LRI

RAPPORT DE RECHERCHE

A HIGH LEVEL QUERY LANGUAGE FOR BIG DATA ANALYTICS

SPYRATOS N / SUGIBUCHI T

Unité Mixte de Recherche 8623
CNRS-Université Paris Sud – LRI

11/2014

Rapport de Recherche N° 1575

CNRS – Université de Paris Sud
Centre d'Orsay
LABORATOIRE DE RECHERCHE EN INFORMATIQUE
Bâtiment 650
91405 ORSAY Cedex (France)

A High Level Query Language for Big Data Analytics

Nicolas Spyratos
Laboratoire de Recherche en Informatique, UMR8623 of CNRS, Université Paris-Sud 11, France
Tsuyoshi Sugibuchi
Internet Memory Research, France

Abstract

In this paper, we propose a high level query language for expressing analytic queries over big data sets. A main contribution of our approach is the clear separation between the conceptual and the physical level. An analytic query and its answer are defined at the conceptual level independently of the nature and location of data. The abstract definitions are then mapped to lower level evaluation mechanisms, taking into account the nature and location of data, as well as other related aspects. Our overall objective is to have query formulation done on an abstract level, while actual query evaluation can adapt to the evaluation mechanisms offered in each case. To achieve this objective it is necessary to raise the level of abstraction, by providing a sound mathematical basis for the mapping of queries to lower level evaluation mechanisms. However, raising the level of abstraction offers three advantages: (a) useful insights into the process of data analytics in general, and MapReduce in particular (b) a formal approach to the rewriting of analytic queries and the generation of query execution plans and (c) the possibility of leveraging structure and semantics in data in order to improve performance. We emphasize that, although theoretical in nature, this work uses only basic and well known mathematical concepts (namely functions and set partitions). \end{abstract}

Category: H.2.4 {Database Management}{Systems}
Terms: MapReduce, Query Language, Data Analytics

Résumé

Dans cet article nous proposons un langage de requêtes de haut niveau pour l'analyse de données massives. Une des contributions principales de notre approche est la séparation claire entre le niveau conceptuel et le niveau physique. Une requête analytique et sa réponse sont définies au niveau conceptuel, indépendamment de la nature et de l'emplacement des données sous-jacentes. Les définitions abstraites sont ensuite traduites vers des mécanismes d'évaluation à un niveau inférieur, en tenant compte de la nature et de l'emplacement des données, ainsi que d'autres aspects pertinents à l'évaluation de requêtes. Notre objectif principal est de pouvoir formuler des requêtes à un niveau abstrait, et adapter leur évaluation aux mécanismes disponibles dans chaque environnement physique. Pour atteindre cet objectif il est nécessaire de raisonner à un niveau abstrait en se servant d'une base mathématique solide pour pouvoir ensuite traduire les requêtes vers des mécanismes d'évaluation concrets. Cette approche offre les avantages suivants : (a) une meilleure compréhension du processus d'analyse de données de manière générale, et de l'analyse de données suivant MapReduce en particulier (b) une approche formelle de la réécriture des requêtes analytiques et de la génération de plans d'évaluation de telles requêtes et (c) la possibilité de s'appuyer sur la structure et sur la sémantique de données pour améliorer les performances. A noter que, bien que notre approche soit de nature théorique, elle ne fait appel qu'à des notions mathématiques élémentaires (notamment des fonctions et des partitions d'un ensemble).

Catégorie : H.2.4 {Database Management}{Systems}
Termes : MapReduce, Query Language, Data Analytics

A High Level Query Language for Big Data Analytics

Nicolas Spyratos[1], Tsuyoshi Sugibuchi[2]

[1] Laboratoire de Recherche en Informatique, UMR8623 of CNRS, Université Paris-Sud 11, France Nicolas.Spyratos@lri.fr
[2] Internet Memory Research, France tsuyoshi.sugibuchi@internetmemory.net

Abstract. In this paper, we propose a high level query language for expressing analytic queries over big data sets. A main contribution of our approach is the clear separation between the conceptual and the physical level. An analytic query and its answer are defined at the conceptual level independently of the nature and location of data. The abstract definitions are then mapped to lower level evaluation mechanisms, taking into account the nature and location of data, as well as other related aspects. Our overall objective is to have query formulation done on an abstract level, while actual query evaluation can adapt to the evaluation mechanisms offered in each case. To achieve this objective it is necessary to raise the level of abstraction, by providing a sound mathematical basis for the mapping of queries to lower level evaluation mechanisms. However, raising the level of abstraction offers three advantages: (a) useful insights into the process of data analytics in general, and MapReduce in particular (b) a formal approach to the rewriting of analytic queries and the generation of query execution plans and (c) the possibility of leveraging structure and semantics in data in order to improve performance. We emphasize that, although theoretical in nature, this work uses only basic and well known mathematical concepts (namely functions and set partitions).

1 Introduction

Data analysis is a well established research field with multiple applications in several domains. However, the methods and tools of data analysis evolve rapidly, and in significant ways, as the size of data accumulated by modern applications increases in unprecedented rates. The work reported in this paper contributes in this evolution by proposing a high level query language for big data analytics. In this section we first introduce the context of our work (i.e big data and big data analytics) and then we present briefly our motivations and contributions.

1.1 Big Data

Today, scientists regularly encounter limitations due to the very large sizes of data sets, in many areas, including meteorology, genomics, complex physics simulations, and biological and environmental research. The limitations also affect Internet search, finance and business informatics.

Examples of such large data sets include web logs, social network data, Internet text and documents, Internet search indexing, call detail records, medical records, photography archives, video archives, and large-scale e-commerce data. Striking examples from the business world include Facebook, which handles 40 billion photos from its user base; and Walmart, which handles more than 1 million customer transactions every hour, imported into databases estimated to contain more than 2.5 petabytes of data.

The term “big data” refers to data sets with sizes beyond the ability of commonly used software tools to capture, curate, manage, and process the data within a reasonable lapse of time [30]. As a consequence, what is considered big data varies depending on the capabilities of the organization managing the data set. However, though big data is a moving target, what is considered big data today is in the order of petabytes to exabytes[26].

It is worth noting here that size is not the only characteristic of big data. As stated in [7], big data are high-volume, high-velocity, and/or high-variety information assets that require new forms of processing to enable enhanced decision making, insight discovery and process optimization.

The potential uses of big data, but also the difficulties connected with their capture, curation, management and processing have been recognized at the highest administration levels [27][28].

However, the use of big data has drawn considerable criticism as well. Broader critiques have been leveled at the assertion that big data will spell the end of theory, focusing in particular on the notion that big data will always need to be contextualized in their social, economic and political contexts [3][22]. Another criticism comes from the fact that, even as companies invest huge amounts to derive insight from information streaming in from suppliers and customers, less than half of the employees have sufficiently mature processes and skills to do so.

Moreover, consumer privacy advocates are concerned about the threat to privacy represented by increasing storage and integration of personally identifiable information; and expert panels have released various policy recommendations to conform practice to expectations of privacy [38] (see also the articles in ”The Guardian” and ”Washington Post”, June 6, 2013, on the controversial PRISM project). All this criticism simply points to the fact that there is another side to big data regarding societal benefits and risks [34][31].

However, inspite the criticism that the use of big data has drawn recently, their collection and processing holds big promises for a variety of human activities. In fact, a new field centered on big data is emerging, usually referred to as “data science”, and several universities and educational institutions already offer degrees in this field [47]. [18].

1.2 Data Analytics

Big data is difficult to analyze and to work with using relational databases and desktop statistics and visualization packages, requiring instead massively parallel software running on tens, hundreds, or even thousands of servers [29][36][37].

The analysis of data, or data analytics, is the process of highlighting useful information drawn from big data sets, usually with the goal to support decision making. Data analysis has multiple facets and approaches, encompassing diverse techniques under a variety of names, in different business, science, and social science domains. It differs from data mining which is a particular data analysis technique that focuses on modeling and knowledge discovery for predictive rather than purely descriptive purposes.

Big data analytics demands real or near-real time information delivery, and latency is therefore avoided whenever and wherever possible. With this difficulty, a new platform of big data tools has arisen, such as in the Apache Hadoop Big Data Platform [13] derived from papers on Google's MapReduce and Google File System.

Actually, MapReduce [15] is emerging as a leading framework for performing scalable parallel analytics and data mining; and there is already an impressive body of literature on MapReduce, but also some controversy coming mainly from the database community [44][4][45].

The success of MapReduce is due to several reasons: it is offered as a free and open source implementation; it is easy to use [40]; it is widely used by Google, Yahoo! and Facebook; and it has been shown to deliver excellent performance on extreme scale benchmarks [14][51]. All these factors have resulted in the rapid adoption of MapReduce for many different kinds of data analysis and processing [10][33][49][39][11].

Originally, MapReduce was mainly used for Web indexing, text analytics, and graph data mining. Today, as MapReduce is becoming the *de facto* data analysis standard, it is also used for the analysis of structured data, an area traditionally dominated by relational databases in data warehouse deployments. Even though many argue that MapReduce is not optimal for analyzing structured data [44][40], it is nonetheless used increasingly frequently for that purpose because of a growing tendency to unify the data management platform.

In fact, there is already a significant body of literature on integrating MapReduce and relational database technology [48][2][12][53] [17][9], following one of two approaches: either adding MapReduce features to a parallel database system [20][12][52] or adding database technology to MapReduce[48][53] [2][17][9]. The second approach seems to be more promising as MapReduce is offered as a free and open source implementation while there exists no widely available open source parallel database system. The most notable representative of the second approach is HadoopDB [2], which is commercialized by Hadapt [24][5].

1.3 Motivation and Contributions

There are many systems developed today for the parallel processing of big data sets [21][48][35] [19][6][2]. All these systems co-exist, each carefully optimized in accordance with the final application goals and constraints. However, their evolution has resulted in an array of solutions catering to a wide range of diverse application environments. Unfortunately, this has also fragmented the big data solutions that are now adapted to particular types of applications.

At the same time, applications have moved towards leveraging multiple paradigms in conjunction, for instance combining real time data and historical data. This has led to a pressing need for solutions that seamlessly and transparently allow practitioners to mix different approaches that can function and provide answers as an all-in-one solution.

Based on this observation, the overall objective of our work is to separate clearly the conceptual and the physical level so that one can express analysis tasks as queries at the conceptual level *independently* of how their evaluation is done at the physical level. To achieve this objective we propose (a) a high level language in which we can formulate queries and study their properties at the conceptual level and (b) mappings to existing evaluation mechanisms (e.g. SQL engines, MapReduce) which perform the actual evaluation of queries. In other words, we propose a language which is agnostic of the application environment as well as of the nature and location of data.

In defining our language, the basic notion that we use is that of *attribute* of the data set. However, we view an attribute as a function from the data set to some domain of values. For example, if the data set D is a set of tweets, then the attribute "character count" (cc) is seen as a function $cc : D \rightarrow Integers$ such that, for each tweet i, $cc(i)$ is the number of characters in i.

A query in our language is defined to be a triple $Q = (g, m, op)$ such that g and m are attributes of the data set D, and op is an aggregate operation applicable on m-values. The evaluation of Q is done in three steps as follows: (a) group the items of the data set D using the values of g (i.e. items with the same g-value g_i are grouped together), (b) in each group of items thus created, extract from D the m-value of each item in the group, and (c) aggregate the m-values in each group to obtain a single value v_i. The aggregate value v_i is defined to be the answer of Q on g_i, that is $ans_Q(g_i) = v_i$. This means that a query is a triple of functions and its answer is also a function.

Conceptually, all one needs in order to perform the three-step query evaluation described above is the ability to extract attribute values from the data set. Now, the method of extraction depends on the nature and location of data. For example, if the data resides in relational tables then one can use SQL in order to extract attribute values, whereas if the data resides in a distributed file system then one needs specialized algorithms to do the extraction. We note in this respect that, while raw, in-situ data processing is traditionally perceived as a significant performance bottleneck, novel indexing and caching structures gradually speed up raw data access.

Anyhow, at the conceptual level, we are not interested in how attribute values are extracted from the data set. Rather, we are interested in using the definition of a query and its answer in order to define formal methods for query rewriting that will help improve performance of query evaluation at a lower level. Additionally, we are interested in the use of rewriting to leverage structure and semantics in data in order to improve performance.

As for user interaction, in our approach, the analyst interacts with the data set using a set of attributes of interest that we call an "analysis context". The

analyst uses the attributes of his context to write analytic queries in the form of triples, in the way described above. He can add or remove attributes to the context at will, so an analysis context can be seen as a "light-weight" schema which is agnostic of the nature and location of data. This is in sharp contrast to "heavy-weight" schemas such as relational schemas, which are aware of the structure of data into tables.

In previous work [8] a functional data model was presented as an alternative to the relational model. Although the scope of that work is different than ours, some of the functional operations used are similar to those that we use in the definition of our language.

In [42], a functional model was presented for data analysis in data warehouses over star schemas, using a definition of query similar to ours. We build on that work by enlarging the scope to big data environments, by introducing the concept of analysis context and by presenting a formal method for query rewriting and generating query execution plans. Moreover, we provide mappings from our model to existing evaluation mechanisms (SQL engines, MapReduce) where actual query evaluation takes place.

In other recent work [43], a language for data analysis was presented based entirely on partitions of the data set. Moreover, a notion of query rewriting was proposed based on the concept of quotient partition. However, no algorithms for query rewriting were presented and no definition of query execution plan was given.

The remaining of the paper is organized as follows. In section 2 we present the conceptual model. In section 2.1 we present the abstract definition of a query and its answer; in section 2.2 we introduce the concept of analysis context and its query language; in section 2.3 we present a formal approach to the rewriting of analytic queries; and in section 2.4 we use query rewriting to provide a formal method for generating query execution plans. Section 3 is devoted to mappings between the conceptual and the physical level. In section 3.1 we present a general, conceptual scheme for query evaluation based on our abstract definition of query. Then we present the mapping of our conceptual evaluation scheme to three important, practical application environments: MapReduce (section 3.2), Column Databases (section 3.3) and Row Databases (section 3.4). Section 4 contains some concluding remarks and outlines research perspectives.

We emphasize that, although theoretical in nature, our approach uses only basic and well known mathematical concepts, namely functions and set partitions.

2 The Formal Model

In the formal model that we present in this section, we consider a data set D, whose elements we call *data items*, and we make two assumptions:

- **Item identification**: We assume that D consists of data items that can be uniquely identified. For example, if D is a set of tweets then each tweet can

be identified by a time stamp or by a URI; and if D is the set of tuples in a relational table then each tuple can be identified by a tuple identifier or some key of the table.

- **Attributes as functions**: We assume that each attribute of D is a function associating each data item of D with a value, in some set of values. For example, if D is a set of tweets then the attribute *Date* is seen as a function associating each tweet in D with the date in which the tweet was sent. In the remaining of the paper we shall use the terms "function on D" and "attribute of D" interchangeably.

We note that the attributes-as-functions assumption is compatible with the notion of attribute in both column databases and row databases (i.e. relational databases). Indeed, a column database stores columns instead of rows of data (e.g. MonetDB). At the conceptual level, a column database can be seen as storing a set of functions of the form $f_A : ID_A \to A$, where A is an attribute and ID_A is a subset of the set ID of identifiers used by the column store. Similarly, in a table T of a row database, we can associate each attribute A of T with a function $f_A : TID_T \to A$, where TID_T is the set of tuple identifiers in T; this function is defined as follows: $f_A(t) = t(A)$, for each tuple identifier t in TID_T, where $t(A)$ denotes the value of t on attribute A. Note that if we assume TID_T to be an extra attribute of T, then the function f_A can be extracted from T by projection over TID_T and A; in other words, $f_A = proj_{TID_T,A}(T)$.

