FORTH-ICS/TR467

July 2017

HiFun - A High Level Functional Query Language for Big Data Analytics

Nicolas Spyratos, Tsuyoshi Sugibuchi

Laboratoire de Recherche en Informatique, Université Paris-Sud 11, France
`Nicolas.Spyratos@lri.fr`

Abstract. We present a high level query language, called *HiFun*, for defining analytic queries over big data sets, independently of how these queries are evaluated. An analytic query in HiFun is defined to be a well-formed expression of a functional algebra that we define in the paper. The operations of this algebra combine functions to create HiFun queries in much the same way as the operations of the relational algebra combine relations to create relational algebra queries. The contributions of this paper are as follows: (a) defining a formal framework (i.e. HiFun) in which to study analytic queries in the abstract in much the same way as the relational algebra provides the formal framework to study relational algebra queries; (b) showing that each HiFun query can be encoded as a map-reduce job, and also as a SQL group-by query when the data set is a relational database; and (c) defining a formal method for rewriting HiFun queries and, as a case study, showing how our method can be applied in the rewriting of map-reduce jobs and of SQL group-by queries. We emphasize that, although theoretical in nature, our work uses only basic and well known mathematical concepts, namely functions and their basic operations. [1]

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 volume of data accumulated by modern applications increases in unprecedented rates.

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.

[1] Work conducted while the first author was visiting at FORTH Institute of Computer Science, Crete, Greece (https://www.ics.forth.gr/)

The potential uses of such “big data”, have been recognized not only in the private sector but also at the highest administration levels [12][13].

Big data analytics, in particular, holds high expectations for a variety of human activities. However, it often demands real or near-real time information delivery, and latency is therefore avoided whenever and wherever possible. With this difficulty, many organizations have turned to a newer class of technologies that includes several big data platforms (such as Hadoop [6] and Spark [24]) as well as NoSQL databases [17]. Those technologies form the core of an open source software framework that supports the processing of large and diverse data sets across clustered systems.

In particular, the Apache Hadoop Big Data Platform [6], derived from papers on Google’s MapReduce and Google File System, has been generating a lot of interest in many areas. It is widely adopted in industry and it is also used to solve a number of non-trivial problems in academia. It is becoming the de facto data analysis standard, also used for the analysis of structured data, an area traditionally dominated by relational databases in data warehouse deployments.

Today, the collection, management and processing of big data is emerging as a new field, usually referred to as “data science”, and several universities and educational institutions already offer degrees in this field [21] [8][1].

In the context of big data, our work aims at developping a formal framework for the study of analytic queries, independently of how such queries are evaluated.

1.1 Motivation and Contributions

There is a proliferation of systems developed today for the parallel processing of big data analytics based on map-reduce [10] [22][15] [9][4][2]. Although Hadoop/MapReduce has prevailed for performing data analytics in recent years, there are currently dozens of big data projects under way either in the private sector or in the context of the Apache Software Foundation [3]. All these projects 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. What is missing here is a common formal framework in which to study analytic queries in the abstract, independently of the specifics of each system [7].

In this paper, we introduce such a formal framework called *HiFun* in which analytic queries are the well-formed expressions of a functional algebra to be defined shortly. We show that each such expression can be encoded either as a map-reduce job or as a SQL group-by query. We then study properties of these expressions and use the results in order to understand the capabilities of map-reduce and its commonalities with SQL.

To explain better the benefits from our approach, let us recall that by studying properties of relational algebra expressions we can obtain important results that we can then apply to their SQL encodings.

For example, one can prove the following equivalence between relational algebra expressions:

$proj_A(T) \equiv proj_A(proj_{AB}(T))$, for all attributes A and B of a table T

If we apply this theoretical result to the SQL encodings we obtain the following equivalence:

`Select` A `From` T $\equiv$ `Select` A `From (Select` A`,` B `from` T`)`

This means that, during query evaluation, if we replace the SQL query on the left by the one on the right we obtain the same result (and this leads in general to a performance gain).

Similar examples of equivalences or of containment between relational algebra expressions can be proved involving more complex expressions, and the results can be applied to their SQL encodings in any SQL-based system.

However, this kind of reasoning is not possible for SQL group-by queries, as they have no corresponding relational algebra abstractions.

The well-formed expressions of the functional algebra that we propose in this paper provide abstractions for both types of analytic queries, namely SQL group-by queries and map-reduce jobs. Therefore one can reason about analytic queries in our algebra, independently of how such queries are evaluated by lower level mechanisms.

