Why- and How-Provenance in Distributed Environments

Paulo Pintor[1](✉), Rogério Luís de Carvalho Costa[2], and José Moreira[1]

[1] IEETA - University of Aveiro, 3810-193 Aveiro, Portugal
{paulopintor,jose.moreira}@ua.pt
[2] CIIC - Polytechnic of Leiria, 2411-901 Leiria, Portugal
rogerio.l.costa@ipleiria.pt

Abstract. With the emergence of new paradigms in data management and processing like Cloud services, the Internet of Things (IoT), and NoSQL, there is a growing trend for distributing data across multiple platforms and using the technologies most suited for each case according to criteria such as performance and cost. But it also raises new challenges and needs, like understanding the sources, the transformations, and the processes made on the data to infer their quality and reliability. Data provenance becomes particularly relevant in such a context.

This paper presents a solution to deal with why- and how-provenance queries on distributed data sources and different database paradigms. The proposed solution does not require any change to the query execution engine. It uses pure SQL with annotations and an algorithm to build data provenance information from the result obtained by the query. We also present experimental evaluation results obtained using an open-source logical integration tool.

Keywords: Provenance · Data provenance · Databases · Distributed systems

1 Introduction

Data management and processing have been changing over the past years. Several factors have made data increasingly distributed, including the emergence of the Cloud, smart devices, and the Internet of Things (IoT). Also, the rise of open data and data science attracted experts in several domains who became interested in data manipulation and processing, knowledge extraction, and results sharing. This context leads to issues regarding the quality, veracity, completeness, and correctness of data sources, thus increasing the need to understand where data comes from, whether the source is trustworthy, and the transformations made on data. Data provenance is metadata information (annotations) about data origin and transformations made on data and helps solve such issues.

Although there exists a standard (PROV [15], a W3C recommendation) for describing this information in terms of agents, entities, activities, and their relationships, an important research topic is how to disclose provenance information

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2022
C. Strauss et al. (Eds.): DEXA 2022, LNCS 13426, pp. 103–115, 2022.
https://doi.org/10.1007/978-3-031-12423-5_8

in database queries, i.e., to know where query results come from and how they were computed.

Some solutions with different approaches allow obtaining data provenance information from database queries, but all are in centralized environments or specific database management systems. The data provenance issue becomes even more essential in distributed database environments in which several (and possibly heterogeneous) databases are accessed to answer a single user's query.

This paper discusses issues and challenges involving data provenance in distributed and heterogeneous databases. It presents a solution for *how*- and *why-provenance* that does not need to make changes to the database engine nor use system-specific functions and procedures. Hence, our solution can build provenance information for the results of a query independently of the data source type (e.g., file, and relational database and NoSQL databases), which is an important feature when dealing with distributed and heterogeneous data sources.

The following section presents some background and related work. Section 3 discusses data provenance in distributed environments and then describes the proposed solution. Then, Sect. 4 presents results from an experimental evaluation. Finally, Sect. 5 concludes the paper and describes future work.

2 Background and Related Work

This section presents some background on building the provenance of the results of database queries (data provenance) and the problems that arise when working with distributed databases. It also reviews existing related works.

2.1 Data Provenance

In [11], the authors proposed four types of provenance, divided hierarchically: provenance meta-data, information system provenance, workflow provenance, and data provenance. Data provenance aims to collect the provenance information from queries over a database. Due to the fact dealing with databases with specific schemas and because the provenance can be at the tuple level, this type of provenance has requirements that do not appear in other types of provenance.

The three most common types of data provenance are *why*-, *how*- and *where-provenance* [4,5,10]. With the increased interest and research in data provenance, other categories have been proposed, such as *Why-not-provenance* [2,11] and *Which-provenance* [10] and perhaps more might follow. The focus of this paper is on *why*- and *how-provenance.*

Why-Provenance – Collects all the inputs that contributed to a query result [3–5,10]. The technique to collect why-provenance information is called Witnesses basis. It is a set of tuples that contribute to a particular result. These tuples are called witnesses of the production of the resulting tuple.

Based on the definition in [3,5], given a database I, a query Q over I and a tuple t in *Q(I)*, an instance of $I' \subseteq I$ is a witness for t if $t \in Q(I')$. This can be denoted as: $Why(Q, I, t) = \{I' \subseteq I | t \in Q(I')\}$

Orderspt			
sname	dest	vehicle	provtoken
LisboaStore	Porto	Train	tk3
LisboaStore	Braga	Truck	tk4
PortoStore	Braga	Train	tk5
PortoStore	Madrid	Airplane	tk6

Fig. 1. An example of a table with orders.