We emphasize that, apart from the two assumptions above, we make no other assumption whatsoever throughout the paper. In other words, the data set can be structured or unstructured, homogeneous or heterogeneous, centrally stored or distributed. Our results apply in all these cases. Moreover, as we shall see in Section 3, if the data is structured, then our approach can leverage structure and semantics to improve performance.

2.1 The definition of analytic query and its answer

As we have already mentioned in the introduction, a query in our language is defined to be a triple $Q = (g, m, op)$ such that g and m are attributes of the data set D, and op is an aggregate operation applicable on m-values. The attributes g and m are called the "grouping attribute" and the "measuring attribute", respectively. The evaluation of Q is done in three steps as follows:

- **Grouping**: Group the items of the data set D using the values of g (i.e. items with the same g-value g_i are grouped together)
- **Measuring**: In each group of items thus created, extract from D the m-value of each item in the group
- **Reduction**: Aggregate the m-values in each group to obtain a single value v_i

The aggregate value v_i is defined to be the answer of Q on g_i, that is $ans_Q(g_i) = v_i$.

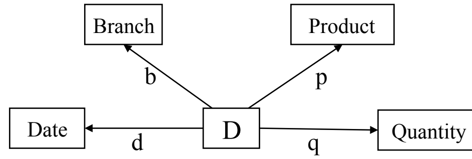

Fig. 1. Running example

Let us see an example in detail in order to motivate the definition of "query" in our approach. We shall use this example as our running example throughout the paper. Suppose D is the set of all delivery invoices over a year, in a distribution center (e.g. Walmart), which delivers products of various types in a number of branches. A delivery invoice has an identifier (e.g. an integer) and shows the date of delivery, the branch in which the delivery took place, the type of product delivered (e.g. $CocaLight$) and the quantity (i.e. the number of units delivered of that type of product). There is a separate invoice for each type of product delivered.

The data of all invoices during the year is stored in a database for analysis purposes. The stored data is shown schematically in Figure 1, where D represents the set of all delivery invoices and the arrows represent attributes of D. For each stored invoice, the function d returns the date in which the delivery took place; the function b returns the branch in which the delivery occurred; and the functions p and q return the type of product delivered and the quantity (i.e. the number of units) of that type of product.

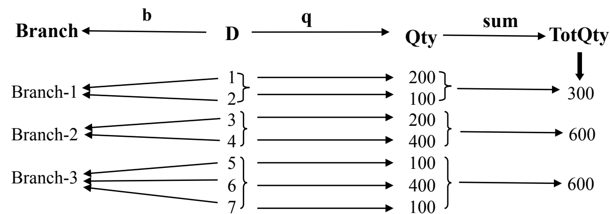

Fig. 2. Computing the total quantity delivered by branch

Suppose now that we want to know the total quantity delivered to each branch (during the year). This computation needs the extensions of the functions b and q. Figure 2 shows a toy example of the data returned by b and q, where the data set D consists of seven invoices, numbered 1 to 7. In order to find the total quantity by branch we proceed in three steps as follows:

Grouping: During this step we group together all invoices referring to the same branch (using the function b). We obtain the following groups of invoices (as shown in the figure):

- Branch-1: 1, 2
- Branch-2: 3, 4
- Branch-3: 5, 6, 7

Measuring: In each group of the previous step, we find the quantity corresponding to each invoice (using the function q):

- Branch-1: 200, 100
- Branch-2: 200, 400
- Branch-3: 100, 400, 100

Reduction: In each group of the previous step, we sum up the quantities found:

- Branch-1: 200+100= 300
- Branch-2: 200+400= 600
- Branch-3: 100+400+100= 600

Then the association of each branch to the corresponding total quantity is the desired result (called TotQty in the figure):

- Branch-1 $\rightarrow$ 300
- Branch-2 $\rightarrow$ 600
- Branch-3 $\rightarrow$ 600

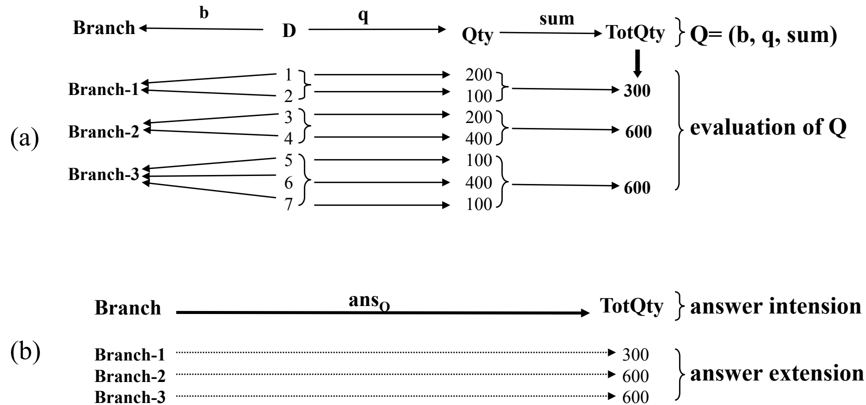

Fig. 3. An analytic query and its answer

We view the ordered triple $Q = (b, q, sum)$ as a query over D (see Figure 3 (a)), the function $ans_Q : Branch \rightarrow TotQty$ as the answer to Q (see Figure 3 (b)), and the computations in Figure 3 (a) as the query evaluation process. Note that what makes the association of branches to total quantities possible is the fact that b and q have a common source (which is D in this example).

The function b that appears first in the triple (b, q, sum) and is used in the grouping step is called the *grouping function*; the function q that appears second in the triple is called the *measuring function*, or the measure; and the function sum that appears third in the triple is called the *reduction operation* or the aggregate operation. Actually, a triple such as (b, q, sum) should be regarded as the specification of an analysis task to be carried out over the data set D.

Note that, as our example shows, the requirements for a triple such as (b, q, sum) to qualify as a query over D are the following:

– the grouping and measuring function must both be attributes of D (i.e. functions with D as their common source)
– the reduction operation must be an operation among those applicable over the target of the measuring function

Also note that, as the only requirement for b and q is that they must be attributes of D, each of them can play the role of either a grouping function or a measuring function. In other words, the triple $(q, b, count)$ is a valid query and asks for the number of branches by quantity delivered. To answer this query we use q to group together all invoices having the same quantity delivered (this is the grouping step); then we use b to find the branches that were delivered that quantity (this is the measuring step); and finally we count the branches in each group (this is the reduction step). For example, in Figure 2, if we consider the quantity 200, then we find that there are 2 branches that were delivered that quantity (namely Branch-1 and Branch-2).

To see another example of query, suppose that D is a set of tweets accumulated over a year; dd is the function associating each tweet t with the date $dd(t)$ in which the tweet was sent; and cc is the function associating each tweet t with its character count, $cc(t)$. If we want to know the average number of characters in a tweet by date, then we can follow the same steps as in the delivery invoices example: first we group the tweets by date (using dd); then we find the number of characters per tweet (using cc); and finally we take the average of the character counts in each group (using "average" as the reduction operation). The appropriate query formulation in this case is (dd, cc, avg).

We note that the grouping and reduction steps can be applied *independently* of the nature of data. Indeed, in the delivery invoices example, the data will most likely be a set of records in a relational table (therefore structured data), whereas in the tweets example the data is text of variable length (therefore unstructured data). Yet in both cases the definition of a query and its answer is done in the same way. However, in order to actually evaluate the answer, we need the extensions of the grouping and measuring attribute (as was the case for attributes b and q in Figure 2). These extensions will have to be extracted

from the underlying data set. This means that for each attribute (grouping or measuring attribute) we need an algorithm for extracting that attribute's value from each data item.

In view of the preceding discussion, we can now give the formal definition of a query and its answer in our model.

Definition 1 (Query). *Let $D = \{d_1, \ldots, d_n\}$ be a finite set. An analytic query (or simply query) over D is a triple $Q = (g, m, op)$ such that $g : D \to A$ and $m : D \to V$ are attributes of D, and op is an operation over V taking its values in a set W.*

The reduction operation *op* is actually a function that associates every finite tuple of elements from V with an element in a set W. In our running example, where the reduction operation is "sum", the set V is the set of integers (number of units delivered) and so is W (sums of quantities delivered); therefore in this example we have $V = W$. However, in general, V can be different than W. Indeed, in our tweets example, where V is the set of integers (character counts) and *op* is the operation "average", the set W is the set of real numbers (averages of character counts) and therefore $V \neq W$.

It is important to note that the reduction operation must be among those operations that are applicable on V. For example, if V is the set of integers (as in the tweets example), then the reduction operation can be any among "sum", "average", "median", "count", "max", "min", and so on.

In order to define formally the answer to a query over D, we need to define formally the steps of grouping and reduction.

Definition 2 (Grouping).
Let $D = \{d_1, \ldots, d_n\}$ be a finite set, let $g : D \to A$ be an attribute of D and let $\{a_1, \ldots, a_k\}$ be the values of g over D (clearly, $k \leq n$). We call grouping of D by g the partition induced by g on D.

We denote this partition by π_g, therefore we have:

$$\pi_g = \{g^{-1}(a_1), \ldots, g^{-1}(a_k)\}$$

Note: We recall that a partition of D is any collection of nonempty subsets of D (also called the "blocks" of the partition), with the following properties: (a) the subsets are pairwise disjoint and (b) their union is D.

For example, in Figure 3 (a), the grouping of D by b consists of three groups, one for each of the values of b:

- b^{-1}(Branch-1) $= \{1, 2\}$
- b^{-1}(Branch-2) $= \{3, 4\}$
- b^{-1}(Branch-3) $= \{5, 6, 7\}$

We now define formally the reduction operation.

Definition 3 (Reduction).
Let $D = \{d_1, \ldots, d_n\}$ *be a finite set, let* $m : D \to V$ *be an attribute of* D*, and let* op *be an operation over* V *with values in a set* W*. The reduction of* m *with respect to* op*, denoted* $red(m, op)$*, is a value of* W *defined as follows:*

$$red(m, op) = op(\langle m(d_1), \ldots, m(d_n)\rangle)$$

Note that in the above definition of reduction we use the notation $op(\langle m(d_1), \ldots, m(d_n)\rangle)$ to emphasize that all values of m must be taken into account, even if there are repeated values (i.e. even if $m(d_i) = m(d_j)$, for some $d_i \neq d_j$). For example, in Figure 3(a), although we have $q(5) = q(7) = 100$, the value 100 is taken twice into account when computing the sum of all values.

Here is an example of reduction of the function $q : D \to Quantity$ in Figure 3 (a): $red(q, sum) = 200 + 100 + 200 + 400 + 100 + 400 + 100 = 1500$. Other examples of reductions in that same figure are the following:

- $red(q/b^{-1}(\text{Branch-1}), sum) = 200 + 100 = 300$
- $red(q/b^{-1}(\text{Branch-2}), sum) = 200 + 400 = 600$
- $red(q/b^{-1}(\text{Branch-3}), sum) = 100 + 400 + 100 = 600$

Here $q/b^{-1}(\text{Branch-i})$ denotes the restriction of function q to the subset $b^{-1}(\text{Branch-i})$ of D, i= 1, 2, 3.

We can now define formally the notion of query answer.

Definition 4 (Query Answer). *Let* $Q = (g, m, op)$ *be a query over* D*, where* $D = \{d_1, \ldots, d_n\}$ *is a finite set,* $g : D \to A$ *and* $m : D \to V$ *are attributes of* D*, and* op *is an operation over* V *with values in a set* W*. Let* $\{a_1, \ldots, a_k\}$ *be the values of* g *over* D*. The answer to* Q*, denoted* ans_Q*, is a function from the set of values of* g *to* W *defined by:*

$$ans_Q(a_i) = red(m/g^{-1}(a_i), op), i = 1, 2, ..., k,$$

Figure 3(b) shows schematically the answer, ans_Q (which is a function); and Figure 4 shows the relationship between ans_Q and the functions appearing in the query Q. It is worth noting that a query is a triple of functions and the answer is also a function. Moreover, as we shall see later, the fact that $ans_Q(a_i)$ is given by a closed formula facilitates the proof of theorems when studying query rewriting.

It should be clear from the above definition of query answer that the task of evaluating Q can be easily parallelized. Indeed, if for each i we consider the evaluation of $ans_Q(a_i)$ as a sub-task then we can assign the sub-tasks to a number of processors, each processor receiving one or more sub-tasks. Each processor then executes its own sub-task(s) independently of all other processors, and the results from all processors, put together, constitute the answer to the query.

Note: While evaluating a sub-task, a processor may decide to further partition the sub-task block into smaller blocks, before performing reduction (and this can be done recursively). This is possible under the assumption that the reduction operation is “distributive”, a concept to be defined later on.

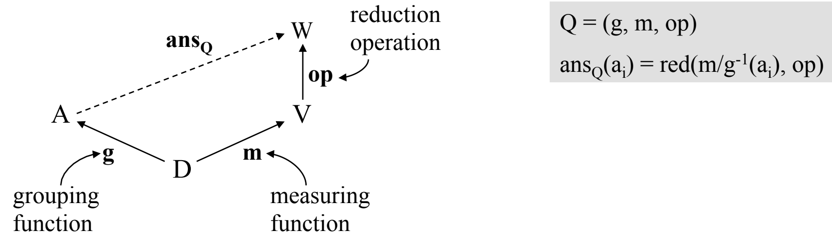

Fig. 4. A query Q and its answer, ans_Q

For example, in Figure 2, if the sub-tasks ans_Q(Branch-1), ans_Q(Branch-2) and ans_Q(Branch-3) are assigned to three different processors, then the first processor will produce as a result ans_Q(Branch-1) = 300, the second ans_Q(Branch-2) = 600, and the third ans_Q(Branch-3) = 600; and all three results, put together, constitute the answer to the query.

A query $Q = (g, m, op)$ over D can be enriched by introducing functional restriction at either of two levels: at the level of attributes or at the level of the query answer (recall that the query answer is itself a function). This is stated formally in the following definition.

Definition 5 (Restricted Query).
Let $Q = (g, m, op)$ be a query over D as defined earlier. Then the following are also queries over D:

- ***Attribute-Restricted Query:*** $(g/E, m, op)$*, where E is any subset of D (i.e. of the domain of definition of g).*
- ***Result-Restricted Query:*** $(g, m, op)/F$*, where F is any subset of the target of g (i.e. of the domain of definition of ans_Q).*

In our discussions we shall use the term "restricted query" to mean a query which is attribute-restricted and/or result-restricted. Conceptually, the evaluation of a restricted query is straightforward:

Attribute-Restricted Query: to evaluate the query $(g/E, m, op)$, compute the restriction g/E and then evaluate the query $(g/E, m, op)$ over E

Result-Restricted Query: to evaluate the query $(g, m, op)/F$, evaluate the query $Q = (g, m, op)$ over D, to obtain its answer (i.e. the function ans_Q), and then compute the restriction ans_Q/F

Attribute- and Result-Restricted Query: to evaluate the query $(g/E, m, op)/F$ evaluate the query $Q' = (g/E, m, op)$ over E and then compute the restriction $ans_{Q'}/F$

As an example of attribute-restricted query, refer to Figure 3(a) and suppose that we want the totals by branch for the subset $E = \{3, 4, 5, 6\}$ of D. Formally,

this query (call it Q_1) is written as $Q_1 = (b/E, q, sum)$, and its answer is obtained by first computing the restriction b/E and then evaluating Q_1 over E. We find the following answer:

- ans_{Q_1}(Branch-2) $= 600$ (because $b(3) = b(4) =$Branch-2, $q(3) = 200$ and $q(4) = 400$)
- ans_{Q_1}(Branch-3) $= 500$ (because $q(5) = q(6) =$Branch-3, $q(5) = 100$ and $6(9) = 400$)

Note that the grouping based on b/E creates a group for Branch-3 which is different than that obtained when grouping is based on b; whereas the group for Branch-2 is the same when the grouping is based either on g/E or on g (as invoices 3 and 4 are present both in D and in E). Also note that Branch-1 does not appear among the values of q/E, as no invoice in E is associated with Branch-1.