2 Our approach through an example

In defining our language, the basic notion that we use is that of *attribute* of a 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" (denoted as cc) is seen as a function $cc : D \rightarrow Integers$ such that, for each tweet t, $cc(t)$ is the number of characters in t.

Let us see an example in detail in order to motivate the definition of analytic query as a well-formed expression of HiFun. 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, and the data on all invoices during the year is stored in a database for analysis purposes.

The information provided by each invoice would most likely be represented as a record with the following fields: Invoice number, Date, Branch, Product, Quantity. In our approach, we view this information as a set of four functions, namely d, b, p and q, as shown in Figure 1, where D stands for the set of all invoice numbers and the arrows represent attributes of D. Following this view,

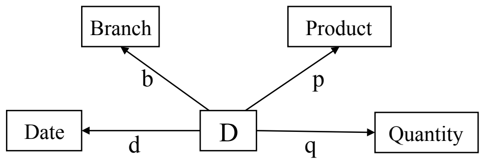

Fig. 1. Running example

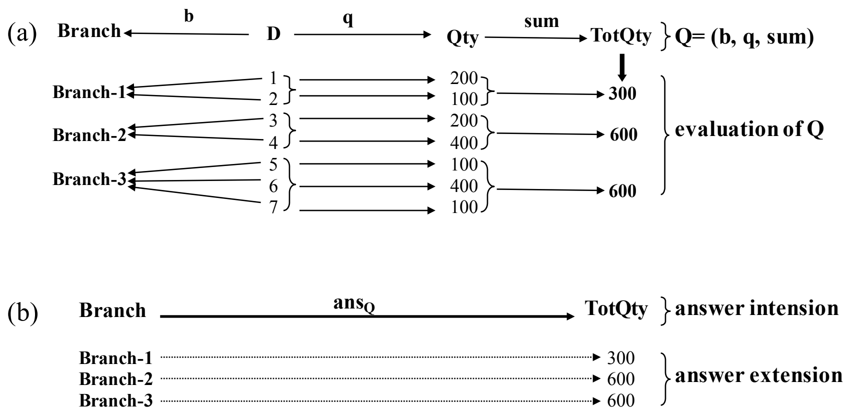

Fig. 2. An analytic query and its answer

given an invoice number, the function d returns a date, the function b a branch, the function p a product and the function q a quantity.

Suppose now that we want to know the total quantity delivered to each branch (during the year). This computation needs the extensions of only two among the four functions, namely b and q. Figure 2 (a) 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 (also 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 in the group (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, as shown in Figure 2, is the desired result.

We view the ordered triple $Q = (b, q, sum)$ as a query over D (see Figure 2 (a)), the function $ans_Q : Branch \to TotQty$ as the answer to Q (see Figure 2 (b)), and the computations in Figure 2 (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).

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, an ordered 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.

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 published; 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 function dd); then we find the number of characters per tweet (using function 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 the triple (dd, cc, avg).

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. Indeed, all we need to carry out the computations in Figure 2 (a) are the values of the attributes b and q mentioned in the query. Now, the method used to extract these attribute values depends on the structure of the data. For example, if the data resides in a relational database then one can use SQL in order to extract attribute values, whereas if the data is unstructured then one needs specialized algorithms to do the extraction (as in Hadoop).

Anyhow, at the conceptual level, we are not concerned with *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 performing various tasks (to be carried out by specific systems at a lower level).

Summarizing our discussion so far, a query in HiFun is defined to be an ordered 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 thus obtained 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. As we shall see shortly, this functional specification of an analytic query and its answer leads to non trivial results that can be applied to *any* system processing analytic queries.

2.1 Related work

In [18], a functional model was presented for data analysis in data warehouses over star schemas, using a definition of query similar to the one used in this paper. We build on that work by enlarging the scope to big data environments; by introducing the HiFun language and showing how HiFun queries can be encoded either as map-reduce jobs or as SQL group-by queries; and by defining a formal method for rewriting HiFun queries and, as a case study, applying the method to the rewriting of map-reduce jobs and of SQL group-by queries.

In [19], 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.

In previous work [5], 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 HiFun language.