For instance, consider the table in Fig. 1 representing orders. The field "sname" is the supplier name, "dest" is the destination, "vehicle" is the type of vehicle, and the tuple identifier is called "provtoken".

$$Q1 : \pi_{dest}\sigma_{dest=\text{“}Braga''}(\pi_{sname,dest}Orderspt \bowtie \pi_{vehicle,dest}Orderspt)$$

The *Why-provenance* is the set of tuples with all the possible combinations, without duplicates. The result of Q1 is displayed in Table 1 and shows that the witnesses of *"Braga"* are *tk4* and *tk5* alone or the conjugation of both.

Table 1. Result of Q1

dest	why	how
Braga	{tk4}, {tk4,tk5}, {tk5}	$(tk4 \otimes tk4) \oplus (tk4 \otimes tk5) \oplus (tk5 \otimes tk4) \oplus (tk5 \otimes tk5)$

How-Provenance – Explains how the inputs contributed to the result and is obtained using algebraic identities and polynomials (semirings) [3–5,9,10,16]. Each tuple must also have an annotation called prove token.

A semiring is defined as $(K, 0, 1, \oplus, \otimes)$ where K is a set of data elements that will be annotated using the constants 0 and 1. Given a query Q if the tuple t contributes to the output result is annotated with 1, otherwise is annotated with 0. The binary operators $\oplus, \otimes$ are used as alternative $\oplus$ and as joint $\otimes$.

Different types of semirings can be used to achieve different answers. For *how-provenance* the universal semiring or how-semiring $(N[X], 0, 1, \oplus, \otimes)$ is used. As stated in [9], *unions* are associative and commutative operations and are represented by $\oplus$. The *joins* also have those two properties, but they are also distributive over *unions* and they represented by $\otimes$. The *projections* and *selections* are also commutative among themselves.

Hence regarding the result present in Table 1 about *How-Provenance*, *"Braga"* is obtained by the conjugation of *tk4* with itself (*join*), or (*union*) by the conjugation of *tk5* with itself (*join*), or (*union*) by the conjugation of *tk4* and *tk5* (*join*) or (*union*) by the conjugation of *tk5* and *tk4*.

Regarding the *joins* properties, more specifically, the distributive property of the results in Table 1 can also be simplified to: $(tk4 \oplus tk5) \otimes (tk4 \oplus tk5)$. In [16] it is proposed to use m-semirings with the operator *monus* ($\ominus$) to be able to give the provenance for non-monotone queries.

2.2 Distributed Databases

While Multi-Model databases allow having different types of models (e.g., graph, key-value, and documents) in the same Database Management System (DBMS) [13], Polystore databases are built on the top of multiple storage engines that are integrated and enable to query multiple data sources using different models and paradigms [7].

Using distributed query engines (e.g., Presto [18]), users may query over distributed and heterogeneous databases using standard SQL language. The query engines act as mediators between the querying interface and the underlying systems, but they do not deal with distribution transparency, i.e., the location of each data structure (e.g., table) must be included in the query. This forces users to have deep knowledge about the different data sources and their schemas.

Distribution transparency can be achieved by logical data integration. It commonly comprises a high-level global model, i.e., a Global Conceptual Schema (GCS), and Local Conceptual Schemas (LCS), which represent the physically distributed data [21]. The GCS stores the information about how to link global and local entities. There are no Extract-Transform-Load (ETL) methods. Queries are written considering the global entities, thus hiding distribution complexity from the end-users. This approach is especially useful in scenarios where the users need data to always be up to date.

But the logical integration requires the mapping between global and local entities. One global entity may match a single entity of a specific data source (local entity). But a single logical global entity may map to two or more local entities (i.e., partitioning). In **horizontal partitioning**, a global entity maps to two or more local entities (i.e., partitions) storing distinct instances of conceptually related data. For example, a global entity representing customers' data can map to two local entities, one storing data about customers from Europe and another storing data about customers from America. Thus, the global entity is the union of the local partitions. In **vertical partitioning**, a global entity maps to two or more local entities (i.e., partitions), and each partition stores distinct features (attributes) of the global entity. Thus, to retrieve all the attributes of a global entity instance (e.g., tuple), one should join data from two or more local entities (i.e., vertical partitions). For example, a global entity representing customers' information can map to two local entities at distinct sources, one storing customers' mailing addresses and another storing customers' billing data.

