available: https://en.wikipedia.org/wiki/Federated_database_system

[35] Spark SQL, DataFrames and Datasets Guide. [Online]. Available: https://spark.apache.org/docs/latest/sql-programming-guide.html

[36] 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.* Institute of Electrical and Electronics Engineers Inc., aug 2015, pp. 345–349, doi: 10.1109/ISITIA.2015.7220004.

[37] M. Rodrigues, M. Y. Santos, and J. Bernardino, "Big data processing tools: An experimental performance evaluation," vol. 9, no. 2, mar 2018, doi: 10.1002/widm.1297.

[38] A. Kolychev and K. Zaytsev, "Research of the effectiveness of SQL engines working in HDFS," *Journal of Theoretical and Applied Information Technology*, vol. 95, no. 20, pp. 5360–5368, 2017. [Online]. Available: www.jatit.org

[39] About Presto Connectors. [Online]. Available: https://www.starburstdata.com/learn-presto/connectors/

[40] Presto Concepts. [Online]. Available: https://prestosql.io/docs/current/overview/concepts.html#server-types

[41] Deploying Presto. [Online]. Available: https://prestosql.io/docs/current/installation/deployment.html

[42] Pushdown. [Online]. Available: https://prestosql.io/docs/current/optimizer/pushdown.html

[43] EasyBDI Github Repository. [Online]. Available: https://github.com/bsilva3/EasyBDI

[44] R. Fagin, L. M. Haas, M. Hernández, R. J. Miller, L. Popa, and 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, doi: 10.1007/978-3-642-02463-4_12.

[45] Difference Between SQL and MDX. [Online]. Available: https://helicaltech.com/sql-and-mdx-queries/

[46] (2018) SQL Order of Operations – In Which Order MySQL Executes Queries? [Online]. Available: https://www.eversql.com/sql-order-of-operations-sql-query-order-of-execution/

[47] About Presto Connectors. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html

[48] C. Costa and 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. Springer Verlag, 2018, pp. 459–473, doi: 10.1007/978-3-319-91563-0_28.

[49] TPC-H. [Online]. Available: http://www.tpc.org/tpch/

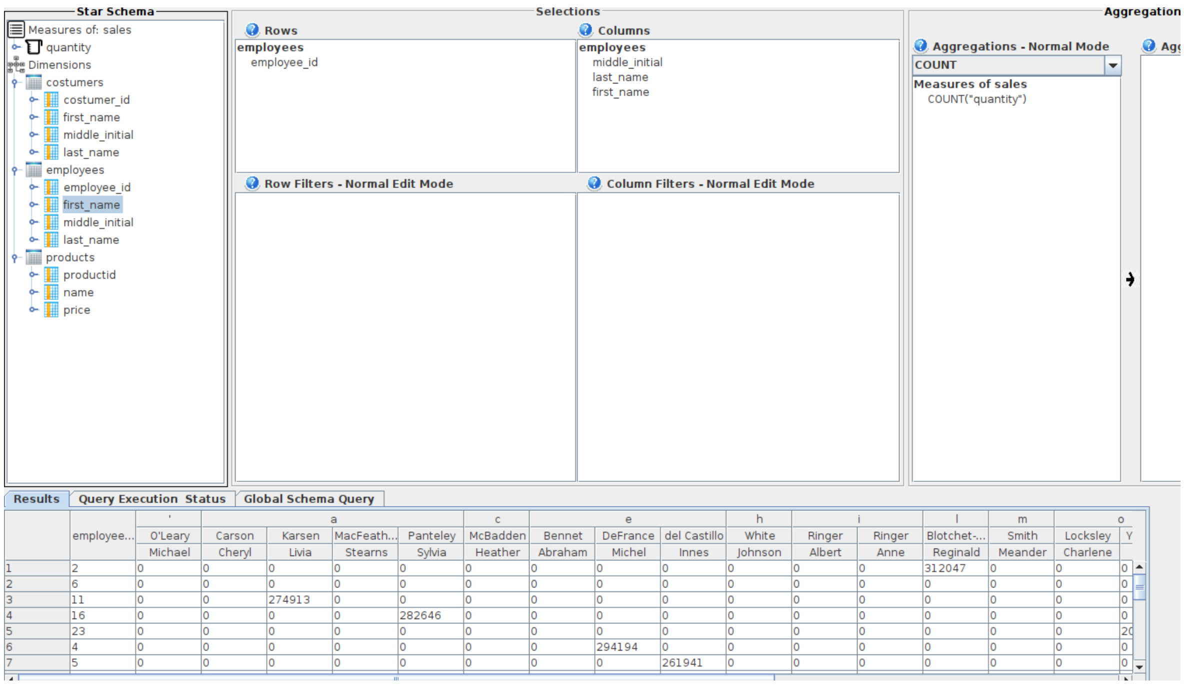
[50] P. O’Neil, E. O’Neil, and X. Chen, “The Star Schema Benchmark (SSB),” no. January, pp. 1–10, 2007. [Online]. Available: http://www.cs.umb.edu/{~}xuedchen/research/publications/StarSchemaB.PDF