Now, as an example of result-restricted query, assume we want the totals by branch, but only for branches Branch-1 and Branch-2. Formally, this query (call it Q_2) is written as $Q_2 = (b, q, sum)/F$, where $F = \{$Branch-1, Branch-2$\}$. Its answer is obtained by first evaluating the query $Q = (b, q, sum)$ over D (as shown in Figure 3) and then restricting its answer, ans_Q, to the subset F of its domain of definition. We find the following answer:

- ans_{Q_2}(Branch-1) $= 300$
- ans_{Q_1}(Branch-2) $= 600$

In other words, we keep from ans_Q only its values on Branch-1 and Branch-2 and their corresponding totals.

Note that, in practice, it is often the case that some query Q is used as a basis for defining several restricted queries. In such cases, the definition of Q is stored in a cache together with its answer at some point in time, in order to accelerate the evaluation of all restricted queries using Q as a basis. Such a stored query is an example of what is usually referred to as a *materialized view*. One problem with materialized views is their maintenance, as the stored answer might need to be changed when the data set D changes (see for example [23]).

One issue regarding restricted queries is how to define the sets E and F that appear in their definitions. More generally, given a function $h : X \rightarrow Y$, the issue is how to define a subset E of X to which we want the function h to be restricted. The obvious way is to enumerate the elements of E as we did in our examples above (assuming E is finite). There is however another way to define E which is in fact generally used in data management. It consists in (a) considering a second function, say $r : X \rightarrow Z$, with the same domain of definition as h, (b) specifying a subset W of Z and (c) defining E to be equal to the inverse image of W under r, that is $E = r^{-1}(W)$.

As an example, referring to Figure 1, we can define a subset E of D by giving a set W of products of interest, and defining E to be the set of all invoices in D that correspond to one of the given products; formally, $E = p^{-1}(W)$. This is

precisely what is done when querying a relational table T by issuing a statement such as:

```
Select * From T Where Product=P1 or Product=P2.
```

Indeed, this statement returns all tuples whose value on attribute Product is either $P1$ or $P2$.

Note that we can even use the function h itself for defining a subset E of its domain of definition. For example, consider the function b of Figure 2 and let $W = \{\text{Branch-1,Branch-2}\}$. Then we can define $E = b^{-1}(W) = \{1, 2, 3, 4\}$.

We end this section with two important remarks regarding the definition of a query and its answer. First, the mathematical concepts used are actually elementary: (a) the concept of function, (b) the inverse of a function and the partition that this inverse induces on its domain of definition, and (c) the restriction of a function to a subset of its domain of definition. As we have seen in this section these concepts are sufficient in order to define an analytic query and its answer at the conceptual level; and as we shall see shortly, these same concepts are sufficient in order to define formally query rewriting.

Our second remark concerns the fact that, in a query $Q = (g, m, op)$, the functions g and m might not be defined on every item of D. Therefore grouping and reduction as described earlier can be performed only on the set of items on which g and m are both defined. However, in order to simplify the presentation, and without loss of generality, we shall assume that g and m are defined on all items of D. In other words, D actually represents the common domain of definition of g and m, defined as $D = def(g) \cap def(m)$, where $def(g)$ and $def(m)$ denote the domains of definition of g and m, respectively.

2.2 Analysis Context

Analysts are usually interested in analyzing a data set in many ways, using a number of different attributes in their analytic queries. For example, in Figure 1, one can define analytic queries using any of the attributes b, p and q. These are "factual", or direct attributes of D as their values appear on the delivery invoices.

However, apart from these factual or direct attributes, analysts might be interested in attributes that are not direct but can be "derived" from the direct attributes. Figure 5(a) shows several derived attributes. For instance, the attributes m and y are derived attributes as their values can be computed from those of attribute d (e.g. from the date 05/06/1986 one can derive the month 06/1986 and the year 1986). Similarly, the attribute r can be derived from geographical information on the locations of the branches; and the attributes s and c might be possible to derive from data carried by RFID tags embedded in the products themselves.

Roughly speaking, the set of attributes of interest to a group of analysts is what we call an analysis context (or simply a context); and these attributes can be direct or derived attributes of the data set. Hence the following definition.

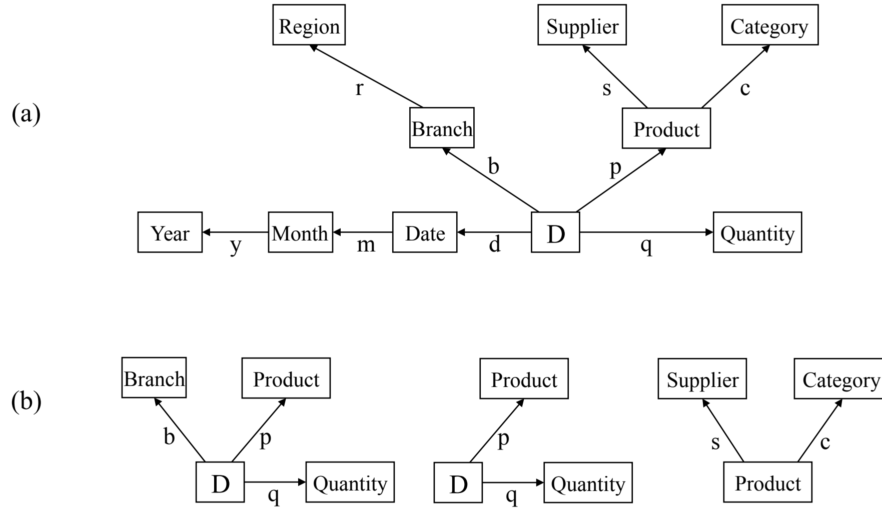

Fig. 5. Examples of contexts

Definition 6 (Analysis Context). *Let D be a data set and let $\mathcal{A}$ be the set of all attributes of D (direct or derived). An analysis context over D (or simply "context") is any set of attributes from $\mathcal{A}$.*

Figure 5(a) shows a context in our running example, containing direct and derived attributes. Figure 5(b) shows three other contexts, one of which is not rooted in D (it is rooted in $Product$).

Actually, an analysis context is the interface between the analyst and the data set, therefore it can be seen as a schema. However, in contrast to, say, a relational schema, an analysis context is not aware of structure in data. Moreover, a context is not fixed: analysts can remove attributes from their context at will, or add new attributes if necessary for the analysis process. In other words, a context is a kind of "light weight", dynamic schema reflecting the analyst's needs.

As a context customizes the needs of a group of analysts, a parallel can be made with the notion of view in relational databases or the notion of data mart in data warehouses.

Note that a context is a directed labelled graph whose nodes represent data sets, and whose edges are functions (i.e. attributes) between these data sets. We note here that the nodes represent sets of values that are independent from each other (in much the same way as attribute domains in the relational model are assumed to be sets of independent values).

Seen as syntactic objects, the edges of a context are triples of the form (source, label, target), therefore two edges are different if they differ in at least one component of this triple. This implies that two edges can have the same label if they have different source and/or different target. Moreover, two different edges

can have the same source and the same target as long as they have different labels (we call such edges "parallel edges").

Most importantly, a context is always an acyclic graph, in the sense described by the following proposition.

Proposition 1 (Context Acyclicity).
Let $\mathcal{C}$ be a context over D. Then for every node A of $\mathcal{C}$ the only possible cycle on A is ι_A, that is the identity function on A.

Proof. Let A and B be two nodes of $\mathcal{C}$, and suppose there is a cycle on A consisting of two functions: $f : A \rightarrow B$ and $g : B \rightarrow A$. Suppose that there is some element a in A such that $g \circ f(a) = a'$ and $a \neq a'$. This implies that a' depends on a, a contradiction to our assumption that the nodes of $\mathcal{C}$ represent sets of independent values. Therefore the only possibility to have a cycle on A is when $g \circ f(a) = a$ for all a in A; in other words the only possible cycle on A is ι_A, where ι_A is the identity function on A.

A typical example where identity cycles occur is when prices of products are given in two or more different currencies, such as the price in dollars and the price in euros of the same product: *Price-in-Dollars* $\rightarrow$ *Price-in-Euros* and *Price-in-Euros* $\rightarrow$ *Price-in-Dollars*. In such cases the two nodes are equivalent, in the sense that there is one-to-one correspondence between their values.

We note that, although acyclic, a context is not necessarily a tree. It can have one or more roots and it can also have parallel edges and parallel paths ("parallel" in the sense "same source and same target").

In general, the users of a context have two main ways for expressing queries. First, they can express queries on any node of the context - not just on D. For example, in the context of Figure 5(a), suppose we add an attribute $u : Product \rightarrow UnitPrice$, giving the unit price for each product. Then one can formulate the following query on $Product$: (c, u, max), asking for the maximum unit price by product category.

Second, users of a context can combine its attributes to form complex grouping functions. For example, in the context of Figure 5(a) one can ask for the total quantities by region, using as grouping function the composition of the attributes b and r: $(r \circ b, q, sum)$.

In our model, we can form complex grouping functions using the following four operations on functions: composition, pairing, restriction and Cartesian product projection. These operations form the so called *functional algebra* (see for example [42]). We note that the operations of the functional algebra are well known, elementary operations except probably for pairing, which is defined as follows.

Definition 7 (Pairing). *Let $f : X \rightarrow Y$ and $g : X \rightarrow Z$ be two functions with common domain X. The pairing of f and g, denoted by $f \times g$ is a function from X to $Y \times Z$ defined as follows: $f \times g(x) = (f(x), g(x))$, for all x in X*

The above definition of pairing can be extended to more than two functions in the obvious way. Roughly speaking, pairing works as a tuple constructor.

Indeed, if we view the elements of X as identifiers, then for each x in X the pairing constructs a tuple of the images of x under its input functions; and this tuple is identified by x.

To see an example of using pairing, refer to Figure 5(a) and consider the following query: $(b \times p, q, sum)$. This query asks for the total quantities delivered by branch *and* product (i.e. its answer associates every pair $(branch, product)$ with a total quantity).

Note that the order of the images in the result of pairing is immaterial, as long as each image is prefixed by the function that produced it. In other words, pairing can be actually defined as follows: $f \times g(x) = \{f : f(x), g : g(x)\}$, for all $x \in X$. This definition implies that pairing is a commutative operation. On the other hand, when pairing two or more other pairings we obtain nested sets. If we agree to "flatten" the results, then pairing becomes an associative operation as well, and we can parenthesize at will, or even omit inner parentheses altogether, without ambiguity.

Using the operations of the functional algebra we can form not only complex grouping functions but also complex conditions when defining restrictions. For example, in Figure 5(a), we can ask for the total quantities by region and supplier, only for the month of January, using the following query: $(((r \circ b) \times (s \circ p))/E, q, sum)$, where $E = \{x|x \in D \wedge m \circ d(x) = January\}$

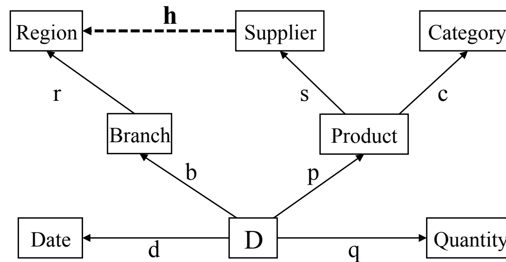

Fig. 6. A context with parallel paths

As another example, refer to Figure 6, showing a context with two parallel paths from D to *Region*. In this context, for each supplier, the attribute h gives the region where the suppliers' headquarters is located. Suppose now that we want the total quantities by Category, only for those invoices in D for which the branch is located in the same region as the headquarters of the product supplier. This is expressed by the following query: $((h \circ c)/E, q, sum)$, where $E = \{x|x \in D \wedge (h \circ s \circ p)(x) = (r \circ b)(x)\}$.

Note that in the above restricted query we use equality of two functional expressions in order to define attribute restriction. As we shall see in the section on rewriting (Section 2.3), equality of two functional expressions can be also

used as an integrity constraint on the context itself, and as such it can be used in query rewriting.

In general, a query over a context is a usual query (as defined in the previous section) in which we can use functional expressions instead of just functions. More formally, a functional expression over a context is defined as follows.

Definition 8 (Functional Expression). *A functional expression over a context $\mathcal{C}$ is either an edge of $\mathcal{C}$ or a well formed expression whose operands are edges and whose operations are those of the functional algebra.*

We note that if we evaluate a functional expression we obtain again a function. Therefore every functional expression e can be associated with a source and a target, defined recursively based on the notions of source and target of the edges in $\mathcal{C}$. For example, if $e_1 = r \circ b$ then $source(e_1) = D$ and $target(e_1) = Region$; similarly, if $e_2 = (r \circ b) \times p$ then $source(e_2) = D$ and $target(e_2) = Region \times Product$.

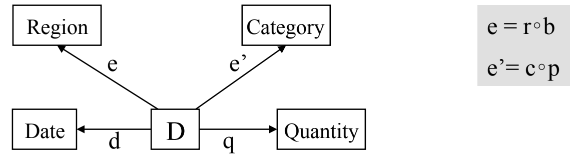

Fig. 7. A context with "complex" attributes (and their definitions)

Functional expressions should be regarded as complex attributes that are derived from other attributes using operations of the functional algebra. Therefore they can be used in defining contexts. For example, the context of Figure 7 uses two direct attributes (d and q) and two complex attributes (e and e'), whose definitions are given in the figure. Queries over such extended contexts can be defined as usual. For example, in the context of Figure 7, the following query asks for the total quantities by region and category: $(e \times e', q, sum)$. Actually, the labels e and e' can be seen as macros facilitating the reference to possibly complex expressions. They work in much the same way as view names in relational databases.

We are now ready to give the definition of the query language of a context.

Definition 9 (The Query Language of a Context). *Let $\mathcal{C}$ be a context. A query over $\mathcal{C}$ is a triple (e, e', op) such that e and e' have a common source (that can be any node of $\mathcal{C}$) and op is an operation over the target of e'. Restricted queries are defined in the same way as we have seen in the previous section. The set of all queries over $\mathcal{C}$ is called the query language of $\mathcal{C}$.*

Now, as stated in Proposition 1 above, a context is an acyclic graph in which the only possible cycle on a node A is ι_A, the identity function on A. In view

of this proposition, we shall assume that every node A of a context is endowed with the identity function ι_A. Moreover, we shall assume that every context $\mathcal{C}$ is endowed with an extra node denoted by K such that: (a) K denotes a singleton set $\{All\}$ and (b) for every node A of $\mathcal{C}$ there is an edge from A to K denoted by κ_A; that is $\kappa_A : A \to K$.

From a strictly technical point of view, the introduction of the functions ι_A and κ_A is justified as follows. The inverses of all functions on A induce the set of all partitions of A. This set is partially ordered by: $\pi \leq \pi'$ if each block of π is a subset of a block of π'. Under this ordering the set of partitions becomes a complete lattice with least, or bottom element the fine partition (i.e. the partition $\{\{a\}/a \in A\}$); and with largest, or top element the coarse partition (i.e. the partition $\{\{A\}\}$). Now, the fine partition of A is induced by any injective function on A, and in particular by ι_A, the identity function on A; and the coarse partition of A is induced by any constant function on A, and in particular by κ_A [3].

Clearly, given a context $\mathcal{C}$, we can use ι_A and κ_A in the same way as any other edge of $\mathcal{C}$. In particular, we can use them in queries or in functional expressions. For example, referring to Figure 5(a), consider the following queries using ι_A:

$$Q_1 = (\iota_D, q, sum) \text{ and } Q_2 = (q, \iota_D, count)$$

During the evaluation of Q_1, in the grouping step, the function ι_D puts each element of D in a single block. Therefore summing up the values of q in each block we simply find the value of q on the single element of D in that block; then the measuring step simply returns this value of q. It follows that $ans_{Q_1} = q$.