The remaining of the paper is organized as follows. In section 3 we present the formal model, namely the definition of an analytic query, the concept of analysis context and the HiFun language. In section 4 we present how a HiFun query can be encoded either as a map-reduce job or as a SQL group-by query. In section 5 we present a formal method for rewriting HiFun queries and use it to define recursive analytic queries. In section 6 we present a case study, namely we use our formal method for rewriting HiFun queries to derive rewriting rules for map-reduce jobs and for SQL Group-by queries. Finally, in section 7 we present concluding remarks and outline research perspectives. All proofs of propositions are given in the appendix.

3 The Formal Model

3.1 The Formal Definition of Analytic Query

In the formal definition of analytic query 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 uniquely either 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 either by a tuple identifier or by a key of the table. In the remaining of the paper we shall often think of D as being the set of identifiers of its data items (e.g. in Figure 1 we can think of D as being the set of all invoice numbers).
- **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 published. In the remaining of the paper we shall use the terms "function on D" and "attribute of D" interchangeably.

We emphasize that, apart from the two assumptions above, we make no other assumption regarding the data set. In other words, the data set can be structured or unstructured, homogeneous or heterogeneous, centrally stored or distributed. Our results apply in all cases.

As we have already mentioned in the introduction, a query in our language is defined to be an ordered 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. Formally, we have the following definition:

Definition 1 (Query). *Let $D = \{d_1, \ldots, d_n\}$ be a finite set. An analytic query (or simply query) over D is an ordered 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 $V \neq W$.

In order to define formally the answer to a query over D, we need to define 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 of D induced by g.

We denote this partition by π_g, therefore we have:

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

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:

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

We now give the formal definition 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:*

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

where $m/g^{-1}(a_i)$ stands for the restriction of the function m to the subset $g^{-1}(a_i)$ of its domain.

Figure 2(b) shows schematically the answer of Q, denoted as ans_Q; and Figure 3 shows the relationship between the function ans_Q and the functions appearing in the query $Q = (g, m, op)$. As we shall see later, the fact that $ans_Q(a_i)$ is given by a closed formula facilitates the proofs of theorems when studying query rewriting.

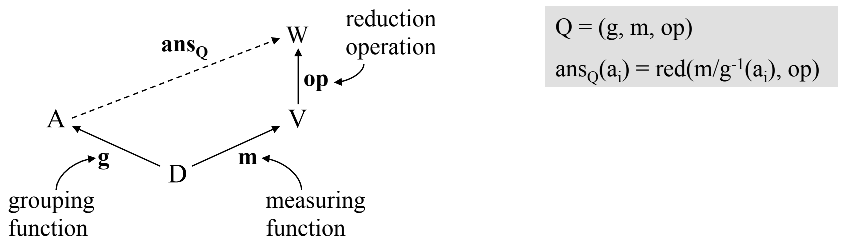

Fig. 3. A query Q and its answer, ans_Q

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.

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. It is evaluated by computing the restriction g/E and then evaluating the query $(g/E, m, op)$ over E*

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). It is evaluated by evaluating the query $Q = (g, m, op)$ over D, to obtain its answer (i.e. the function ans_Q), and then computing the restriction ans_Q/F*

As an example of attribute-restricted query, refer to Figure 2(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 b/E or on b (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.

It is worth noting that, in an attribute-restricted query, one can define the subset E of D by giving attribute values (instead of invoice numbers). For example, if we want the totals by branch only for product “Milk”, this will be expressed by the query: $(b/E, q, sum)$, where $E = \{i \in D/p(i) = Milk\}$.

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 2) and then restricting its answer 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) $= 600$

Note that, as ans_Q is a function, one can define the subset F of the domain of ans_Q by giving values of ans_Q (as in our example of attribute-restricted query above). For example, if we want the totals by branch only if the total is less or equal to 500, this will be expressed by the query: $(b, q, sum)/F$, where $F = \{b_i \in Branch/ans(b_i) \leq 500\}$.

Clearly one can combine attribute restriction and result restriction to express more complex queries. In our discussions we shall use the term "restricted query" to mean a query which is attribute-restricted and/or result-restricted.

We end this section with two technical remarks regarding the definition of a query and its answer.

First, 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.

Second, as the only requirement for g and m in the definition of a query is that they must be attributes of D, each of them can play the role of either a grouping function or a measuring function. Thus, in our running example, the queries $Q_1 = (b, q, count)$ and $Q_2 = (q, b, count)$, both are valid queries. The query Q_1 is the one we have already seen (asking for totals by branch) while the query Q_2 asks for the number of branches by delivered quantity. To answer Q_2 we use q to group together all invoices having the same delivered quantity (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). Therefore we have: $ans_{Q_2}(200) = 2$.