Figure 2 exemplifies partitioning over the table *Orderspt* represented in Fig. 1. In Fig. 2, the table is split into two, one physically stored in Portugal and the other in Spain. The data in Portugal represent the stores located in Portugal and the same for Spain. Figure 2 also represents the Stores tables, which contain the store's name, localization, and e-mail.

In a distributed database scenario, one must obtain data provenance information considering all the data sources involved in the distributed query. Thus it is not possible to use plugins for a specific database, and in the case of the use of a mediator, it needs to deal with different types of databases.

Portugal				Spain			
Storespt				Storesen			
name	city	email	provtoken	name	city	email	provtoken
LisboaStore	Lisboa	ls@store.pt	t1	MadridStore	Madrid	ms@store.en	tk1
PortoStore	Porto	ps@store.pt	t2	BarcelonaStore	Barcelona	bs@store.en	tk2
Orderspt				Ordersen			
sname	dest	vehicle	provtoken	sname	dest	vehicle	provtoken
LisboaStore	Porto	Train	tk3	BarcelonaStore	Madrid	Truck	tk7
LisboaStore	Braga	Truck	tk4	BarcelonaStore	Braga	Train	tk8
PortoStore	Braga	Train	tk5	MadridStore	Barcelona	Truck	tk9
PortoStore	Madrid	Airplane	tk6	MadridStore	Bilbao	Truck	tk10

Fig. 2. An example of a distributed environment for stores and orders.

2.3 Related Work

In the literature, there are several works with methods to apply W3C PROV, most of them in Workflows [12,20]. There are also works to describe Geospatial datasets in distributed environments [6].

In terms of data provenance there are examples such as ProvSQL [17], Perm [8], and GProM [1]. These are of solutions to visualize information about *where-*, *how-* and *why-provenance* and solutions for probabilistic query evaluation. ProvSQL is a lightweight extension for PostgreSQL that supports several relational database formalisms, including *where-provenance* and *how-provenance*. GProM approached it with a middleware solution for Oracle, SQLite, and PostgreSQL, but only in a centralized environment. Perm promotes rewriting the queries. However, extending these formalisms to distributed environments with different data sources (e.g., NoSQL and semi-structured) is an open issue.

The transparency in distributed environments integration helps the users to have a high-level model of the domain. Hence, they do not need to be concerned about how the data sources are connected and distributed or their heterogeneity. Nevertheless, users continue to need the information to infer the veracity and quality of the result, making the use of data provenance essential.

3 Provenance in Distributed Databases

This section shows how to obtain *how-* and *why-provenance* in a distributed databases environment using SQL. In [17], ProvSQL is an extension to PostgresSQL that changes the query execution engine. Our approach is non-intrusive and aims to work independently of the database and without changing the engine. This improves portability because our solution uses only standard SQL and not functions or stored procedures coded in languages that depend on the database management system. Furthermore, nowadays there are distributed query engines that can create an abstraction layer across data sources of different paradigms using SQL, our solution can also be used to build data provenance over distributed and heterogeneous databases (e.g., relational and NoSQL).

3.1 Architecture

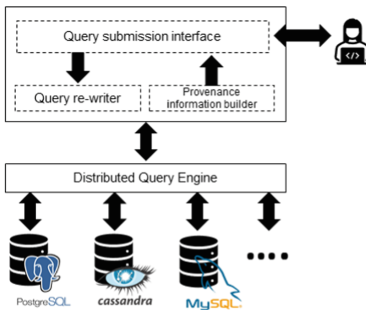

Fig. 3. Architecture and main components.

The architecture of the proposed solution is depicted in Fig. 3. The user submission interface allows users to write the queries to retrieve data from one or more databases. It is assumed that the mapping between global entities and local entities in source databases is known a priori, as discussed in the previous section. Then, the Query re-writer adds annotations to the query to obtain the provenance data and submits the request to the source databases through a distributed query engine. The latter transforms the query into sub-queries that are sent to be executed in the source databases. The distributed query execution engine gets query results containing provenance tokens from each data source and assembles a global query execution result. Then, the engine sends such a result to the Provenance Information Builder, which builds the provenance sentences and sends them to the user together with the user's query results.