[51] E. L. Ratnam, S. R. Weller, C. M. Kellett, and A. T. Murray, “Residential load and rooftop PV generation: an Australian distribution network dataset,” *International Journal of Sustainable Energy*, vol. 36, no. 8, pp. 787–806, sep 2017, doi: 10.1080/14786451.2015.1100196.

Appendix A

Examples Of Queries Submitted In EasyBDI

A.1 Easy BDI Examples using a simple dataset



Star Schema

- Measures of: sales
  - quantity
- Dimensions
  - costumers
    - costumer_id
    - first_name
    - middle_initial
    - last_name
  - employees
    - employee_id
    - first_name
    - middle_initial
    - last_name
  - products
    - productid
    - name
    - price

Selections

Rows: employees — employee_id

Columns: employees — middle_initial, last_name, first_name

Row Filters - Normal Edit Mode

Column Filters - Normal Edit Mode

Aggregation

Aggregations - Normal Mode: COUNT

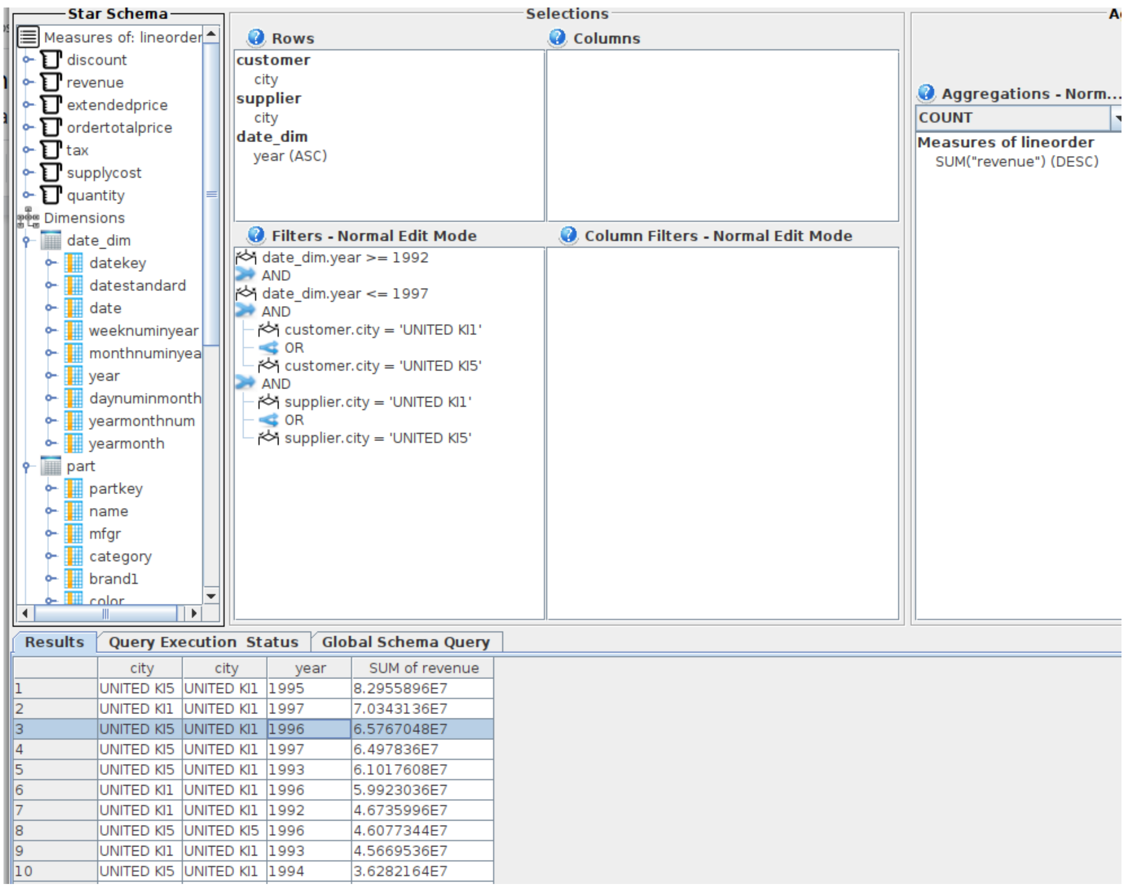
Measures of sales: COUNT("quantity")