3.2 Analysis Context and the HiFun Language

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 d, b, p and q. These are "factual", or direct attributes of D as their values appear on the delivery invoices.

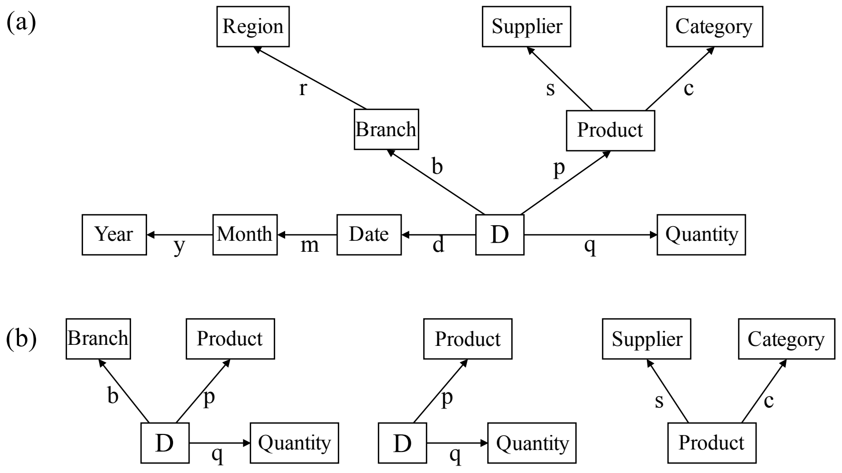

Fig. 4. Examples of contexts

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 4(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 delivered.

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.

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 4(a) shows a context in our running example, containing direct and derived attributes. Figure 4(b) shows three other contexts, one of which is not rooted in D (it is rooted in $Product$).

Actually, a context is the interface between the analyst and the data set, in the sense that the analysts use attributes of the context in order to formulate queries (in the form of triples as we saw earlier). Therefore a context plays the role of a schema. However, in contrast to, say, a relational schema, a context is *not* aware of structure in data (e.g. it is not aware of how attributes might be grouped together into tables). In other words, a context is a kind of "light weight" schema based on the semantics rather than the structure of data. Moreover, an analyst might want to use only a part of a context (i.e. a "sub-context") reflecting the analyst's needs (such as the sub-contexts of Figure 4(b)).

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 shall assume 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, in particular, that two edges can have the same label if they have different sources and/or different targets. 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, in which each node represents a set of independent values. Then for every node A of $\mathcal{C}$ the only possible cycle on A is ι_A, that is the identity function on A.*

The proof is given in the appendix. 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 (and can be represented as a single node equipped with the identity arrow on that node).

We note that, although acyclic, a context is not necessarily a tree. In particular, this means that a context can have parallel paths ("parallel" in the sense "same source and same target"); it also means that a context can have more than one root and this raises the following question: what does it mean for a context to have two or more different roots?

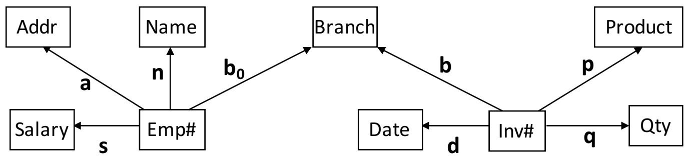

Fig. 5. A context with two roots

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 different data sets, possibly of different nature and possibly sharing one or more attributes. 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 inter-relationships of the data sets through their common attributes.

As an example, consider the context of figure 5 which has two roots: Emp# and Inv#. It consists of two "single-rooted" sub-contexts. The Employee sub-context (with Emp# as its root) represents personnel data of the enterprise; and the Invoice sub-context (with Inv# as its root) is that of our running example.

Note that the Employee sub-context shares the attribute Branch with the Invoice sub-context. As a result, an analyst of this context can formulate queries spanning both subcontexts, as the following example shows: (b_0, s, sum), asking for the total amount of salaries paid by branch.

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 the root. For example, in the context of Figure 4(a), suppose we add an attribute $u : Product \to UnitPrice$, 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 attributes to form complex grouping functions. For example, in the context of Figure 4(a), in order to ask for the total quantities by region, we need to use the composition $r \circ b$ as grouping function in expressing the query: $(r \circ b, q, sum)$.