For instance, considering that a user wants to execute query Q1 in a distributed environment using the data displayed in Fig. 2, the query would be as follows.

$$Q2 : \pi_{dest} \sigma_{dest=\text{``}Braga''} (\pi_{sname,dest} Orderspt \cup \pi_{sname,dest} Ordersen) \bowtie \\ (\pi_{vehicle,dest} Orderspt \cup \pi_{vehicle,dest} Ordersen)$$

Despite that the user might only see the global entities, the *unions* in the query are required to retrieve the data from the two local data sources. The provenance information resulting from Q2 is as follows.

Why-Provenance – {{p.orderspt:tk4,p.orderspt:tk5}, {p.orderspt:tk4,p.orderspt:tk8}, {p.orderspt:tk4}, {p.orderspt:tk5}, {p.orderspt:tk5,p.orderspt:tk8}, {p.orderspt:tk8}}

How-Provenance – (p.orderspt:tk4 ⊗ (p.orderspt:tk5 ⊕ c.ordersen:tk8)) ⊕ (p.orderspt:tk4 ⊗ p.orderspt:tk4) ⊕ (p.orderspt:tk5 ⊗ (p.orderspt:tk5 ⊕ c.ordersen:tk8)) ⊕ (p.orderspt:tk5 ⊗ p.orderspt:tk4) ⊕ (c.ordersen:tk8 ⊗ (p.orderspt:tk5 ⊕ c.ordersen:tk8)) ⊕ (c.ordersen:tk8 ⊗ p.orderspt:tk4)

Since we are in a distributed environment, and the data provenance information is given with tokens, we add additional information. The format of the provenance results has three parts separated by dot and colon characters: the first is the data source (*"p"* for PostgreSQL and *"c"* for Cassandra, in the example), the second is the table name (*orderspt* or *ordersen*) and the third is the provenance token.

3.2 Annotations

The solution proposed in this work has two premises. First, each data element (e.g., a token) in a data source must have a unique identifier as shown in [5,10,16]. The annotations can be seen as provenance tokens and they support the witness basis theory for *why-provenance* and the semiring theory for *how-provenance*.

As almost all databases have a function to create Universally Unique Identifiers (UUID), these are a natural choice to be used as provenance tokens. If the system does not provide UUIDs, it is needed to create a column with a unique identifier, e.g., a number or a string.

We also assume the existence of a distributed query engine (as shown in Fig. 3) that supports the standard SQL function *Listagg* [14] or a similar one. This function allows to aggregate/concatenate string values from a group of rows and separate them with a delimiter.

Our approach is to add annotations to user queries to retrieve provenance information from the data sources together with the query results themselves. The annotations depend on the operators in the query.

Distinct, Union and Group By – The annotation consists of adding columns to the user queries. In the case of a *distinct* clause, as a tuple t in a query result $Q(I)$ may have several witnesses ($I' \in I$), we use the function *listagg* to aggregate all the tokens of I' into a single value. The tokens are separated using the special character ⊙. The *distinct* clause must be removed from the query because, as each tuple has a unique identifier (token), it would prevent the aggregation of the witnesses i' of t in a single tuple. The annotation for the operator *union* is similar to the *distinct* clause because there are also no duplicates in the result of a query, and in the case of a *group by*, we need to use a different separator, in this case ⦶. The different separators will help the algorithm to combine the tokens properly.

Join – In this case, it is not necessary to use the function *listaagg*, only add the tokens columns for the tables involved in the *join*. If the query is composed of sub-queries, it is required that the sub-queries have the tokens columns. For example, if we want to *join* two *unions*, we need to apply the *union* transformation explained above and add the *unions* token columns to the *join* projection.

The splitters and the columns for the *joins* will allow the built provenance information algorithm to interpret it by splitting and joining the columns and applying the *how-* and *why-provenance* methods as defined in the literature. All the columns with annotations have the name *"prov"*.

3.3 Build Provenance Information

Algorithm 1 demonstrates how the annotations are processed to obtain *how-* and *why-provenance*. Even though most of the times it is possible to derive *why-* from *how-provenance*, we opted for separate approaches in this solution. This option was based on [10], where it is demonstrated that the derivation is not straightforward. Also, if we utilize the m-semiring technique [17], the derivation becomes even more complex.