As for the query Q_2, the grouping function q groups together all invoices having the same delivered quantity; and as ι_D doesn't change the values in each block, the answer to Q_2 is the number of invoices by quantity delivered.

The function ι_A is typically used for finding the cardinality of A, using the following query: $(\kappa_A, \iota_A, count)$. The constant function κ_A, on the other hand, is typically used for finding the reduction of the whole of A under some measuring function.

Consider for example the following query: $Q_3 = (\kappa_D, q, sum)$. During the evaluation of Q_3, in the grouping step, the function κ_D puts all elements of D in a single block. Therefore by summing up the values of q in that block we find the total of all quantities delivered (i.e. for all dates, branches and products).

Regarding the use of ι_A and κ_A in functional expressions, we note the following facts: for any nodes A and B, and any functional expression $e : A \to B$, we have:

$$e \circ \iota_B = \iota_A \circ e = e$$

$$e \circ \kappa_B = \kappa_A$$

[3] The reason why we denote the unique value of κ by $\mathcal{A}ll$ is in order to hint to the fact that the function κ_A puts *all* the elements of A in a single block of the partition it induces on A

One important aspect regarding the evaluation of analytic queries in a context is visualization of the results. In our model, as the answer to a query is a function, its visualization can be done in any of the known ways for representing a (finite) function.

For example, consider the query $Q = (b \times (s \circ p), q, sum)$ asking for the totals by branch and supplier. The answer to this query is the following function:

$$ans_Q : Branch \times Supplier \rightarrow TotQty$$

One way to represent its extension is by a binary table $< y, TotQty(y) >$, in which each pair $y = (b_i, s_j)$ of a branch b_i and a supplier s_j, corresponds to a total quantity $TotQty(y)$ (this table can also be seen as a ternary table with *Branch*, *Supplier* and *TotQty* as its columns). This is the standard way for representing a function by its graph.

However, a different (but equivalent) representation is possible, based on the following observation: any triple of values (x, y, z) is equivalent to the triple (x, (y, z)), in the sense that they both carry the same information but encoded in different ways. Following this observation, the above answer can be also represented as follows: if we fix a value b_i of b then for each value s_j of s there corresponds a value of $TotQty$. In other words, each branch b_i is associated with a function $f_i : Supplier \rightarrow TotQty$. Therefore, for each branch b_i, one can "visualize" the corresponding function f_i using some visualization template (e.g. a histogram, a pie or any other template). Similarly, if we fix a value s_i of s then for each value b_j of b there corresponds a value of $TotQty$. In other words, each supplier s_i is associated with a function $h_i : Branch \rightarrow TotQty$, providing a different visualization of the answer.

The existence of two or more representations of the answer, combined with various visualization templates, is highly valuable in data analysis. Indeed, one might envisage a user friendly interface allowing the user to formulate an analytic query, and then explore its answer by switching from one representation to another, while selecting appropriate templates for visualizing each representation. In doing so, the user can explore the answer from different angles, thus getting better insight into the answer, and "discovering" patterns of information that might miss in a single representation.

In general, when the target of the grouping function is a Cartesian product, there is a formal method for generating all possible representations of the answer using *Currification* [46]. However, a detailed treatment of visualization and exploration of query results lies outside the scope of the present paper.

As a final remark, the fact that a context is an acyclic graph implies that it might have one or more roots. The existence of a single root means that data analysis concerns a single data set, such as the set D of our running example. The existence of two or more roots means that data analysis concerns two or more data sets possibly sharing attributes (and possibly being of different nature and structure). In such cases, one might want to combine information coming from queries over the two or more data sets to obtain further insights into the data.

2.3 Query Rewriting

In the previous section, we presented the definition of our query language over an analysis context. In section 3, we shall see how queries in our model can be evaluated by providing mappings to existing evaluation mechanisms. However, no matter how a query is evaluated, an orthogonal issue is the following: how can we *rewrite* a given query, at the conceptual level, in terms of one or more other queries. In this section we present the basic rewriting rules of our model. First, we note that query rewriting has two major applications:

- *Optimizing the evaluation of a query*: This is done by rewriting an incoming query in terms of the results of queries which have already been evaluated and their results stored (for example in a cache). The stored queries and their results are usually referred to as "materialized views". In query optimization, finding a rewriting of a query using a set of materialized views can yield a more efficient query execution plan.This problem also arises in data integration and data warehousing systems, where data sources can be described as precomputed views over a mediated schema.
- *Optimizing the evaluation of a set $\mathcal{Q}$ of queries*: In this approach, the queries of $\mathcal{Q}$ are arranged in a graph, in which there is an edge from query Q to query Q' if Q' can be rewritten in terms of Q. The problem is then to find an optimal execution plan for the whole set $\mathcal{Q}$, (using eventually materialized views if such views are available).

Query rewriting has been studied extensively in the 1990s (see [25] for a survey), and it is still an active topic of research in areas such as the semantic web [50]. Basically, as far as we are concerned in this paper, there are three distinct cases of rewriting a set $\mathcal{Q}$ of queries:

$\mathcal{Q} = \{(g, m, op_1), ..., (g, m, op_n)\}$: Here, the set $\mathcal{Q}$ contains n queries, all having the same grouping function and the same measuring function but possibly different reduction operations. In this case we can rewrite $\mathcal{Q}$ as follows: $\mathcal{Q} = ((g, m), \{op_1, ..., op_n\})$, meaning that grouping and measuring is done only once and the n reduction operations are applied to the result of measuring. In other words, grouping and measuring are "factored out".

$\mathcal{Q} = \{(g, m_1, op_1), ..., (g, m_n, op_n)\}$: Here, the set $\mathcal{Q}$ contains n queries, all having the same grouping function but possibly different measuring functions and reduction operations. In this case we can rewrite $\mathcal{Q}$ as follows: $\mathcal{Q} = (g, \{(m_1, op_1), ..., (m_n, op_n)\})$, meaning that grouping is done only once whereas the n measuring and reduction steps are applied to the result of grouping. In other words, grouping is "factored out".

$\mathcal{Q} = \{(g_1, m, op), ..., (g_n, m, op)\}$: Here, the set $\mathcal{Q}$ contains n queries, having the same measuring function and the same reduction operation but possibly different grouping functions. There is no obvious rewriting of the set $\mathcal{Q}$ this time, and this is precisely the problem that we tackle in this section.

Our approach to query rewriting is based on the form that a functional expression can have when used as a grouping function. To see intuitively how our approach works, consider the following queries on the context of Figure 5(a):

$Q = (p, q, sum)$, asking for the totals by product
$Q' = (c \circ p, q, sum)$, asking for the totals by product category

Clearly, the query Q' can be answered directly, following the abstract definition of answer (i.e. by grouping, measuring and reduction). However, Q' can also be answered indirectly, if we know (a) the totals by product and (b) which products are in which category. Then all we have to do is to sum up the totals by product in each category to find the totals by category. Now, the totals by product are given by the answer to Q, and the association of products with categories is given by the function c. Therefore the query Q' can be answered by the following query Q'', which uses the answer to Q as its measure: $Q'' = (c, ans_Q, sum)$, asking for the sum of answers to Q by product category. Note that the query Q'' is well formed as c and ans_Q have *Product* as their (common) source.

This observation leads to our basic rewriting rule, stated formally in Proposition 2 below. However, in order to state this proposition, we need the following definition of "distributive operation".

Definition 10 (Distributive Operation).
Let X be a finite set, let $m : X \to V$ be an attribute of X, and let op be an operation over V with values in a set W. Then op is called distributive if for every partition $\pi = \{X_1, \ldots, X_r\}$ of X the following holds:

– $red(m, op) = op(red(m/X_1, op) \ldots, red(m/X_r, op))$

Many common operations (such as sum, max, min etc.) are distributive but some common operations, such as "average" are not, as the following example shows: $avg(1, 2, 3, 4, 5) \neq avg(avg(1, 2), avg(3, 4, 5))$. Although there are "corrective" algorithms allowing the use of many non-distributive operations (average, median, etc.), we shall not pursue this subject any further. Rather, in order to simplify the discussion, we shall tacitly assume that all reduction operations are distributive.

Proposition 2 (Rewriting Compositions). *Let $\mathcal{C}$ be a context; let $f : A \to B$ and $g : B \to C$ be two (composable) edges of $\mathcal{C}$. Let $m : A \to V$ be an edge of $\mathcal{C}$ and let op be a distributive operation on V (with values in V). Let $Q = (f, m, op)$, $Q' = (g \circ f, m, op)$, $Q'' = (g, ans_Q, op)$ be three queries on $\mathcal{C}$. Then we have: $ans_{Q'} = ans_{Q''}$*

Proof. Observe first that, as ans_Q is a function with source B, the query Q'' is well formed, that is, its grouping function g and its measuring function ans_Q have the same source (namely B), and *op* is an operation on the target of ans_Q.

Let $c \in C$. It follows from well known properties of functions that:

$$\text{if } g^{-1}(c) = \{b_1, \ldots, b_k\} \text{ then } (g \circ f)^{-1}(c) = f^{-1}(b_1) \bigcup \ldots \bigcup f^{-1}(b_k)$$

From our definition of answer, we have:

$$ans_{Q'}(c) = red(m/(g \circ f)^{-1}(c), op)$$

As op is a distributive operation and the family $\{f^{-1}(b_1), \ldots, f^{-1}(b_k)\}$ is a partition of $(g \circ f)^{-1}(c)$, we have:

$$\begin{aligned} ans_{Q'}(c) &= red(m/(g \circ f)^{-1}(c), op) \\ &= op(red(m/f^{-1}(b_1), op), \ldots, red(m/f^{-1}(b_k), op)) \\ &= op(ans_Q(b_1), \ldots, ans_Q(b_k)) \\ &= red(m/g^{-1}(c), op) \\ &= ans_{Q''}(c) \end{aligned}$$

Therefore $ans_{Q'}(c) = ans_{Q''}(c)$ for all c in C and this concludes the proof.

In the above proposition, it is important to note that Q is a query on A, whereas Q'' (which is used for the rewriting of Q) is a query on B. This fact points to an important side effect, namely the possibility of avoiding join computations through rewriting.

Indeed, suppose that the extensions of f and g reside in different files, say F and G respectively. If the query Q' is evaluated directly (i.e. without rewriting) then a join of F and G followed by a projection is necessary in order to compute the composition $g \circ f$ (which serves as the grouping function for Q'). On the other hand, if Q' is evaluated indirectly (i.e. using rewriting), then this can be done *without* joining F and G as follows: first evaluate the query Q on A (in the file F), and then the query Q'' on B (in the file G) to obtain the answer to Q' (since $ans_{Q'} = ans_{Q''}$).

Therefore, when using rewriting, the extension of f (the grouping function of Q) is extracted from F, and the extension of g (the grouping function of Q'') is extracted from G, and no join is needed. In other words, the grouping functions f and g are each extracted from the file in which its extension resides and no join is needed.

In view of the previous proposition, we shall adopt the following notation for rewritings:

The Basic Rewriting Rule : $(g \circ f, m, op) = (g, (f, m, op), op)$

We shall refer to the above notation as *a rewriting of* $(g \circ f, m, op)$ based on g. The meaning of this rewriting is as follows: to obtain the answer of the query on the left, first replace the "nested query" $Q = (f, m, op)$ on the right by its answer, ans_Q, and then evaluate the resulting query (g, ans_Q, op). The basic rewriting rule will be used in the next section for generating efficient query execution plans.

Now, using the basic rewriting rule we can derive a rewriting rule for pairings. To this end, we need the following proposition which ties together composition, pairing and projection of Cartesian product of sets. Its proof is an immediate consequence of the definitions (and it is actually a rephrasing of the mathematical definition of Cartesian product of sets [41]).

Proposition 3 (Decomposing a pairing). *Let $f : X \to Y$ and $g : X \to Z$ be two functions with common domain X. Then the following hold:*

- $f = proj_Y \circ (f \times g)$ *and* $g = proj_Z \circ (f \times g)$

In other words, each of the factors of $f \times g$ can be reconstructed from $f \times g$ by composition with the corresponding projection function. This leads naturally to the following rewriting rule for pairings (which is a direct consequence of our basic rewriting rule and the above proposition).

Proposition 4 (Rewriting with Pairings).

Let $\mathcal{C}$ be a context; let $f : A \to B$, $g : A \to C$ be two edges of $\mathcal{C}$ with common source. Then we have:

- $(f, m, op) = (proj_B, (f \times g, m, op), op)$
- $(g, m, op) = (proj_C, (f \times g, m, op), op)$

To see how this rewriting rule for pairings works, refer to Figure 5(a) and suppose that the query $Q = (b \times p, q, sum)$ has been evaluated and its result stored (e.g. in a cache). Then we can compute the totals by branch and the totals by product from the result of Q, using the following rewritings:

$(b, q, sum) = (proj_{Branch}, (b \times p, q, sum), sum)$
$(p, q, sum) = (proj_{Product}, (b \times p, q, sum), sum)$

Clearly, the more the rewritings the better the chances for improving performance during the evaluation of a set of queries. One way to increase the number of possible rewritings among queries is to use properties of functional operations. A prime example in case is distributivity of composition over pairing as stated in the following proposition. Its proof is an immediate consequence of the definitions.

Proposition 5 (Composition Distributes over Pairing).

Let $\mathcal{C}$ be a context; let $f : A \to B$, $g : B \to C$ and $h : B \to D$ be three edges of $\mathcal{C}$. Then we have: $(g \times h) \circ f = (g \circ f) \times (h \circ f)$

This proposition can be used to derive additional rewriting rules. For example, refer to Figure 5(a) and consider the following rewritings:

$((s \times c) \circ p, q, sum) = ((s \times c), (p, q, sum), sum)$
$(s \circ p, q, sum) = (proj_{Supplier}, ((s \times c) \circ p, q, sum), sum)$
$(c \circ p, q, sum) = (proj_{Category}, ((s \times c) \circ p, q, sum), sum)$

The first rewriting allows to compute the totals by supplier and category from the totals by product; the second rewriting allows to compute the totals by supplier from the totals by supplier and category; and the third rewriting allows to compute the totals by category from the totals by supplier and category.

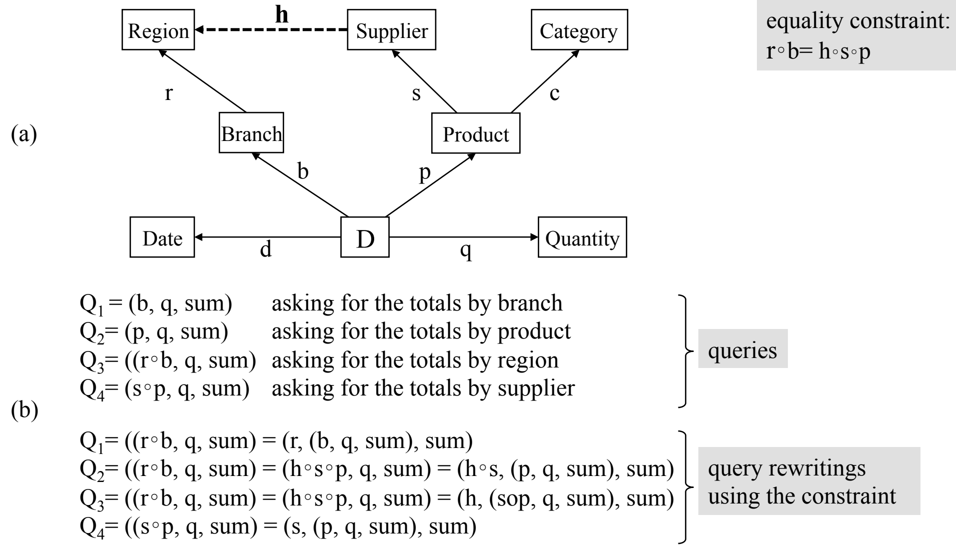

Fig. 8. A context with equality constraint and examples of query rewritings using the constraint

What makes it possible to have the second and third rewriting is the fact that composition distributes over pairing (Proposition 5).