Results | Query Execution Status | Global Schema Query

		'	a				c	e			h	i		l	m	o	
	employee...	O'Leary	Carson	Karsen	MacFeath...	Panteley	McBadden	Bennet	DeFrance	del Castillo	White	Ringer	Ringer	Blotchet-...	Smith	Locksley	Y
		Michael	Cheryl	Livia	Stearns	Sylvia	Heather	Abraham	Michel	Innes	Johnson	Albert	Anne	Reginald	Meander	Charlene	
1	2	0	0	0	0	0	0	0	0	0	0	0	0	312047	0	0	0
2	6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	11	0	0	274913	0	0	0	0	0	0	0	0	0	0	0	0	0
4	16	0	0	0	0	282646	0	0	0	0	0	0	0	0	0	0	0
5	23	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2
6	4	0	0	0	0	0	0	0	294194	0	0	0	0	0	0	0	0
7	5	0	0	0	0	0	0	0	0	261941	0	0	0	0	0	0	0

Figure A.1: Creation of a query where the results contain multiple headers using Easy BDI's User Interface.

A.2 Easy BDI Examples Using SSB+ Dataset



customer
  city
supplier
  city
date_dim
  year (ASC)

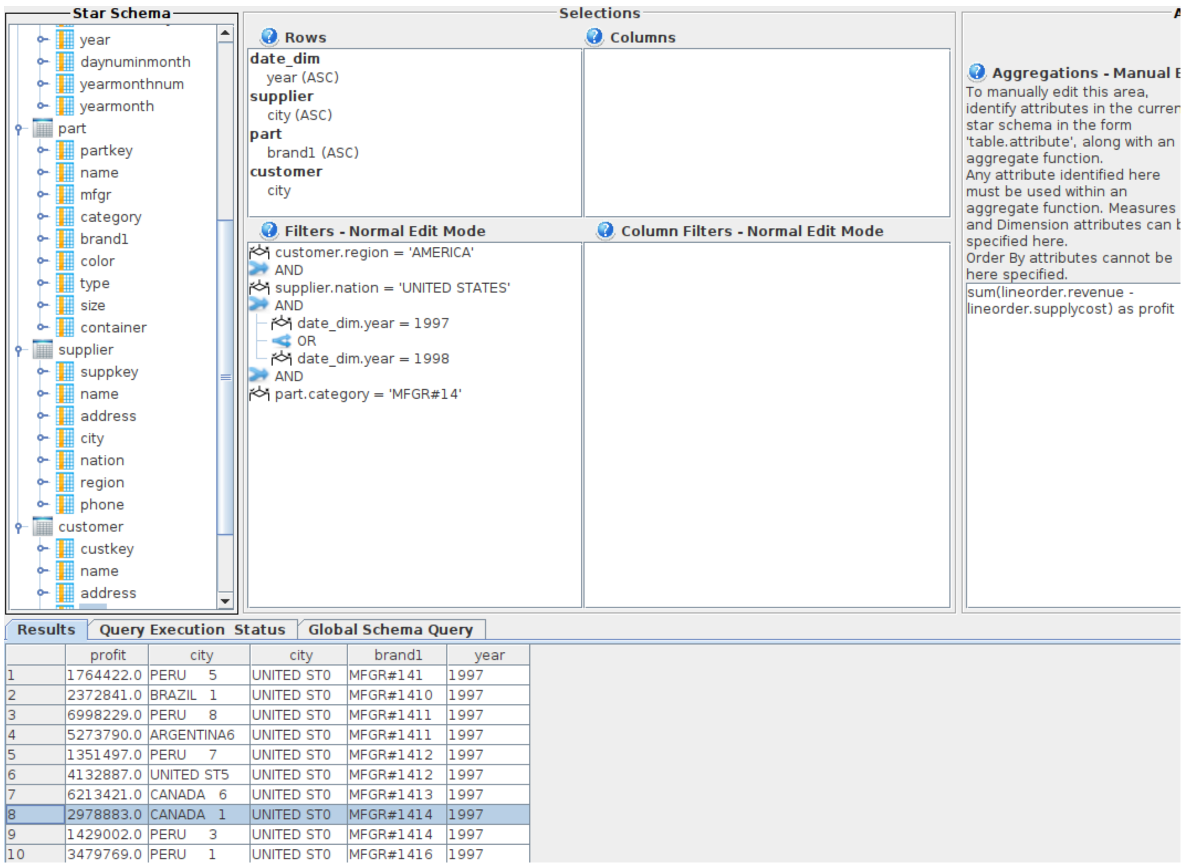
```
date_dim.year >= 1992
AND
date_dim.year <= 1997
AND
    customer.city = 'UNITED KI1'
    OR
    customer.city = 'UNITED KI5'
AND
    supplier.city = 'UNITED KI1'
    OR
    supplier.city = 'UNITED KI5'
```

COUNT

Measures of lineorder
  SUM("revenue") (DESC)

	city	city	year	SUM of revenue
1	UNITED KI5	UNITED KI1	1995	8.2955896E7
2	UNITED KI1	UNITED KI1	1997	7.0343136E7