Before using the functions *HowProvenance* and *WhyProvenance*, the columns with the annotations are aggregated in an array of arrays, which will be the input parameter of both functions. As an example, the first column of *"prov"* can contain *"tk4;tk3|tk5"* and the second column *"tk8|tk9;tk10"*, and the array will have the final result *[["tk4;tk3", "tk8"], ["tk5", "tk9;tk10"]]*. This avoids the repeated looping through the annotations columns for each function.

The *HowProvenance* function starts looping through the input array and initiates variables *"temp"* and *"paren"*. In the second loop if the tokens has the character ";", it replaces the character by $\oplus$ because it means a *union* or a *distinct*. Between the replace function, it adds to the string *"temp"* the parenthesis and the $\otimes$ because the next token is part of a *join*.

If the character is not present, it adds the token and $\otimes$ to the string *"temp"* for the same reason as above, and the boolean *"paren"* helps place the parenthesis in the right place. In lines 12 and 13, the extra characters are removed from *"temp"* and added to an array since the second loop ended. The function will return a string that concatenates all the array positions with $\oplus$. This last step uses the $\oplus$ because the *"aggTokens"* array is created by splitting the *group by* character clause.

To obtain the *why-Provenance* we need to apply the distributive property to the *how-provenance's* result and apply the rules of witnesses basis. Thus, we need two nested loops again because the *WhyProvenance* input parameter is an array of arrays. In the first iteration of the second loop (lines 28 to 32), we populate the array *"why"* with a *set* for every token obtained from the split by the character ";".

In the subsequent iterations, we need to apply the distribution. If the array *"why"* length is higher than the length of the split array, for each *set* in *"why"* we add the tokens obtained from the split (lines 34 to 38). Else for each token in the split, we loop through the *"why"* array and copy the *"why"* to a temporary variable, add to this temporary variable the token in the split and add it to a temporary array. In the end, *"why"* will be equal to the temporary array. The return clause will return a string constructed by the function *CheckDoubles* that also removes the possible similar sets.

Algorithm 1. How- and Why-provenance algorithm

```
1: function HowProvenance(aggTokens)
2:     how ← []
3:     for each agt ∈ aggTokens do
4:         temp ←''; paren ← False;
5:         for each t ∈ agt do
6:             if t ⊂';' then
7:                 temp ← temp +' ('+t.replace(';' ,' ⊕')+')⊗'
8:             else
9:                 if notparen then
10:                    temp ←' ('+temp + t +' ⊗'
11:                    paren ← True
12:                else
13:                    temp ← temp + t+')⊗'
14:                    paren ← False
15:                end if
16:            end if
17:        end for
18:        temp ←' ('+temp.RemoveExtraChars()+')'
19:        how.add(temp)
20:    end for
21:    return how.join(⊕)
22: end function
23:
24: function WhyProvenance(aggTokens)
25:    why ← []
26:    for each agt ∈ aggTokens do
27:        for i = 0, 1, . . . length(agt) do
28:            if i == 0 then
29:                for each t ∈ agt[i].split(';' ) do
30:                    tSet ← Set(); tSet.add(t); why.add(tSet)
31:                end for
32:            else
33:                if length(agt[i].split(';' )) < lenght(why) then
34:                    for each wt ∈ why do
35:                        for each t ∈ agt[i].split(';' ) do
36:                            wt.add(t)
37:                        end for
38:                    end for
39:                else
40:                    copyWT ← []
41:                    for each t ∈ agt[i].split(';' ) do
42:                        for each wt ∈ why do
43:                            temp ← wt.copy(); temp.add(t); copyWT.add(temp)
44:                        end for
45:                    end for
46:                    why = copyWT
47:                end if
48:            end if
49:        end for
50:    end for
51:    return CheckDoubles(why)
52: end function
```

4 Experimental Evaluation

As proof of concept for our solution, we use EasyBDI [19], an open-source prototype for logical integration of distributed databases that provides mapping functionalities between local and global schemas. EasyBDI has a graphical interface that allows the users to query over the global schemas without writing SQL commands. The interface provides different frames where the user can drag and drop entity columns and the operators (e.g., group by) to use on the query.

When the user executes the query, EasyBDI builds the SQL command based on the mapping between GCS and LCSs and submmits it to Trino, a distributed query execution engine. Since the software is open-source, we modified the query generation module to add the annotation columns when performing the query build. We also applied the proposed algorithm to the query execution result.