Another way to increase the possibilities of rewritings among queries is to make use of equality constraints over the context (if any). For example, in the previous section, we mentioned the following equality constraint (also shown in Figure 8 (a)): $r \circ b = h \circ c \circ p$, meaning that the invoices in the data set D concern branches that are located in the same region as the headquarters of their suppliers. Figure 8(b) shows examples of how we can use this constraint in query rewriting. Note that, in the absence of the constraint, we can't have the second and third rewriting shown in the figure.

We note here that in traditional database systems, constraints are typically expressed as part of a stable schema, and involve the structure of data. If the data is unstructured then there is no stable schema and equality constraints seem to provide a natural alternative. Indeed, equality constraints express the semantics of data based on attributes without reference to structure. However, a detailed study of constraints lies outside the scope of the present paper.

In the next section we use query rewriting as the main tool for defining and generating query execution plans for a set of queries.

2.4 Query Execution Plans

A query execution plan is an ordered set of steps used to access data. Our goal in this section is to define formally the concept of query execution plan, as well

as its graphical representation using the concepts introduced so far. We describe how one can generate execution plans but do not address the issue of generating optimal query execution plans.

The formal definition of a query execution plan in our model relies essentially on two concepts, namely that of rewriting graph and that of query execution graph that we define next.

As we have seen earlier, at the conceptual level, a query in our model can be evaluated either directly, based on the definition of its answer, or after rewriting. The goal of rewriting an incoming query Q' in terms of an already evaluated query Q is to reuse the result of Q in evaluating Q' (assuming that the result of Q has been stored either in temporary memory or in a cache).

In this section we consider a more general problem that can be stated roughly as follows: given a set of queries that have to be evaluated in a given analysis context, define an ordered set of steps such that (a) each query is evaluated once (and only once) and (b) the evaluation order implied by rewriting is maintained. Our approach is based on the following two observations:

Sharing Two or more queries might have rewritings in terms of the same query, and therefore they can *share* its (stored) result.

Choice A given query might have two or more different rewritings in terms of other queries in the set, therefore a *choice* is necessary (according to some criterion).

Let us see an example illustrating the above observations. Consider again the context of Figure 8(a) with its equality constraint: $r \circ b = h \circ s \circ p$, where the function h gives the region in which the headquarters of each supplier is located.

We recall that an equality between two functional expressions can be used in two different ways: (a) as a means to formulate a restricted query or (b) as a constraint over the data set being analyzed (as in Figure 8(a)).

In Figure 8(b) we see four queries, Q_1, Q_2, Q_3 and Q_4 and four rewritings among these queries. Note that what makes possible the second and third rewriting is precisely the presence of the equality constraint. These rewritings are shown in Figure 9(a) encoded in the form of a graph, where each edge $Q \to Q'$ with label l means that Q' can be rewritten in terms of Q using the attribute l of the context. For example, the edge from Q_1 to Q_3 has label r because the following rewriting holds: $(r \circ b, q, sum) = (r, (b, q, sum), sum)$. We shall call this graph the "query rewriting graph". However, in order to define this concept formally we need the following definition.

Definition 11 (Comparing Grouping Functions).
Let f and g be two functions having the same source, say A.

- *(a) We shall say that f is less than or equal to g, denoted by $f \leq g$, if for all a, a' in A, $f(a) = f(a')$ implies $g(a) = g(a')$; or equivalently, if $\pi_f \leq \pi_g$ (i.e. if the partition of A induced by f is less than or equal to the partition of A induced by g).*

– *(b) We shall say that f and g are equivalent, denoted by $f \equiv g$, if $f \leq g$ and $g \leq f$ (i.e. if $\pi_f = \pi_g$).*

This definition can be extended to functional expressions in the obvious way. We note that two queries having equivalent grouping functions, the same measuring function and the same reduction operation are equivalent queries in the sense that they always return the same answer. This follows from the fact that equivalent grouping functions induce the same partition on their common source.

Definition 12 (Query Rewriting Graph). *Let $\mathcal{Q}$ be a set of queries to be evaluated in a given analysis context, such that:*

– *No two queries have equivalent grouping functions*
– *All queries have the same measuring function and the same reduction operation.*

We define the query rewriting graph of $\mathcal{Q}$ to be a directed graph with labelled edges such that:

– *The nodes of the graph are the queries of $\mathcal{Q}$*
– *There is an edge from node Q to node Q' if Q' can be rewritten in terms of Q. The label of the edge $Q \rightarrow Q'$ is the function on which is based the rewriting of Q' in terms of Q*

Now, rewriting is a binary relation, which is actually a partial order over queries. Indeed, we can show that rewriting is a reflexive, transitive and anti-symmetric relation (up to equivalence of grouping functions):

Reflexivity Every query Q can be rewritten in terms of itself (trivially, based on the identity function).

Transitivity Consider three queries $Q = (g, m, op)$, $Q' = (g', m, op)$ and $Q'' = (g'', m, op)$ such that: Q rewrites Q' and Q' rewrites Q''. It follows that: there exist functions h', h'' such that $g = h' \circ g'$ and $g' = h'' \circ g''$. Therefore we have: $g = h' \circ g' = h' \circ h'' \circ g'' = (h' \circ h'') \circ g''$ It follows that Q rewrites Q'' based on $(h' \circ h'')$.

Anti-symmetry Suppose Q rewrites Q' and Q' rewrites Q. It follows that there exist functions h', h'' such that $g' = h' \circ g$ and $g = h'' \circ g'$. Therefore we have: $g = h' \circ h'' \circ g$. It follows that $h' \circ h''$ is the identity function and therefore $Q = Q'$.

We shall denote by $\leq_w$ the above partial order defined by query rewriting.

Recall now that, given a set of queries, its query rewriting graph gives all possible rewritings among the queries in the set; and that each edge $Q \rightarrow Q'$ implies that query Q should be evaluated *before* query Q' if we want Q' to use the result of Q. Moreover, as we observed earlier, each query in the query rewriting graph might have more than one rewriting in terms of other queries; therefore a choice of one among the possible rewritings of each query is necessary before evaluating the given set of queries. So the problem is to find a subgraph

of the query rewriting graph such that (a) the evaluation order implied by the rewritings is maintained and (b) each query is evaluated exactly once. These observations lead to the following definition of query execution graph.

Definition 13 (Query Execution Graph). *A query execution graph for a set $\mathcal{Q}$ of queries is defined to be a subgraph EG of the query rewriting graph of $\mathcal{Q}$ such that:*

- *The nodes of EG are the queries of $\mathcal{Q}$ (i.e. EG has the same nodes as the query rewriting graph of $\mathcal{Q}$)*
- *Each node of EG has at most one predecessor*

Note that every query execution graph is an acyclic graph. This follows from the fact that query rewriting is a partial order and therefore the rewriting graph is an acyclic graph.

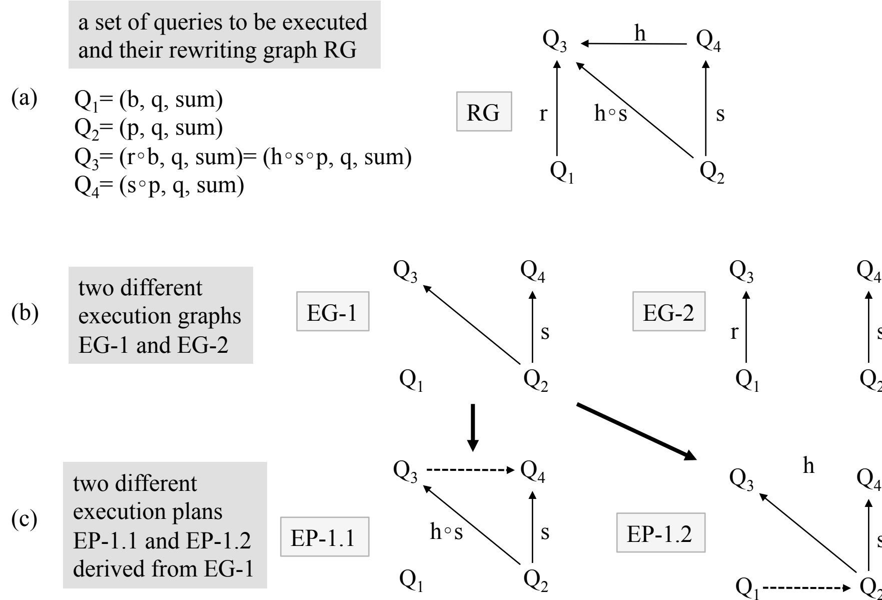

Fig. 9. A query rewriting graph RG, two of its query execution graphs, EG-1 and EG-2, and two query execution plans EP-1.1 and EP-1.2 from EG-1

Figure 9(a) shows a set of queries and their rewriting graph RG; and Figure 9(b) shows two query execution graphs, EG-1 and EG-2 of RG. The fact that an edge of the rewriting graph is not present in a query execution graph simply means that the corresponding rewriting is not considered as an option (for some reason). In other words the query execution graph allows the freedom of not

using some rewritings present in the rewriting graph. On the other hand, both, rewriting graphs and query execution graphs might have isolated nodes (as in EG-1). An isolated node simply means that it will be evaluated in isolation, neither reusing the result of another query nor being reused in the evaluation of another query).

Now, while the rewriting graph of a set $\mathcal{Q}$ of queries gives all possible rewritings among the queries in $\mathcal{Q}$, each query execution graph of $\mathcal{Q}$ gives those rewritings that are to be used during the evaluation of the queries in $\mathcal{Q}$. The rewritings of a query execution graph indicate the order in which the queries in $\mathcal{Q}$ should be executed in order to reuse the results of previously executed queries.

Privileging query rewriting is certainly reasonable, as query rewriting takes into account the semantics of data. However, apart from rewriting there might be other factors that influence the overall performance of query evaluation. In general, these other factors include the following:

- The physical storage of data, namely their format and their possible distribution
- The model of distributed computation
- The availability of cached query results
- The availability of processors and their configuration
- Load balancing
- etc.

In this work we privilege query rewriting and do not discuss the above factors any further. Instead, we assume that their influence is expressed as an ordering of the queries to be evaluated. We shall call this ordering the *external order* and we shall denote it by $\leq_e$. Actually, what we call a query execution plan, is a query execution graph together with an external order.

Definition 14 (Query Execution Plan). *A query execution plan for a set $\mathcal{Q}$ of queries is defined to be a query execution graph together with an external order $\leq_e$ compatible with the rewriting order $\leq_w$.*

Compatibility in the above definition means that the resulting graph is acyclic. Technically speaking, this means that the relation $\leq_w \cup \leq_e$ is an acyclic relation; we shall denote this relation by $\leq$. In case $\leq_w \cup \leq_e$ is not an acyclic relation, we shall remove a minimal number of external edges so that it becomes acyclic (and this is a non-deterministic process). In other words, we shall privilege the rewriting order while leaving in the result as big a part of the external order as possible.

We note that a query execution graph is itself a query execution plan (with the trivial order as external order). We also note that the rewriting order $\leq_w$ depends on the semantics of queries whereas the external order $\leq_e$ is application dependent.

Figure 9(c) shows two query execution plans, EP-1.1 and EP-1.2, derived from the query execution graph EG-1. The rewriting order $\leq_w$ is indicated by the edges in solid line, whereas the external order $\leq_e$ is indicated by the edges in

dotted line. The basic difference between an external edge and a rewriting edge is that the latter carries semantic information in addition to ordering information. For example, in EP-1.1 the external edge from Q_3 to Q_4 simply means that evaluation of Q_4 must follow that of Q_3, whereas the rewriting edge from Q_2 to Q_4 means that evaluation of Q_4 must follow that of Q_2 *and* that the evaluation of Q_4 must use the results produced by the evaluation of Q_2.

We note that it is not necessary for external edges to have labels. However, such labels might be useful. Indeed, the label of an external edge might be a clickable name, holding the address of a short text (e.g. in a web page) explaining the reasons for having the external edge. For example, in Figure 9(c), one reason for having the external edge from Q_3 to Q_4 could be that the evaluation of Q_3 uses much fewer resources than that of Q_4.

One of the intentions behind the definition of a query execution plan is that nodes which are non-comparable under $\leq$ can be evaluated in any order, or in parallel; whereas the evaluation of comparable nodes has to follow the order $\leq$.

For example, in the query execution plan EP-1.1, the nodes Q_1 and Q_2 can be evaluated in any order, or in parallel. This means that the evaluation of Q_1 can precede or follow that of Q_2, or it can be done in parallel with that of Q_2. By the way, a node such as Q_1 is called an *isolated node* as it is comparable to no other node in the query execution plan. On the other hand, the evaluation of nodes Q_3 and Q_4 will take place after that of Q_2, and moreover will use the results of the evaluation of Q_2. However, Q_3 will be evaluated before Q_4, as indicated by the external edge from Q_3 to Q_4.

As another example, in the query execution plan EP-1.2, the node Q_2 will be evaluated after Q_1 but will not use the results of the evaluation of Q_1; whereas the nodes Q_3 and Q_4 will be evaluated after Q_2, in any order or in parallel, but will both use the result of the evaluation of Q_2.

We note that the evaluation of a set $\mathcal{Q}$ of queries according to a query execution plan can be organized in various ways. For example, as query execution plans are acyclic graphs, one can organize evaluation to take place "levelwise". To explain how this is done, we first recall briefly the concept of "level" in an acyclic graph.

Definition 15 (Query Level). *The level of a query Q in a query execution plan, denoted by $l(Q)$, is defined as follows:*

- *if Q is a root then $l(Q) = 0$*
- *else $l(Q)$ is the length of a longest path from a root to Q*

We denote by L_i the set of all queries of level i, $i = 0, \dots k$, where k is the maximum length of path among all paths from a root of the query execution plan. For example, in the query execution plan EP-1.1 of Figure 9(c) the queries at level i are as follows, $i = 0, 1, 2$ (in this example, the length of a longest path is 2):

$$L_0 = \{Q_1, Q_2\},\ L_1 = \{Q_3\}, L_2 = \{Q_4\}$$

The following facts are immediate consequences of the above definition of level:

Fact 1 No two queries of L_i are comparable, $i = 0, \dots k$ (therefore the queries of L_i can be executed in any order or in parallel)

Fact 2 Each query Q' at level i has at most one predecessor at level $i-1$ (i.e. there is at most one rewriting of Q' in terms of a query Q at level $i-1$)

The sets L_i can be computed by topological sorting of the query execution plan, in time linear with respect to $n+e$, where n is the number of nodes and e is the number of edges of the query execution plan. Once the levels have been computed, evaluation can be done in a number of different ways. For example, one way is to follow the order of the levels (i.e. first executing the queries of L_0, then the queries of L_1, and so on).

It is important to note that, as all nodes of any given level are pairwise non comparable, their evaluation can be done in any order or in parallel. This means, in particular, that once the evaluation of a node Q at level i has terminated, we can proceed with the evaluation of the successors of Q (if any) at level $i+1$ (and this independently of all other nodes at level i).

We end this section with a few remarks regarding query execution plans. First, in our definition of query execution plan, we have privileged the rewriting order over the external order. Clearly one could privilege the external order over the rewriting order (or mix the two, provided that the result is a partial order). The method described above would remain the same, as the only thing that matters is acyclicity.