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 [18]). 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 $f : X \to Y$ and $g : X \to Z$ be two functions with common domain X. The pairing of f and g, denoted $f \wedge g$ is a function from X to $Y \times Z$ defined by: $f \wedge g(x) = (f(x), g(x))$, 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 4(a) and consider the following query: $Q = (b \wedge p, q, sum)$. The answer to this query is a function, namely $asn_Q : Branch \times Product \to TotQty$ associating each pair $(branch, product)$ with a total quantity. In other words, Q asks for the total quantities delivered by branch *and* product.

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: $f \wedge g(x) = \{f : f(x), g : g(x)\}$, 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 of values. 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 4(a), we can ask for the total quantities by region and supplier, only for the month of January, using the following query:

– $(((r \circ b) \wedge (s \circ p))/E, q, sum)$, where $E = \{x | x \in D \wedge m \circ d(x) = January\}$

Here, $(((r \circ b) \wedge (s \circ p))/E$ means restriction of the function $(r \circ b) \wedge (s \circ p)$ to the subset E of its domain of definition).

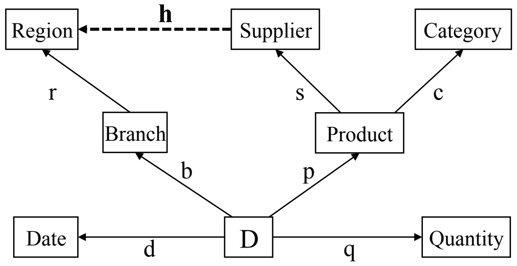

Fig. 6. A context with parallel paths

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: $((c \circ p)/E, q, sum)$, where $E = \{x | x \in D \wedge (h \circ s \circ p)(x) = (r \circ b)(x)\}$.

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 $\mathcal{C}$ 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 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 $e_2 = (r \circ b) \wedge p$ then $source(e_2) = D$ and $target(e_2) = Region \times Product$.

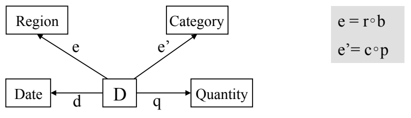

Fig. 7. A context with "complex" attributes (and their definitions)

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: $(e \wedge e', q, sum)$. 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 $\kappa_A : A \rightarrow K$.

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 that can be defined on a node A induce the set of all partitions of A. This set is partially ordered as follows: for all partitions π, π' of A, $\pi \leq \pi'$ if each block of π is a subset of a block of π'. Under this ordering the set of all partitions of A 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. In fact, the reason why we denote the unique value of K 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.

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 to form functional expressions and we can use such expressions in defining queries. For example, referring to Figure 4(a), consider the following queries using ι_D:

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

During the evaluation of Q_1, in the grouping step, the function ι_D puts each invoice of D in a single block. Therefore summing up the values of q in a block simply finds the value of q on the single invoice 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 node 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:

– $e \circ \iota_B = \iota_A \circ e = e$
– $e \circ \kappa_B = \kappa_A$

4 Evaluating HiFun Queries in Map-Reduce and in SQL

Let $\mathcal{C}$ be a context with a single root D and recall that D represents the data set of some application while the edges of $\mathcal{C}$ represent attributes of D. Moreover, suppose that the data set is stored in the file system of some big data platform (e.g. in the HDFS of the Apache Hadoop Big Data Platform). Let $Q = (e, e', op)$ be a HiFun query over $\mathcal{C}$ and suppose that we want to evaluate Q by encoding it as a map-reduce job over the underlying big data platform. To explain how this is done, consider the pairing $e \wedge e'$ of the functional expressions e and e' appearing in Q. This pairing returns a pair of values $(e(i), e'(i))$, for each item i in set D. Let's call the first value in this pair the *key* and the second value the *value* of the pair. Then the HiFun query Q can be encoded as a map-reduce job J_Q as follows:

Mapping step Each mapper receives the pair (e, e') to be used for the extraction of the key-value pairs $(e(i), e'(i))$, from each data item i.

Reduction step Each reducer uses the operation *op* of Q to reduce the set of key-value pairs received after shuffling has ended.