As dataset, we used the tables represented in Fig. 2. PostgreSQL stores the data about Portugal, and the ones about Spain is in a Cassandra database. EasyBDI allows the user to identify mapping types. In this case, there is a horizontal mapping (which means that the global entity is horizontally partitioned through two data sources), i.e., the global entity representing the orders maps to structures in Cassandra and PostgreSQL. The first example is a *distinct* query to obtain all the vehicles used in orders. The executed query is:

SELECT vehicle, listagg(prov, ';') WITHIN GROUP (ORDER BY vehicle) as prov FROM (SELECT sname, dest, vehicle, listagg(provtoken, ';') WITHIN GROUP (ORDER BY sname) as prov FROM(SELECT sname, dest, vehicle, **provtoken** FROM postgresql.public.orderspt UNION SELECT sname, dest, vehicle, **provtoken** FROM cassandra.stkspace.ordersen **) GROUP BY sname, dest, vehicle) GROUP BY vehicle**

In the above query, the clauses in bold are the ones we added to the query. Starting with the sub-query, the local schemas' union is needed to obtain the global entity. Since we add *"provtoken"* to the tables and they might be different, the *union* result would be erroneous without the *group by* clause. Thus, we also add the *group by* clause and the *listagg* function. In the main query, the *distinct* clause has been removed, and we used a *group by* clause again with the column in the *distinct* and the *listagg* to aggregate the tokens. The result obtained is the following:

For *"Airplane"*, the provenance is simple. We have only one token as a witness for the *why-provenance* and the same token for *how-provenance*. For *"Train"* and *"Truck"* we have different witnesses, and we can also obtain each row using one of the tokens. Since in the *How-provenance* column the tokens are separated by $\oplus$, we can use one of the tokens only to obtain the rows.

Results | Query Execution Status | Global Schema Query

	vehicle	why
1	Airplane	{{c.ordersen:tk6}}
2	Truck	{{c.ordersen:tk10}, {p.orderspt:tk4}, {c.ordersen:tk7}, {c.ordersen:tk9}}
3	Train	{{p.orderspt:tk3}, {p.orderspt:tk5}, {c.ordersen:tk8}}

Results | Query Execution Status | Global Schema Query

	vehicle	how
1	Airplane	(c.ordersen:tk6)
2	Truck	(c.ordersen:tk10 ⊕ p.orderspt:tk4 ⊕ c.ordersen:tk7 ⊕ c.ordersen:tk9)
3	Train	(p.orderspt:tk3 ⊕ p.orderspt:tk5 ⊕ c.ordersen:tk8)

Fig. 4. The result of the distinct query.

The following query is a *join* between the stores and orders to obtain the orders' destination and the stores' e-mail responsible for the orders. The *unions* are simplified, since they are equal to the last query, just now for the two tables.

SELECT s.email, o.dest, **s.prov as prov, o.prov as prov** FROM (– UNION STORES –) s, (– UNION ORDERS –) o WHERE s.name = o.sname

The *unions* are again applied to obtain the GCS. We added the annotations' columns for the tables/view/query involved in the join to the query projection. The result in Fig. 5 shows that all *why-provenance's* tokens are in pairs of witnesses: in order to obtain any row, we need both of the tokens. In contrast with the query of Fig. 4, now the tokens are separated by ⊗ in *how-provenance*, each means that we need a join between both tokens.

Results | Query Execution Status | Global Schema Query

	email	dest	why	how
1	ls@store.pt	Porto	{{p.storespt:tk1 , p.orderspt:tk3}}	(p.storespt:tk1 ⊗ p.orderspt:tk3)
2	ls@store.pt	Braga	{{p.storespt:tk1 , p.orderspt:tk4}}	(p.storespt:tk1 ⊗ p.orderspt:tk4)
3	ps@store.pt	Madrid	{{p.storespt:tk2 , p.orderspt:tk6}}	(p.storespt:tk2 ⊗ p.orderspt:tk6)
4	ps@store.pt	Braga	{{p.storespt:tk2 , p.orderspt:tk5}}	(p.storespt:tk2 ⊗ p.orderspt:tk5)
5	ms@store.pt	Barcelona	{{c.storesen:tk1 , c.ordersen:tk9}}	(c.storesen:tk1 ⊗ c.ordersen:tk9)
6	ms@store.pt	Bilbao	{{c.storesen:tk1 , c.ordersen:tk10}}	(c.storesen:tk1 ⊗ c.ordersen:tk10)
7	bs@store.pt	Madrid	{{c.storesen:tk2 , c.ordersen:tk7}}	(c.storesen:tk2 ⊗ c.ordersen:tk7)
8	bs@store.pt	Braga	{{c.storesen:tk2 , c.ordersen:tk8}}	(c.storesen:tk2 ⊗ c.ordersen:tk8)

Fig. 5. The result of the query with join

The last query example is a *group by* the previous query applied to *"dest"*. Since it is a *group by*, we need to use the *listagg* function in the *joins*' columns.