3	UNITED KI5	UNITED KI1	1996	6.5767048E7
4	UNITED KI5	UNITED KI1	1997	6.497836E7
5	UNITED KI5	UNITED KI1	1993	6.1017608E7
6	UNITED KI1	UNITED KI1	1996	5.9923036E7
7	UNITED KI1	UNITED KI1	1992	4.6735996E7
8	UNITED KI5	UNITED KI5	1996	4.6077344E7
9	UNITED KI1	UNITED KI1	1993	4.5669536E7
10	UNITED KI5	UNITED KI1	1994	3.6282164E7

Figure A.2: Recreation of query model 9 from SSB+ in Easy BDI. The image shows an example of nested filters.


Star Schema

- year
- daynuminmonth
- yearmonthnum
- yearmonth
- part
  - partkey
  - name
  - mfgr
  - category
  - brand1
  - color
  - type
  - size
  - container
- supplier
  - suppkey
  - name
  - address
  - city
  - nation
  - region
  - phone
- customer
  - custkey
  - name
  - address

Selections

Rows

date_dim
year (ASC)
supplier
city (ASC)
part
brand1 (ASC)
customer
city

Columns

Filters - Normal Edit Mode

```
customer.region = 'AMERICA'
AND
supplier.nation = 'UNITED STATES'
AND
    date_dim.year = 1997
    OR
    date_dim.year = 1998
AND
part.category = 'MFGR#14'
```

Column Filters - Normal Edit Mode

Aggregations - Manual E
To manually edit this area, identify attributes in the curren 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 b specified here.
Order By attributes cannot be here specified.