Figure 8 sketches how the query $Q = (b, q, sum)$ of our running example would be encoded as a map-reduce job using two mappers and three reducers (one reducer per branch). Each mapper reads the values of attributes Branch and Quantity from each data item in its current block (Branch being the key and Quantity being the value) and transmits the pairs to the appropriate reducers during shuffling (i.e. key-value pairs with key k_i go to reducer R_{Br_i}, $i = 1, 2, 3$). Each reducer sums up the quantities in the set of all pairs received to obtain a result v_i. The resulting association $k_i \to v_i$ is the answer to query Q. The whole process constitutes the map-reduce job J_Q evaluating the query Q.

Therefore every HiFun query Q can be encoded as a map-reduce job J_Q (over any big data platform using map-reduce as the basic tool for parallel processing). Clearly, what we said about HiFun queries of the form $Q = (e, e', op)$ can be extended to restricted HiFun queries in a straightforward manner.

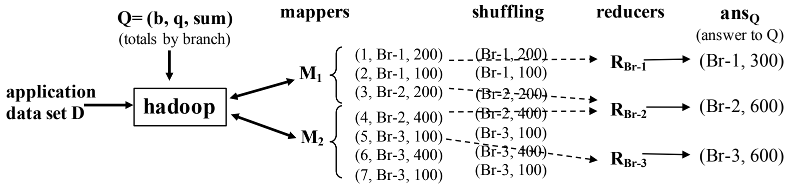

Fig. 8. Evaluating the query of our running example in Hadoop

In the opposite direction, consider a data set D and let $\mathcal{C}$ be the context rooted in D and having as edges all the attributes of D. Moreover suppose that the result of a map-reduce job is a function from keys to some domain of values. Then every map-reduce job J can be abstracted as a HiFun query Q_J over $\mathcal{C}$ as follows:

1. Let $(k(i), v(i))$ be the key-value pairs extracted during the mapping step, for all for all i in D. Then the associations $i \to k(i)$ and $i \to v(i)$, for all i in D, define two functions, $k : D \to K$ and $v : D \to V$, where K and V are value domains.
2. Let *op* be the operation applied to keys by the reducers of J. Then the triple $Q_J = (k, v, op)$ is the HiFun query Q_J abstracting J.

Note that the encoding $Q \to J_Q$ and the abstraction $J \to Q_J$ set up a 1-1 correspondence between HiFun queries and map-reduce jobs (up to the number of mappers used in the map-reduce job).

When the data set D resides in a relational table then instead of using map-reduce we can also use SQL to evaluate a HiFun query. To explain how this can be done we need some preliminary definitions and notation.

First we recall that the projection of a table T over an attribute A can be seen as a function, namely $proj_A : TID_T \rightarrow A$, defined as follows: $proj_A(t) = t(A)$, for all tuples t in T. Here, TID_T denotes the set of tuple identifiers in T; and $t(A)$ denotes the value of tuple t on attribute A (also called the A-value of 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 $proj_{AB} : TID_T \rightarrow AB$ 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): $proj_{AB} = proj_A \wedge proj_B$.

Now, consider again a context $\mathcal{C}$ with a single root D and suppose that the data set resides in a table $T_{\mathcal{C}}$ defined as follows: the attributes of $T_{\mathcal{C}}$ are the nodes of $\mathcal{C}$; the functional dependencies of $T_{\mathcal{C}}$ are the edges of $\mathcal{C}$; and D is a key of T.

Let $Q = (e, e', op)$ be a HiFun query over $\mathcal{C}$ and suppose that we want to evaluate Q by encoding it as a SQL group-by query G_Q over $T_{\mathcal{C}}$. In view of our discussion above, each of the expressions e and e' will be encoded as the set of attributes in its target, and the query Q will be encoded as a SQL group-by query G_Q over $T_{\mathcal{C}}$ defined as follows:

```
Select target(e), op(target(e')) As Result
From T
Group by target(e)
```

The answer to this query is a binary table with two attributes, $target(e)$ and $Result$ (recall that $Result$ is a user defined attribute that holds the aggregate result).

For example, let $\mathcal{C}$ be the context of Figure 1 and let $Q = (b \wedge p, q, sum)$ be the HiFun query over $\mathcal{C}$ (asking for the total by branch and product). Then Q will be encoded as the following SQL group-by query over $T_{\mathcal{C}}$:

```
Select Branch, Product, sum(Quantity) As Tot
From T
Group by Branch, Product
```

Therefore every HiFun query Q over a context $\mathcal{C}$ can be encoded as a SQL group-by query over the table $T_{\mathcal{C}}$.