SELECT o.dest, **listagg(s.prov, '|') WITHIN GROUP (ORDER BY o.dest) as prov, listagg(p.prov, '|') WITHIN GROUP (ORDER BY o.dest) as prov** FROM (– UNION STORES –) s, (– UNION ORDERS –) o WHERE s.name = o.sname GROUP BY o.dest

Results | Query Execution Status | Global Schema Query

	dest	why
1	Barcelona	{c.storesen:tk1 , c.ordersen:tk9}
2	Bilbao	{c.storesen:tk1 , c.ordersen:tk10}
3	Braga	{{c.storesen:tk2 , c.ordersen:tk8}, {p.storespt:tk1 , p.orderspt:tk4}, {p.storespt:tk2 , p.orderspt:tk5}}
4	Madrid	{{c.storesen:tk2 , c.ordersen:tk7}, {p.storespt:tk2 , p.orderspt:tk6}}
5	Porto	{p.storespt:tk1 , p.orderspt:tk3}

	dest	how
1	Barcelona	(c.storesen:tk1 ⊗ c.ordersen:tk9)
2	Bilbao	(c.storesen:tk1 ⊗ c.ordersen:tk10)
3	Braga	((c.storesen:tk2 ⊗ c.ordersen:tk8) ⊕ (p.storespt:tk1 ⊗ p.orderspt:tk4) ⊕ (p.storespt:tk2 ⊗ p.orderspt:tk5))
4	Madrid	((c.storesen:tk2 ⊗ c.ordersen:tk7) ⊕ (p.storespt:tk2 ⊗ p.orderspt:tk6))
5	Porto	(p.storespt:tk1 ⊗ p.orderspt:tk3)

Fig. 6. The result of the group by query

As demonstrated in Fig. 6, some rows now have more than one pair of witnesses for the *why-provenance*. *How-provenance* column shows that it is possible to *join* different tokens to obtain the rows. In the result of destinations *"Braga"* and *"Madrid"*, we can see that the result can be obtained from the two databases because both *"why"* and *"how"* have tokens from the two sources.

5 Conclusions and Future Work

This work discusses data provenance in distributed environments, which is essential to infer the data's veracity and quality.

We present a solution to generate *how-* and *why-provenance* using pure SQL queries with annotations and an algorithm to build the provenance information. It is a non-intrusive solution that does not require any change to the distributed query execution engine. Also, it is not specific to any database system or model. We also present an implementation of our proposals on EasyBDI. It is a logical database integration tool based on which users query entities from global schemas that abstract the data organization on each data source. There is no materialization. Distributed query processing and provenance data generation are done on the fly, without materializations.

In future work, we plan to study how to generate other types of provenance (e.g., *where-provenance*) following the same logic used here. Since we are working with distributed environments, another issue is how to generate provenance information in contexts where materializations are used for database integration and analytic processing.

Acknowledgments. Paulo Pintor has a research grant awarded by the Portuguese public agency for science, technology and innovation FCT - Foundation for Science and Technology - under the reference 2021.06773.BD. This work is partially funded by National Funds through the FCT under the Scientific Employment Stimulus - Institutional Call - CEECINST/00051/2018, and in the context of the projects UIDB/04524/2020 and UIDB/00127/2020.