Second, our definition of query execution plan is "static", in the sense that the execution plan is defined before query evaluation starts, and it is followed until the last query is evaluated. However, our formal framework applies also in dynamic environments in which the actual order (i.e. the next query to be evaluated) is decided "on the fly" depending on the current values of a set of parameters (e.g. currently available processors). In this case, we start not with a query execution plan but with the query rewriting graph to which we add the external order; the resulting graph shows all possibilities of rewritings but will possibly have conflicts with the external order. Choosing one among all possible rewritings of a query and resolving conflicts between the rewriting order and the external order is done on the fly when deciding the next query to be evaluated. However, we shall not elaborate any further on dynamic query execution plans in this paper.

3 From Theory to Practice

As we mentioned in the introduction, data analysis is the process of highlighting useful information "hidden" in big data sets, usually with the goal to support decision making. Data analysis usually requires the execution of several analysis tasks, and a query $Q = (g, m, op)$, as defined in section 2.1, can be seen as a means to specify one such task.

In the formal model presented in the previous section, there is a clear separation between the conceptual and the physical level. An analytic query and its answer are defined at the conceptual level independently of the nature and location of data. However, the abstract definitions have to be mapped to lower level evaluation mechanisms, at the physical level, where the actual query evaluation is done.

In this section we first propose a conceptual query evaluation scheme and then we explain how this scheme can be mapped in three important cases, namely in MapReduce, in column databases and in row databases (i.e. relational databases). Moreover, we explain how our model can leverage structure and semantics in data in order to improve the performance of the query evaluation process.

3.1 Our Conceptual Query Evaluation Scheme

At the conceptual level, the answer to a query $Q = (g, m, op)$ is defined in a straightforward manner that we briefly recall here. Let $g_1, \ldots, g_n$ be the values of g and let $\pi_g = \{G_1, \ldots, G_n\}$ be the partition of D into blocks, induced by g (i.e. $G_j = g^{-1}(g_j), j = 1, \ldots, n$). Then the answer of Q on value g_j is defined as follows: $ans_Q(g_j) = red(m/G_j, op)$. In other words the answer of Q on value g_j is the reduction (under op) of the values of m on the block $G_j = g^{-1}(g_j)$.

However, this abstract definition of the answer assumes that the (extensions of the) functions g and m are stored in some repository and that they are accessible for processing. In practice, this assumption is rarely true. Actually, the functions g and m usually have to be *extracted* from the (possibly) much larger data set D in which they are embedded; and their extraction may require one or more joins, which are expensive operations.

Moreover, the data set from which the functions g and m are extracted is usually distributed, and this further complicates their extraction. Indeed, the data assets of an institution usually reside in several different repositories that can even be geographically distributed.

For example, in a big company with several branches, it is very likely that the sales transactions in each branch will be stored in a local database. As a result, data analysis queries concerning the company data assets, as a whole, will have to take into account data distribution. Moreover, even if we assume that all transactions are stored in a central repository, the size of the data set might require their deployment over several computational nodes for reasons of parallel processing. Indeed, in some application environments, the size of the data set is such that massive parallel processing is the only way to achieve acceptable performance.

The conceptual evaluation scheme that we propose in this section consists of three steps:

- Query input preparation
- π_g construction
- π_g reduction

In what follows we explain each step in detail. More precisely, we consider the query $Q = (g, m, op)$ over the data set D of Figure 3, and we explain how we can obtain its answer following the abstract definition of answer.

Query input preparation Let us call *query input*, denoted by $IN(Q)$, the set of triples $(i, g(i), m(i))$, where i is the data item identifier and $g(i)$, $m(i)$ are the values of i under the attributes g and m, respectively. This set of triples contains the data necessary for evaluating the query Q (together with op which we shall tacitly assume). We distinguish two cases, as shown in Figure 10:

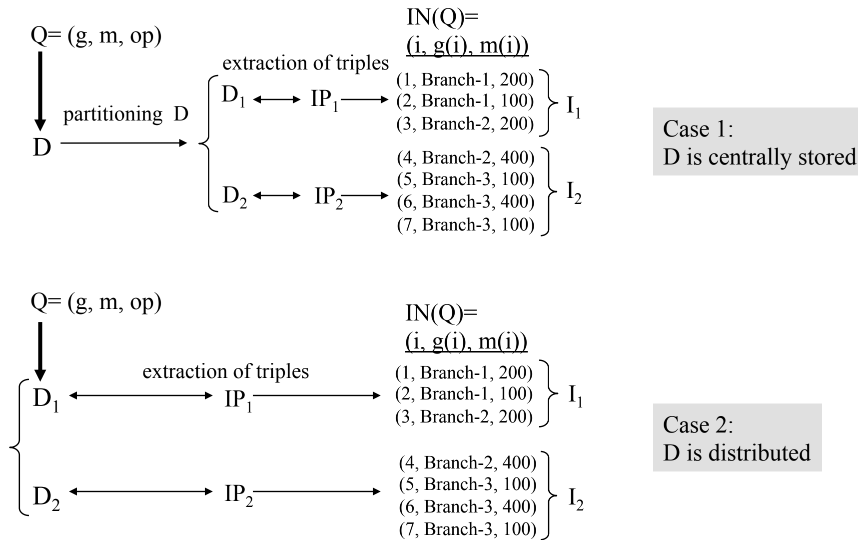

Fig. 10. Query input preparation

- *Case 1: The data set D is centrally stored.*
  In this case, D is first partitioned into k subsets $D_1, \ldots, D_k$, and each subset D_j is assigned for processing to an input processor IP_j (also called a "mapper"), which extracts the triples $(i, g(i), m(i))$ from D_j, $j = 1, \ldots, k$ (in Figure 10, in order to simplify our discussions, we have assumed that each subset is processed by exactly one input processor). The result of this phase is a partition $\pi_{IN(Q)} = \{I_1, \ldots, I_k\}$ of the input set $IN(Q)$. In the example of Figure 10, we have $k = 2$. Note that the k processors may reside in n nodes, where $n \leq k$.
- *Case 2: The dataset D is distributed in the form of k data sets, $D_1, \ldots, D_k$, stored in k locations (which might be geographically distant).*
  Let us assume that there is a processor IP_j (a mapper) assigned to each

location for extracting the triples $(i, g(i), m(i))$ at that location, and let I_j be the set of triples extracted from D_j, $j = 1, \ldots, k$. Again, the result of this phase is a partition $\pi_{IN(Q)} = \{I_1, \ldots, I_k\}$ of the input set $IN(Q)$.

We call the above process *query input preparation*. In each case, it returns k sets of triples $I_1, \ldots, I_k$ that form a partition $\pi_{IN(Q)}$ of the query input $IN(Q)$. Therefore $\pi_{IN(Q)} = \{I_1, \ldots, I_k\}$.

π_g construction This step constructs the partition $\pi_g = \{G_1, \ldots, G_n\}$, whose reduction will give the answer to the query. The partition π_g is constructed using the partition $\pi_{IN(Q)}$ of the previous step.

Note first that $\pi_{IN(Q)}$ and π_g are partitions of the same set, namely the query input $IN(Q)$. Therefore, conceptually, each block G_j of π_g intersects each block I_i of $\pi_{IN(Q)}$ (some intersections might be empty).

It follows that a block G_j can be constructed from its intersections with the blocks of $\pi_{IN(Q)}$. Let us call a *fragment* of G_j in $\pi_{IN(Q)}$ each of its intersections with a block of $\pi_{IN(Q)}$; and let us denote by F_{ji} the fragment of G_j in block I_i of $\pi_{IN(Q)}$ (i.e. $F_{ji} = G_j \cap I_i$). Then we have: $G_j = F_{j1} \cup \ldots \cup F_{jk}$.

Therefore, conceptually, each block G_j of π_g is made up of its fragments in $\pi_{IN(Q)}$. Now, to find the fragment F_{ji} of G_j in I_i, it is sufficient to do the following: group together all triples of I_i having the same value g_j.

However, each block I_i might also contain fragments of blocks of π_g other than G_j. One way to find all fragments of blocks of π_g contained in I_i is to sort the triples of I_i according to their values under g. Let us call *fragment construction* the process of finding all fragments of blocks of π_g contained in I_i, for $i = 1, \ldots, k$. Clearly, finding the fragments of blocks of π_g contained in two different blocks of $\pi_{IN(Q)}$ can be done in parallel.

Now, in order to construct each block G_j of π_g, the fragments of G_j in $\pi_{IN(Q)}$ will have to be put together. Let us suppose that the construction of each block G_j of π_g is under the responsibility of an output processor OP_j (also called a "reducer"). Let us also suppose that as soon as a processor IP_i computes its fragments, it sends them to the appropriate output processors (i.e. it sends F_{1i} to OP_1, F_{12} to OP_2, and so on). Clearly, as soon as processor OP_j receives the fragments sent by all processors IP_i it can construct the block G_j by taking the union of all fragments received. The output of this step is the partition π_g (hence the name "π_g construction"). The whole process is shown schematically in Figure 11.

π_g reduction Once the block G_j has been constructed by processor OP_j, it can be reduced (using *op*) to obtain the answer on the value g_j of g: $ans_Q(g_j) = red(m/G_j, op)$; and once this is done for $j = 1, \ldots, n$, the evaluation of the answer is complete.

Figure 11 shows schematically the π_g reduction phase as well as the flow of information during the three steps that lead to the computation of the answer, namely query input preparation, π_g construction, and π_g reduction.

In the following sections we shall see how these steps can be implemented in MapReduce, in column databases and in row databases.

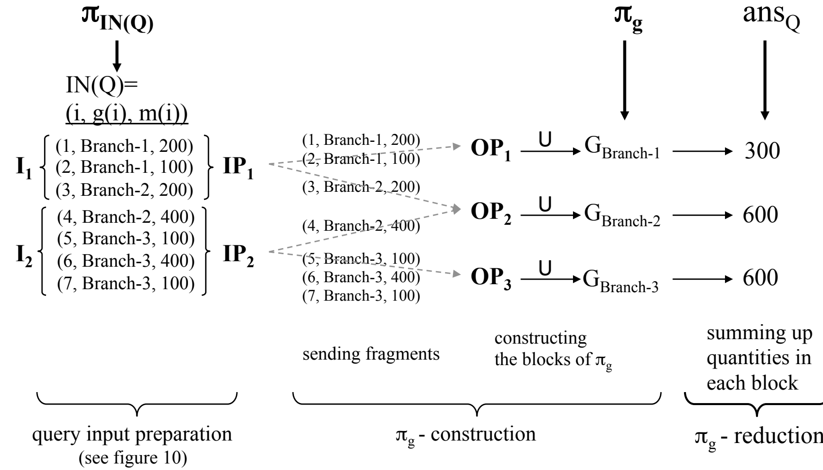

Fig. 11. The three steps of our conceptual evaluation scheme

However, before ending this section, we would like to make a few remarks regarding our conceptual evaluation scheme. First, in order to simplify the presentation, we have assumed as many input processors as there are blocks in the partition $\pi_{IN(Q)}$. Obviously this assumption is not necessary, as an input processor might be able to process two or more blocks; and conversely, a block might be further partitioned and assigned to two or more different processors. In general, the number of blocks into which the set of triples is partitioned (i.e. the partition $\pi_{IN(Q)}$), the number of input processors used and the assignment of blocks of $\pi_{IN(Q)}$ to input processors depend on several practical considerations (e.g. load balancing) that we do not discuss here.

The second remark is that the output processors do not have to work synchronously. In other words, as soon as the output processor OP_j receives all fragments of G_j, it can proceed to construct G_j (by taking the union of all fragments) and then it can reduce G_j to obtain $ans_Q(G_j)$ - and this independently of the remaining output processors.

The third remark is that our conceptual evaluation scheme can be improved by applying "early reduction". This means that, instead of sending fragments to the output processors, each input processor first reduces the fragments it contains, and then sends fragment reductions to the output processors (instead of sending the fragments themselves). However, early reduction can be done only if each output processor can compute the reduction of a block of π_g from the reductions of its fragments - and this depends on the reduction operation: if the

reduction operation is distributive, in the sense defined in section 2.3 (Definition 10) then early reduction is possible.

As a final remark, the conceptual evaluation scheme that we have seen in this section makes no claim of efficiency. Its only purpose is to show how the abstract definition of answer at the conceptual level can be reformulated so that to be mapped to existing evaluation mechanisms that work on the actual data.

In what follows we shall see first how our conceptual evaluation scheme can be mapped to MapReduce (section 3.2); then to column databases (section 3.3); and finally to row databases (section 3.4). In the last two cases, we shall also explain how the presence of functional dependencies influences the rewriting process.

3.2 MapReduce

Our conceptual evaluation scheme can be implemented in MapReduce in a straightforward manner using the correspondences shown in Figure 12(a).

	Our evaluation scheme	Map Reduce evaluation	
	query input preparation	map	
π_g construction	fragment construction	sorting	shuffling
	union of fragments	merging	
	π_g reduction	reduction	
	input processor	mapper	
	output processor	reducer	

Our terminology	Map Reduce Terminology
value of g	key
value of m	value
(g(i), m(i))	key-value pair

Fig. 12. The mapping of our evaluation scheme to MapReduce

Moreover, the correspondence between our terminology and that of MapReduce is shown in Figure 12 (b).

We note that, in MapReduce, fragment construction is done by sorting the triples of each block of $\pi_{IN(Q)}$. Clearly, this sorting can be done only under the assumption that the codomain of g is linearly ordered - which is usually the case. However, there are cases where this assumption does not hold. For example if g is the attribute "Color" then sorting of colors is not possible unless the colors are given some numeric code (at an additional cost). In such cases, grouping the triples of each block of $\pi_{IN(Q)}$ must be done in some other way.

Similarly, if the fragments of each block of π_g are sorted then it is possible to perform their union efficiently using merge-sort. Otherwise their union must be done in some other way.

Finally, we note that, in MapReduce, the construction of fragments of π_g blocks by the input processors (using sorting), together with the construction of π_g blocks by the output processors (using merge-sort) is referred to, collectively, as "shuffling".

3.3 Column Stores

There has been a significant amount of recent work on column oriented databases (column-stores) [1].

A column-oriented database stores data tables as sections of columns of data rather than as rows of data. In comparison, traditional relational databases store data in rows (row-stores). Column oriented databases have been shown to perform more than an order of magnitude better than traditional relational databases on analytical workloads such as those found in data warehouses, decision support, and business intelligence applications, or other ad-hoc query systems where aggregates are computed over large numbers of similar data items. The reason behind this performance difference is straightforward: column-stores are more I/O efficient for read-only queries since they only have to read from disk (or from memory) those attributes accessed by a query.

At the conceptual level, a column oriented database can be seen as storing a set of functions (i.e. attributes) of the form $f_A : ID_A \to A$, where ID_A is a subset of a set ID of identifiers used throughout the column store, and A is some set of values. Therefore mapping our conceptual evaluation scheme to a column store is straightforward. Indeed, let $g_A : ID_A \to A$ and $m_B : ID_B \to B$ be attributes in a column store, and suppose we want to evaluate the query $Q = (g_A, m_B, op)$, following our conceptual evaluation scheme.

During the query input preparation step, the query input set $IN(Q) = \{(i, g_A(i), m_B(i))/i \in ID_A \cap ID_B\}$ is computed by accessing the attributes $g_A : ID_A \to A$ and $m_B : ID_B \to B$ in the column store, and by "joining" them on ID. In other words, $IN(Q) = \{(i, g_A(i), m_B(i))/i \in ID_A \cap ID_B\}$. This completes the query input preparation step. The remaining two steps, namely, π_{g_A} construction and π_{g_A} reduction can be performed (in the way explained earlier) by passing the set $IN(Q)$ to MapReduce.

We note that if the attributes g_A and m_B reside in different column stores then we can compute the set $IN(Q)$ by performing a semi-join.