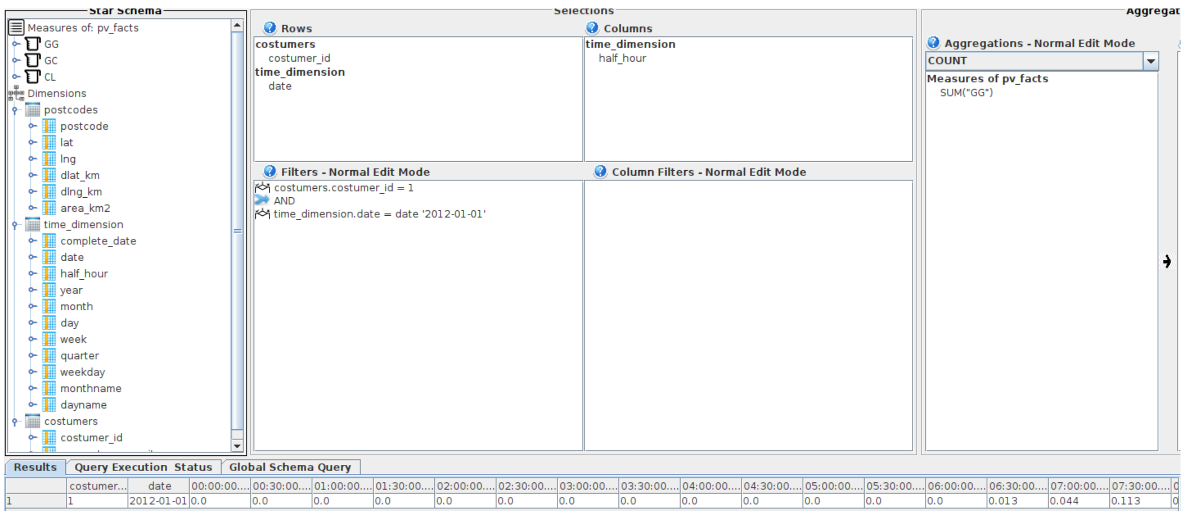
```
sum(lineorder.revenue -
lineorder.supplycost) as profit
```

Results | Query Execution Status | Global Schema Query

	profit	city	city	brand1	year
1	1764422.0	PERU 5	UNITED ST0	MFGR#141	1997
2	2372841.0	BRAZIL 1	UNITED ST0	MFGR#1410	1997
3	6998229.0	PERU 8	UNITED ST0	MFGR#1411	1997
4	5273790.0	ARGENTINA6	UNITED ST0	MFGR#1411	1997
5	1351497.0	PERU 7	UNITED ST0	MFGR#1412	1997
6	4132887.0	UNITED ST5	UNITED ST0	MFGR#1412	1997
7	6213421.0	CANADA 6	UNITED ST0	MFGR#1413	1997
8	2978883.0	CANADA 1	UNITED ST0	MFGR#1414	1997
9	1429002.0	PERU 3	UNITED ST0	MFGR#1414	1997
10	3479769.0	PERU 1	UNITED ST0	MFGR#1416	1997

Figure A.3: Recreation of query model 13 from SSB+ in Easy BDI. The image shows an example of nested filters and manual edition of aggregations.