References

1. Arab, B.S., Feng, S., Glavic, B., Lee, S., Niu, X., Zeng, Q.: GProM - a swiss army knife for your provenance needs. IEEE Data Eng. Bull. **41**(1), 51–62 (2018). http://sites.computer.org/debull/A18mar/p51.pdf
2. Bidoit, N., Herschel, M., Tzompanaki, A.: Efficient computation of polynomial explanations of why-Not questions. In: International Conference on Information and Knowledge Management, Proceedings 19–23 October 2015, pp. 713–722 (2015). https://doi.org/10.1145/2806416.2806426
3. Buneman, P., Khanna, S., Tan, W.C., Chiew, W.: Why and where: a characterization of data provenance. Comput. Sci. **1973**, 316–330 (2001)
4. Buneman, P., Tan, W.C.: Data provenance: what next? SIGMOD Rec. **47**(3), 5–16 (2018). https://doi.org/10.1145/3316416.3316418

5. Cheney, J., Chiticariu, L., Tan, W.C.: Provenance in databases: why, how, and where. Found. Trends Databases **1**, 379–474 (2007). https://doi.org/10.1561/1900000006
6. Closa, G., Masó, J., Proß, B., Pons, X.: W3C PROV to describe provenance at the dataset, feature and attribute levels in a distributed environment. Comput. Environ. Urban Syst. **64**, 103–117 (2017). https://doi.org/10.1016/j.compenvurbsys.2017.01.008, http://dx.doi.org/10.1016/j.compenvurbsys.2017.01.008
7. Duggan, J., et al.: The BigDAWG polystore system. SIGMOD Rec. **44**(2), 11–16 (2015). https://doi.org/10.1145/2814710.2814713
8. Glavic, B., Alonso, G.: Perm: processing provenance and data on the same data model through query rewriting. In: Proceedings of the International Conference on Data Engineering, pp. 174–185. IEEE, Shanghai, China (2009). https://doi.org/10.1109/ICDE.2009.15
9. Green, T.J., Karvounarakis, G., Tannen, V.: Provenance semirings. In: Proceedings of the Twenty-Sixth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, pp. 31–40, PODS 2007. Association for Computing Machinery, New York, NY, USA (2007). https://doi.org/10.1145/1265530.1265535
10. Green, T.J., Tannen, V.: The semiring framework for database provenance. In: Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, pp. 93–99, PODS 2017. Association for Computing Machinery, New York, NY, USA (2017). https://doi.org/10.1145/3034786.3056125
11. Herschel, M., Diestelkämper, R., Ben Lahmar, H.: A survey on provenance: What for? What form? What from? VLDB J. **26**(6), 881–906 (2017). https://doi.org/10.1007/s00778-017-0486-1
12. Kock-Schoppenhauer, A.K., Hartung, L., Ulrich, H., Duhm-Harbeck, P., Ingenerf, J.: Practical extension of provenance to healthcare data based on the W3C PROV standard. Stud. Health Technol. Inform. **253**, 28–32 (2018). https://doi.org/10.3233/978-1-61499-896-9-28
13. Lu, J., Holubová, I.: Multi-model databases: a new journey to handle the variety of data. ACM Comput. Surv. **52**, 1–38 (2019). https://doi.org/10.1145/3323214
14. Michels, J., et al.: The new and improved SQL: 2016 standard. SIGMOD Rec. **47**, 51–60 (2018). https://doi.org/10.1145/3299887.3299897
15. Moreau, L., Groth, P., Cheney, J., Lebo, T., Miles, S.: The rationale of PROV. J. Web Semant. **35**, 235–257 (2015). https://doi.org/10.1016/j.websem.2015.04.001
16. Senellart, P.: Provenance and probabilities in relational databases: from theory to practice. SIGMOD Rec. **46**, 5–15 (2017). https://doi.org/10.1145/3186549.3186551
17. Senellart, P., Jachiet, L., Maniu, S., Ramusat, Y.: ProvSQL: provenance and probability management in PostgreSQL. Proc. VLDB Endow. **11**(12), 2034–2037 (2018). https://doi.org/10.14778/3229863.3236253
18. Sethi, R., et al.: Presto: SQL on everything. In: 2019 IEEE 35th International Conference on Data Engineering (ICDE), pp. 1802–1813 (2019). https://doi.org/10.1109/ICDE.2019.00196
19. Silva, B., Moreira, J., de Costa, R.L.C.: EasyBDI: near real-time data analytics over heterogeneous data sources. In: Advances in Database Technology - EDBT 2021-March, pp. 702–705 (2021). https://doi.org/10.5441/002/edbt.2021.88
20. Zhang, M., Jiang, L., Zhao, J., Yue, P., Zhang, X.: Coupling OGC WPS and W3C PROV for provenance-aware geoprocessing workflows. Comput. Geosci. **138**, 104419 (2020). https://doi.org/10.1016/j.cageo.2020.104419
21. Özsu, M.T., Valduriez, P.: Principles of Distributed Database Systems. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-26253-2