3.4 Row Stores

In this section, we explain how a query $Q = (g, m, op)$ in our language can be evaluated when the data set D is stored in a relational database. We present two different methods for doing the evaluation. Following the first method, the set $IN(Q)$ of triples is extracted from the database, using SQL, and then passed

to MapReduce for the remaining two steps, namely π_g construction and π_g reduction. Following the second method, the query Q is mapped directly to an SQL group-by query and evaluation is done by the database management system. In this second case, we show how our approach can leverage structure and semantics in data in order to improve performance of query evaluation.

Extracting the set of triples from the Database First we recall that the projection of a table T over an attribute A can be seen as a function, namely $proj_A : TID_T \to A$, defined as follows: $proj_A(t) = t(A)$, for all tuples t in T. Here $t(A)$ denotes the value of tuple t on attribute A (also called the A-value of t); and TID_T denotes the set of tuple identifiers in T. Note that this definition of projection corresponds to the concept of attribute in our approach, and also to the concept of attribute in column databases. Also note that, as TID_T does not appear explicitly in a relational table, it can be replaced by a key of the table.

Clearly, the definition of projection can be extended to a projection $proj_{AB} : TID_T \to AB$ over two (or more) attributes as follows: $proj_{AB}(t) = t(AB)$, for all tuples t in T, where $t(AB)$ denotes the value of tuple t over the attribute set AB (also called the AB-value of t). We note that the projection $proj_{AB}$ can be defined in an equivalent way using our definition of pairing (Definition 7): $proj_{AB} = proj_A \times proj_B$

Consider now a query $Q = (g_A, m_B, op)$ in our language, using two attributes g_A and m_B, where A and B are attributes appearing in the database, and let us see how we can evaluate Q using our conceptual evaluation scheme. In the first step (i.e. during query input preparation), in order to extract the set of triples $IN(Q)$, we distinguish two cases:

- *Case 1: The attributes A and B appear in the same table, say T.*
  In this case, we can obtain $IN(Q)$ by projection of T over the attribute set $\{TID_T, A, B\}$, that is: $IN(Q) = proj_{TID,A,B}(T)$. Using SQL, the set $IN(Q)$ can be obtained as the answer to the following query: `Select` TID_T`,` A`,` B `From` T.

  Clearly we can also use a "Where" clause if Q is an attribute-restricted query.

- *Case 2: The attributes A and B appear in two different tables, say S and T.*
  In this case a join is needed in order to obtain the set $IN(Q)$:
  $IN(Q) = proj_{TID_T,A,B}(S \bowtie T)$.
  Using SQL we will have: `Select` TID_T`,` A`,` B `From` $join(S,T)$

Clearly, the above method for extracting $IN(Q)$ can be extended to complex queries involving more than two attributes in a straightforward manner.

Once the set $IN(Q)$ is computed, the remaining two steps, namely, π_{g_A} construction and π_{g_A} reduction, can be performed by passing the set $IN(Q)$ to MapReduce.

Mapping directly to SQL Another way to evaluate the answer to the query $Q = (g_A, m_B, op)$ is to map it to a group-by SQL query as follows:

- *Case 1: The attributes A and B appear in the same table, say T.*
  In this case we can obtain the answer of Q using the following group-by statement of SQL:

  `Select` A, $op(B)$ `as` $ans_Q(A)$ `From` T `Group by` A

- *Case 2: The attributes A and B appear in two different tables, say S and T.*
  In this case we can obtain the answer of Q using the following group-by statement of SQL:

  `Select` A, $op(B)$ `As` $ans_Q(A)$ `From` $join(S,T)$ `Group by` A

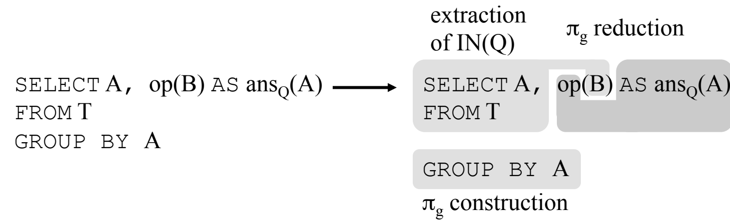

Fig. 13. The group-by query decomposed into the steps of our conceptual evaluation scheme

Figure 13 shows roughly the correspondence between our conceptual evaluation scheme and evaluation by a group-by statement.

In the above SQL queries, "$ans_Q(A)$" is a user-defined attribute, and in each of these two cases, the SQL query returns the answer of $Q = (g_A, m_B, op)$ in the form of a table with two attributes, A and $ans_Q(A)$. Clearly, attribute-restricted queries and result-restricted queries of our language can be mapped to SQL queries in the obvious manner, using "Where" and "Having" clauses, respectively.

Let us see some concrete examples of mapping analytic queries directly to SQL. To this end, we shall use the context of Figure 14 and we shall assume that the data set is stored in the form of a (relational) data warehouse under the star schema shown in that figure[4]. This star schema consists of the fact table FT and two dimensional tables: the dimensional table DT_{Branch} of $Branch$ and the

[4] We recall that a data warehouse is a read-only database in which data is accumulated over long periods of time for analysis purposes in decision support and business intelligence applications [16]

dimensional table $DT_{Product}$ of $Product$. The edges of the context are embedded in these three tables as functional dependencies that the tables must satisfy, and the underlined attribute in each of these three tables is the key of the table.

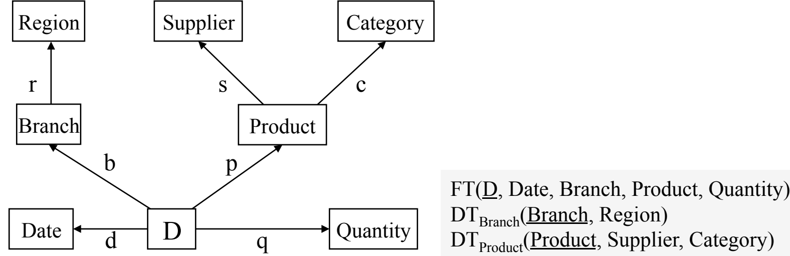

Fig. 14. A context and its underlying data stored in the form of a star schema

In this setting, consider the query $Q = (p, q, sum)$ over the context, asking for the totals by product. This query will be mapped to the following SQL query:

Select $Product$, $sum(Quantity)$ As $ans_Q(Product)$
From FT
Group by $Product$

As another example, consider the attribute-restricted query $Q = (b/E, q, sum)$, where $E = \{x|x \in D \wedge d(x) = 20/01/2014\}$, asking for the totals by branch in January 20, 2014. This query will be mapped to the following SQL query:

Select $Branch$, $sum(Quantity)$ As $ans_Q(Branch)$
From FT
Where Date = 20/01/2014
Group by $Branch$

Continuing with the previous example, consider the query $Q = (b/E, q, sum)/F$, where E is as above and $F = \{y|y \in Branch \wedge ans_Q(y) \leq 1000\}$. This query asks for the totals by branch in January 20, 2014, only for branches for which the total is less than or equal to 1000. This query will be mapped to the following SQL query:

Select $Branch$, $sum(Quantity)$ As $ans_Q(Branch)$
From FT
Where Date= 20/01/2014
Group by $Branch$
Having $ans_Q(Branch) \leq 1000$

As a final and rather extreme example, consider the query $Q = (\kappa_D, q, sum)$, asking for the total quantity delivered (independently of date, branch and product type). We recall that κ_D is the constant function on D (see section 2.3),

hence it will have to be mapped to the constant function over the table FT, which is the projection of FT over the empty set. Indeed, assuming FT is non empty, the function $proj_{\emptyset}$ is a constant function with the empty tuple as its only value. Therefore, the query $Q = (\kappa_D, q, sum)$ will be mapped to the following SQL query:

Select (), $sum(Quantity)$ As ans_Q
From FT
Group by ()

Note that, in the above SQL query, we use the symbol () to indicate the empty list of attributes.

As these examples show, the mapping of queries in our model to queries in SQL is straightforward (and can be easily rendered automatic, using a simple algorithm).

In the rest of this subsection, we explain how our approach can leverage structure and semantics in data in order to improve performance during query evaluation. To simplify the presentation, we discuss these two cases separately, using examples.

Leveraging structure Consider again the context shown in Figure 14, and suppose we need to evaluate the following three queries asking for the totals by supplier and category, by supplier, and by category, respectively:

$Q_1 = ((s \circ p) \times (c \circ p), q, sum)$

$Q_2 = (s \circ p, q, sum)$

$Q_3 = (c \circ p, q, sum)$

We have two options for evaluating these queries: evaluate them separately or evaluate them using rewriting. If we evaluate them separately, then we have to map them to the following SQL queries over the star schema:

Select $Supplier$, $Category$, $sum(Quantity)$ As $ans_{Q_1}(Supplier, Category)$
From $join(FT, DT_{Product})$
Group by $Supplier$, $Category$

Select $Supplier$, $sum(Quantity)$ As $ans_Q(Supplier)$
From $join(FT, DT_{Product})$
Group by $Supplier$

Select $Category$, $sum(Quantity)$ As $ans_Q(Category)$
From $join(FT, DT_{Product})$
Group by $Category$

In this case, the evaluation requires three joins between the tables FT and $DT_{Product}$. However, we observe that Q_2 and Q_3 can be rewritten in terms of Q_1:

$$Q_2 = (proj_{Supplier}, ans_{Q_1}, sum)$$

$$Q_3 = (proj_{Category}, ans_{Q_1}, sum)$$

Following this observation, we can define an execution plan according to which Q_1 is executed first and its result stored in a table, call it $AUX(Supplier, Category, TotQty)$; then the rewritten queries Q_2 and Q_3 are mapped to the following SQL queries:

```
Select Supplier, sum(Quantity) As ans_Q(Supplier)
From AUX
Group by Supplier

Select Category, sum(Quantity) As ans_Q(Category)
From AUX
Group by Category
```

In this case, the evaluation of the three queries requires one join between the tables FT and $DT_{Product}$ for evaluating Q_1 and two scans of the table AUX. Moreover, the evaluations of Q_2 and Q_3 over AUX can be done sequentially or in parallel (using a copy of AUX).

Now, the number of tuples in AUX is the cardinality of *Supplier* times the cardinality of *Category*; and this number is typically an order of magnitude smaller than the cardinality of the fact table FT. Therefore scanning AUX is less time consuming than scanning FT.

Consider now the case where the data is stored in a single table, say T, containing all the attributes appearing in the context of Figure 14:

$$T(D, Date, Branch, Product, Quantity, Region, Supplier, Category)$$

In this case, if we evaluate the three queries separately (i.e. without rewriting), we need no join. However, this time we need to scan a possibly huge table T three times (once for each query). However, if we use rewriting then we need only one scan of T to compute the totals by supplier and category, and two scans of AUX to compute the totals by supplier and by category.

Incidentally, the above two examples show clearly how our approach works: first we define queries and execution plans at the conceptual level; then we map them to a lower level where actual evaluation takes place; and the gains that we obtain depend on the underlying structure of data (i.e. on whether the data is stored in the form of a star schema or in a single table T). In other words, our approach can leverage structure in data to improve query evaluation. Furthermore, such improvements are compatible with optimizations that the database management system may perform on individual query evaluation. Indeed, while evaluating each of the queries of an execution plan, the database management

system may use its own query optimizer (if any) to further improve the query evaluation process.

Leveraging semantics Let us now see how our approach can leverage semantics in data to improve query evaluation. To this end we shall consider the case of the well known functional dependencies of the relational model, and we shall show how they can be incorporated into query rewriting. The basic observation here is that a functional dependency $X \to Y$ between attribute sets X and Y of a table T, is actually a function from X to Y, whose extension can be derived by projecting T over the union of X and Y.

The following proposition will help explain how functional dependecies can be incorporated in query rewriting, and therefore contribute in improving the query evaluation process.

Proposition 6. *Let $f : A \to B$ and $g : A \to C$ be two functions having the same source. Then $f \leq g$ iff there is a function $h : B \to C$ such that $g = h \circ f$. Moreover h is unique up to the range of f (i.e. if there is a function h' such that $g = h' \circ f$ then $h'/range(f) = h/range(f)$).*

Proof. Suppose first that there is a function $h : A \to B$ such that $g = h \circ f$. Then for all a, a' in A such that $f(a) = f(a')$ we have: $g(a) = h(f(a)) = h(f(a')) = g(a')$. Therefore $f \leq g$. Suppose next that $f \leq g$. It follows that $\pi_f \leq \pi_g$. Consider now any b in the range of f and define: $h(b) = g(f^{-1}(b))$. As $f \leq g$, the block $f^{-1}(b)$ of π_f is included in some block of π_g, say $g^{-1}(c)$, where c is in the range of g. It follows that: $g(f^{-1}(b)) = c$, therefore h is a well defined function over the range of f. Moreover, from the definition of h we have: $h(f(a)) = g(f^{-1}(b)) = g(b)$. Therefore $h \circ f = g$. Finally, suppose there is a function $h' : A \to B$ such that $h' \neq h$ and $h' \circ f = g = h \circ f$. Now, for any b in the range of f there is a in A such that $f(a) = b$. Therefore $h'(b) = h'(f(a)) = (h' \circ f)(a) = g = (h \circ f)(a)$. It follows that: $h'/range(f) = h/range(f)$ and this completes the proof.

As an immediate corollary we obtain the following proposition.

Proposition 7. *Let $X \to Y$ be a functional dependency over D. Then we have: $X \to Y$ holds in T if and only if there is a unique function $h : X \to Y$ such that $h \circ proj_X = proj_Y$.*

Note that the projections $proj_X$ and $proj_Y$ are seen as functions, in the way explained earlier.

We note that, if the functional dependency $X \to Y$ holds in T, then the (extension of) function h can be obtained by projecting the table T over XY (i.e. over the union of attribute sets X and Y).

As an immediate corollary of the above proposition, and our basic rewriting rule for compositions, we have the following rewriting rule:

Corollary 1. $(proj_Y, proj_Z, op) = (h, (proj_X, proj_Z, op), op)$

Following this rule, the evaluation of any query Q of the form $Q = (proj_Y, proj_Z, op)$ can be done by first evaluating the query $Q' = (proj_X, proj_Z, op)$ and then the query $Q'' = (h, ans_{Q'}, op)$.

Summarizing our discussion so far, it is clear that functional dependencies can be seamlessly integrated in our model, and moreover, they can be used for improving query evaluation performance through rewriting.

In fact, Proposition 7 above is a powerful tool for query rewriting when the data set is a relational table. Moreover, this proposition, combined with a well known result from relational database theory, provides the basis for studying not only query rewriting but also the generation of query execution plans. Indeed, a basic question is: given a query $Q = (proj_X, proj_Z, op)$ over a table T, what is the set of all queries of the form $Q' = (proj_Y, proj_Z, op)$ that can be rewritten using Q. It follows from Proposition 7 that a query of the form $Q' = (proj_Y, proj_Z, op)$ can be rewritten using Q only if the functional dependency $X \rightarrow Y$ holds in T.

Now, from relational database theory we know that: $X \rightarrow Y$ holds in T if and only if $Y \subseteq X^+$, where X^+ is the closure of the attribute set X with respect to the functional dependencies that T must satisfy [32]. Therefore to answer the above question we need an algorithm for computing X^+. Fortunately, there are efficient (linear) algorithms for the computation of X^+ from X ([32]).