A.3 Easy BDI Examples Using PV dataset



Rows: costumers
costumer_id
time_dimension
date

Columns: time_dimension
half_hour

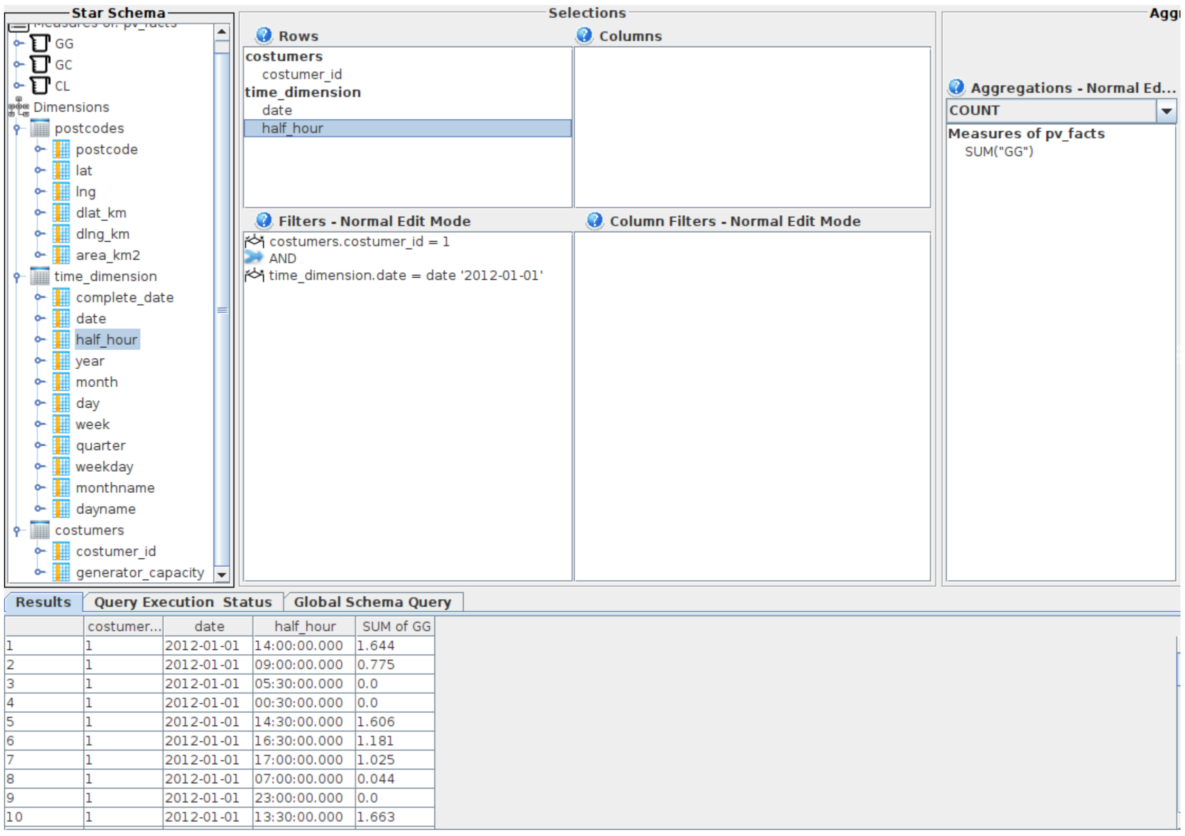
Filters - Normal Edit Mode
costumers.costumer_id = 1
AND
time_dimension.date = date '2012-01-01'

Column Filters - Normal Edit Mode

Aggregations - Normal Edit Mode
COUNT
Measures of pv_facts
SUM("GG")

	costumer...	date	00:00:00....	00:30:00....	01:00:00....	01:30:00....	02:00:00....	02:30:00....	03:00:00....	03:30:00....	04:00:00....	04:30:00....	05:00:00....	05:30:00....	06:00:00....	06:30:00....	07:00:00....	07:30:00....	0
1	1	2012-01-01	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.013	0.044	0.113	0

(a) Energy Production by hour using "hour" as columns.



Rows: costumers
costumer_id
time_dimension
date
half_hour

Columns

Filters - Normal Edit Mode
costumers.costumer_id = 1
AND
time_dimension.date = date '2012-01-01'

Column Filters - Normal Edit Mode

Aggregations - Normal Ed...
COUNT
Measures of pv_facts
SUM("GG")

	costumer...	date	half_hour	SUM of GG
1	1	2012-01-01	14:00:00.000	1.644
2	1	2012-01-01	09:00:00.000	0.775
3	1	2012-01-01	05:30:00.000	0.0
4	1	2012-01-01	00:30:00.000	0.0
5	1	2012-01-01	14:30:00.000	1.606
6	1	2012-01-01	16:30:00.000	1.181
7	1	2012-01-01	17:00:00.000	1.025
8	1	2012-01-01	07:00:00.000	0.044
9	1	2012-01-01	23:00:00.000	0.0
10	1	2012-01-01	13:30:00.000	1.663

(b) Energy Production by hour using "hour" as rows.

Figure A.4: Energy Production of client 1 at 01/01/2012 every half hour using the PV dataset. Two different ways to show data are shown: A.4a pivots the "hour", and A.4b shows the hours as rows.