Actually, the attributes of a relational table T together with the set of functional dependencies that hold in T can be seen as an analysis context. More precisely, let $\mathcal{A} = \{A_1, A_2, \ldots, A_m\}$ be the set of attributes of T, $\mathcal{F} = \{f_1, f_2, \ldots, f_n\}$ be the set of functional dependencies of T and TID the set of tuple identifiers of T. As explained earlier, each attribute A_i, can be seen as a function: $proj_{A_i} : TID \rightarrow A_i$. Then the set $\mathcal{A}F = \{proj_{A_1}, \ldots, proj_{A_m}, f_1, \ldots, f_n\}$ can be seen as a context, with TID as its single root, and we can call this context "the analysis context of T". Using this context we can express analytic queries over T, and this independently of whether T is normalized or not, or whether T is a database table or a table derived from the tables in the database.

Moreover, as attributes may be shared by two or more tables, we can envisage combining the results of analytic queries over the contexts of different tables. For example, referring to Figure 14, suppose that the dimensional table $DT_{Product}$ contains an additional attribute $u : Product \rightarrow UnitPrice$, giving the unit price for each type of product delivered. In other words, the table has now the following form:

$$DT_{Product}(Product, Supplier, Category, UnitPrice)$$

Consider now the following two queries, asking for (a) the total quantity delivered by category and (b) the average unit price by category:

$$Q_1 = (c \circ p, q, sum)$$

$$Q_2 = (c, u, avg)$$

The first query is expressed in the context of the join between the tables FT and $DT_{Product}$, whereas the second is expressed in the context of table $DT_{Product}$. Their results are the following two binary tables:

$$(Category, ans_{Q_1}(Category))$$

$$(Category, ans_{Q_2}(Category))$$

By combining these two analysis results we may discover, for example, that in some product category, although the average unit price is quite high, the total delivered quantity is quite high as well; and this might lead to useful insights as to the sales of products in that category.

4 Concluding Remarks

In this paper, we have presented a high level query language for expressing analytic queries over a big data set, based on its attributes. The main features of our approach are:

- A clear separation between the conceptual and the physical level: queries are defined at the conceptual level independently of the nature and location of data and then they are mapped to lower level evaluation mechanisms (MapReduce, SQL engines) where actual evaluation takes place.
- A lightweight interface, called an analysis context, consisting of a set of attributes arranged in a graph. Analysts can express analytic queries within their context by defining triples of the form (g, m, op), where g and m are attributes of the context having common source, and op is an operation over the target of m.
- A sound, yet simple mathematical basis (using functions and partitions) that allows a formal approach to query rewriting, and to the definition and generation of query execution plans.
- The ability to leverage structure and semantics in data in order to improve performance through rewriting.

Future work includes several research items. The first concerns the computation of the query rewriting graph of a set of queries. We have sketched how this can be done, based on the closure of the set of functional dependencies, in the case where the data set D is a relational table. We would like to use graph theoretic techniques in order to give a general algorithm for computing the rewriting graph.

The second research item concerns the extension of the concept of query execution plan for a set $\mathcal{Q}$ of queries, in two different directions: (a) Considering execution plans that, apart from the queries in $\mathcal{Q}$, contain queries outside $\mathcal{Q}$ which are able to rewrite one or more queries in $\mathcal{Q}$. The objective is to accept the extra cost of evaluating such “external” queries if their rewritings can contribute to improving the overall performance significantly. (b) Studying the definition

and use of dynamic query execution plans in the sense explained only briefly in the section on query execution plans.

The third research item concerns query result visualization and exploration, based on Currification, in the sense explained in the section on analysis contexts.

The last research item concerns change in the data set D. Indeed, throughout the paper we have tacitly assumed that the data set is "static". While this might be a reasonable assumption in several application environments, it is by no means true for big data sets in general. Indeed, there are application environments where the data set changes frequently and where the results to some important, continuous queries have to be updated frequently as well. The objective here is to study incremental algorithms that take as input the increment in data and produce the increment in the query result.

References

1. Abadi, D.J., Madden, S.R., Hachem, N.: Column-stores vs. row-stores: How different are they really? In: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data. pp. 967–980. SIGMOD '08, ACM, New York, NY, USA (2008), http://doi.acm.org/10.1145/1376616.1376712
2. Abouzeid, A., Bajda-Pawlikowski, K., Abadi, D., Silberschatz, A., Rasin, A.: Hadoopdb: An architectural hybrid of mapreduce and dbms technologies for analytical workloads. Proc. VLDB Endow. 2(1), 922–933 (Aug 2009), http://dx.doi.org/10.14778/1687627.1687731
3. Anderson, C.: The end of theory: The data deluge makes the scientific method obsolete. http://archive.wired.com/science/discoveries/magazine/16-07/pb_theory (Jun 2008)
4. Bain, T.: Was stonebraker right? http://blog.tonybain.com/tony_bain/2010/09/was-stonebraker-right.html (2010)
5. Bajda-Pawlikowski, K., Abadi, D.J., Silberschatz, A., Paulson, E.: Efficient processing of data warehousing queries in a split execution environment. In: Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data. pp. 1165–1176. SIGMOD '11, ACM, New York, NY, USA (2011), http://doi.acm.org/10.1145/1989323.1989447
6. Beyer, K.S., Ercegovac, V., Gemulla, R., Balmin, A., Eltabakh, M.Y., Kanne, C., Özcan, F., Shekita, E.J.: Jaql: A scripting language for large scale semistructured data analysis. PVLDB 4(12), 1272–1283 (2011), http://www.vldb.org/pvldb/vol4/p1272-beyer.pdf
7. Beyer, M.A., Laney, D.: The importance of 'big data': A definition. https://www.gartner.com/doc/2057415/importance-big-data-definition (Jun 2012)
8. Buneman, P., Frankel, R.E.: Fql: A functional query language. In: Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data. pp. 52–58. SIGMOD '79, ACM, New York, NY, USA (1979), http://doi.acm.org/10.1145/582095.582104
9. Chen, S.: Cheetah: A high performance, custom data warehouse on top of mapreduce. Proc. VLDB Endow. 3(1-2), 1459–1468 (Sep 2010), http://dx.doi.org/10.14778/1920841.1921020

10. Chu, C., Kim, S.K., Lin, Y.A., Yu, Y., Bradski, G., Ng, A.Y., Olukotun, K.: Map-reduce for machine learning on multicore. In: Proceedings of Advances in neural information processing systems NIPS 2006. pp. 281–288. MIT; 1998 (2006)
11. Cloudera: Why europes largest ad targeting platform uses apache hadoop. http://blog.cloudera.com/blog/2010/03/why-europes-largest-ad-targeting-platform-uses-hadoop/ (2010)
12. Cohen, J., Dolan, B., Dunlap, M., Hellerstein, J.M., Welton, C.: Mad skills: New analysis practices for big data. Proc. VLDB Endow. 2(2), 1481–1492 (Aug 2009), http://dx.doi.org/10.14778/1687553.1687576
13. Cutting, D., Cafarella, M.: Hadoop. http://hadoop.apache.org/ (2005)
14. Czajkowski, G.: Sorting 1pb with mapreduce. http://googleblog.blogspot.fr/2008/11/sorting-1pb-with-mapreduce.html (Nov 2008)
15. Dean, J., Ghemawat, S.: Mapreduce: Simplified data processing on large clusters. Commun. ACM 51(1), 107–113 (Jan 2008), http://doi.acm.org/10.1145/1327452.1327492
16. Devlin, B.: Data Warehouse: From Architecture to Implementation. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1996)
17. Dittrich, J., Quiané-Ruiz, J.A., Jindal, A., Kargin, Y., Setty, V., Schad, J.: Hadoop++: Making a yellow elephant run like a cheetah (without it even noticing). Proc. VLDB Endow. 3(1-2), 515–529 (Sep 2010), http://dx.doi.org/10.14778/1920841.1920908
18. Erez Aiden, J.B.M.: Cthe predictive power of big data. http://www.newsweek.com/predictive-power-big-data-225125 (Dec 2013)
19. Ewen, S., Schelter, S., Tzoumas, K., Warneke, D., Markl, V.: Iterative parallel data processing with stratosphere: An inside look. In: Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data. pp. 1053–1056. SIGMOD '13, ACM, New York, NY, USA (2013), http://doi.acm.org/10.1145/2463676.2463693
20. Friedman, E., Pawlowski, P., Cieslewicz, J.: Sql/mapreduce: A practical approach to self-describing, polymorphic, and parallelizable user-defined functions. Proc. VLDB Endow. 2(2), 1402–1413 (Aug 2009), http://dx.doi.org/10.14778/1687553.1687567
21. Gates, A.F., Natkovich, O., Chopra, S., Kamath, P., Narayanamurthy, S.M., Olston, C., Reed, B., Srinivasan, S., Srivastava, U.: Building a high-level dataflow system on top of map-reduce: The pig experience. Proc. VLDB Endow. 2(2), 1414–1425 (Aug 2009), http://dx.doi.org/10.14778/1687553.1687568
22. Graham, M.: Big data and the end of theory? http://www.theguardian.com/news/datablog/2012/mar/09/big-data-theory (Mar 2012)
23. Gupta, A., Mumick, I.S., Subrahmanian, V.S.: Maintaining views incrementally. SIGMOD Rec. 22(2), 157–166 (Jun 1993), http://doi.acm.org/10.1145/170036.170066
24. Hadapt: http://www.hadapt.com (2013)
25. Halevy, A.Y.: Answering queries using views: A survey. The VLDB Journal 10(4), 270–294 (Dec 2001), http://dx.doi.org/10.1007/s007780100054
26. Horowitz, M.: Visualizing big data: Bar charts for words. http://archive.wired.com/science/discoveries/magazine/16-07/pb_visualizing (Jun 2008)
27. House, W.: Big data across the federal government. http://www.whitehouse.gov/sites/default/files/microsites/ostp/big_data_fact_sheet_final.pdf (Mar 2012)

28. House, W.: Big data: Seizing opportunities, preserving values. http://www.whitehouse.gov/sites/default/files/docs/big_data_privacy_report_may_1_2014.pdf (May 2014)
29. Jacobs, A.: The pathologies of big data. Commun. ACM 52(8), 36–44 (Aug 2009), http://doi.acm.org/10.1145/1536616.1536632
30. James Manyika, Michael Chui, B.B.J.B.R.D.C.R., Byers, A.H.: Big data: The next frontier for innovation, competition, and productivity. http://www.mckinsey.com/insights/business_technology/big_data_the_next_frontier_for_innovation (May 2011)
31. Leinweber, D.: Stupid data miner tricks: How quants fool themselves and the economic indicator in your pants. http://www.forbes.com/sites/davidleinweber/2012/07/24/stupid-data-miner-tricks-quants-fooling-themselves-the-economic-indicator-in-your-pants/ (Jul 2012)
32. Maier, D.: Theory of Relational Databases. Computer Science Pr (1983)
33. Malewicz, G., Austern, M.H., Bik, A.J., Dehnert, J.C., Horn, I., Leiser, N., Czajkowski, G.: Pregel: A system for large-scale graph processing. In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data. pp. 135–146. SIGMOD '10, ACM, New York, NY, USA (2010), http://doi.acm.org/10.1145/1807167.1807184
34. Marcus, G., Davis, E.: Eight (no, nine!) problems with big data. www.nytimes.com/2014/04/07/opinion/eight-no-nine-problems-with-big-data.html (Apr 2014)
35. Melnik, S., Gubarev, A., Long, J.J., Romer, G., Shivakumar, S., Tolton, M., Vassilakis, T.: Dremel: Interactive analysis of web-scale datasets. Proc. VLDB Endow. 3(1-2), 330–339 (Sep 2010), http://dx.doi.org/10.14778/1920841.1920886
36. Monash, C.: ebays two enormous data warehouses. http://www.dbms2.com/2009/04/30/ (Apr 2009)
37. Monash, C.: ebay followup — greenplum out, teradata > 10 petabytes, hadoop has some value, and more. http://www.dbms2.com/2010/10/06/ (Oct 2010)
38. Ohm, P.: Dont build a database of ruin. http://blogs.hbr.org/2012/08/dont-build-a-database-of-ruin/ (Aug 2012)
39. Panda, B., Herbach, J.S., Basu, S., Bayardo, R.J.: Planet: Massively parallel learning of tree ensembles with mapreduce. Proc. VLDB Endow. 2(2), 1426–1437 (Aug 2009), http://dx.doi.org/10.14778/1687553.1687569
40. Pavlo, A., Paulson, E., Rasin, A., Abadi, D.J., DeWitt, D.J., Madden, S., Stonebraker, M.: A comparison of approaches to large-scale data analysis. In: Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data. pp. 165–178. SIGMOD '09, ACM, New York, NY, USA (2009), http://doi.acm.org/10.1145/1559845.1559865
41. Pierce, B.C.: Basic Category Theory for Computer Scientists. MIT Press, Cambridge, MA, USA (1991)
42. Spyratos, N.: A functional model for data analysis. In: Proceedings of the 7th International Conference on Flexible Query Answering Systems. pp. 51–64. FQAS'06, Springer-Verlag, Berlin, Heidelberg (2006), http://dx.doi.org/10.1007/11766254_5
43. Spyratos, N., Sugibuchi, T.: Restrict-reduce: Parallelism and rewriting for big data processing. In: Tanaka, Y., Spyratos, N., Yoshida, T., Meghini, C. (eds.) Information Search, Integration and Personalization, Communications in Computer and Information Science, vol. 146. Springer Berlin Heidelberg (2013), http://dx.doi.org/10.1007/978-3-642-40140-4_8

44. Stonebraker, M.: Mapreduce: A major step backwards. http://databasecolumn.vertica.com/2008/01/mapreduce_a_major_step_back.html (2008)
45. Stonebraker, M., Abadi, D., DeWitt, D.J., Madden, S., Paulson, E., Pavlo, A., Rasin, A.: Mapreduce and parallel dbmss: Friends or foes? Commun. ACM 53(1), 64–71 (Jan 2010), http://doi.acm.org/10.1145/1629175.1629197
46. Strachey, C.: Fundamental concepts in programming languages. Higher Order Symbol. Comput. 13(1-2), 11–49 (Apr 2000), http://dx.doi.org/10.1023/A:1010000313106
47. Swanstrom, R.: Colleges with data science degrees. http://101.datascience.community/2012/04/09/colleges-with-data-science-degrees/ (Apr 2012)
48. Thusoo, A., Sarma, J.S., Jain, N., Shao, Z., Chakka, P., Zhang, N., Anthony, S., Liu, H., Murthy, R.: Hive - a petabyte scale data warehouse using hadoop. In: Proceedings of the 26th International Conference on Data Engineering, ICDE 2010, March 1-6, 2010, Long Beach, California, USA. pp. 996–1005 (2010), http://dx.doi.org/10.1109/ICDE.2010.5447738
49. Vernica, R., Carey, M.J., Li, C.: Efficient parallel set-similarity joins using mapreduce. In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data. pp. 495–506. SIGMOD '10, ACM, New York, NY, USA (2010), http://doi.acm.org/10.1145/1807167.1807222
50. Vidal, M., Raschid, L., Marquez, N., Cardenas, M., Wu, Y.: Query rewriting in the semantic web. In: Data Engineering Workshops, 2006. Proceedings. 22nd International Conference on. pp. 7–7 (2006)
51. Yahoo: Hadoop terasort. http://developer.yahoo.com/blogs/hadoop/Yahoo2009.pdf (2009)
52. Yang, C., Yen, C., Tan, C., Madden, S.: Osprey: Implementing mapreduce-style fault tolerance in a shared-nothing distributed database. In: Proceedings of the 26th International Conference on Data Engineering, ICDE 2010, March 1-6, 2010, Long Beach, California, USA. pp. 657–668. ICDE'10, IEEE, Washington, DC, USA (2010)
53. Yang, H.c., Dasdan, A., Hsiao, R.L., Parker, D.S.: Map-reduce-merge: Simplified relational data processing on large clusters. In: Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data. pp. 1029–1040. SIGMOD '07, ACM, New York, NY, USA (2007), http://doi.acm.org/10.1145/1247480.1247602