In the opposite direction, consider a table T with a set $\mathcal{F}$ of functional dependencies and a key K of T. Then we can define a context $\mathcal{C}_T$ as follows:

1. the root of $\mathcal{C}_T$ is K
2. the edges of $\mathcal{C}_T$ are the functional dependencies in $\mathcal{F}$ together with the following set of edges: $\{f_A : K \to A / A \text{ is an attribute of } T\}$

Now, consider a SQL group-by query G over T defined as follows:

```
Select X, op(Y) As Result
From T
Group by X
```

Then G can be abstracted as a HiFun query Q_G over the context $\mathcal{C}_T$ defined as follows: $Q = (proj_X(T), proj_Y(T), Op)$.

Note that the encoding $Q \to G_Q$ and the abstraction $G \to Q_G$ defined above set up a 1-1 correspondence between HiFun queries and SQL group-by queries over a table. The extension of this correspondence when the context contains more than one root is rather straightforward: each rooted subcontext leads to the definition of a different table, and in the opposite direction, each table leads to the definition of one rooted subcontext (with two or more subcontexts possibly sharing nodes). Moreover, what we said about HiFun queries of the form $Q = (e, e', op)$ holds also for restricted HiFun queries. Indeed, attribute restricted HiFun queries simply require a "where" clause in their SQL encoding to define the set to which the attribute domains are restricted; and answer-restricted HiFun queries require a "having" clause to define the set to which the domain of query answer is restricted.

The 1-1 correspondence between HiFun queries and map-reduce jobs and between HiFun queries and SQL group-by queries are very important. Indeed, as we shall see shortly, these 1-1 correspondences allow to study properties of analytic queries at the HiFun level and then apply them to their encodings in map-reduce and in SQL.

We end this section with two rather extreme examples of encoding, namely encoding HiFun queries that use the identity and the constant function. Consider first the query $Q = (\kappa_D, q, sum)$ in the context of our running example, asking for the total quantity delivered (independently of date, branch and product type). We recall that κ_D is a constant function on D (see section 3.2). If we assume a single table T containing all attributes of the context then κ_D will have to be encoded as the constant projection over the table T, which is the projection of T over the empty set. Indeed, assuming T is non empty, the function $proj_\varnothing$ is a constant function with the empty tuple as its only value. Therefore, the query $Q = (\kappa_D, q, sum)$ will be encoded as the following SQL query (where the symbol () stands for the empty list of attributes):

```
Select (), sum(Quantity) As Tot
From T
Group by ()
```

Consider next the query $Q = (\iota_D, q, sum)$, whose answer is simply q. Then ι_D will have to be encoded as the identity projection over the table T, which is the projection of T over the whole set of attributes (alternatively, over the attributes of a key).

5 Query Rewriting in HiFun

Query rewriting deals with the following problem: how can we express a query in terms of one or more other queries. In this section we present the basic rewriting rules in HiFun for queries having the same measuring function and the same operation but different grouping functions. First, we note that query rewriting has two major applications: (a) optimizing the evaluation of a query; this is done by rewriting an incoming query in terms of other queries which have already been evaluated and their results stored (for example in a cache) and (b) optimizing the evaluation of a set $\mathcal{Q}$ of queries, 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.

We note that query rewriting has been studied extensively in the 1990s (see [11] for a survey) and it is still an active topic of research in areas such as the semantic web [23].

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 4(a):

– $Q = (p, q, sum)$, asking for the totals by product
– $Q' = (c \circ p, q, sum)$, asking for the totals by 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 of Q as its measure: $Q'' = (c, ans_Q, sum)$, asking for the sum of product totals by 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 $m : X \rightarrow V$ 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: $ans_{Q'} = ans_{Q''}$*

See appendix for the proof. In view of the previous proposition, we shall adopt the following notation:

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).

This recursive evaluation is shown on the left of Figure 9, using the query $Q = (r \circ b, q, sum)$, asking for totals by region. We first unfold the query Q using the basic rewriting rule until we reach the base query Q_0; then we evaluate the base query and use its answer to compute the answer to the query $Q_1 = (b, q, sum)$; and finally we use the answer of Q_1 to compute the answer of Q.

Note the parallel between this recursive evaluation of Q and the evaluation of 3!, shown on the right of Figure 9 .

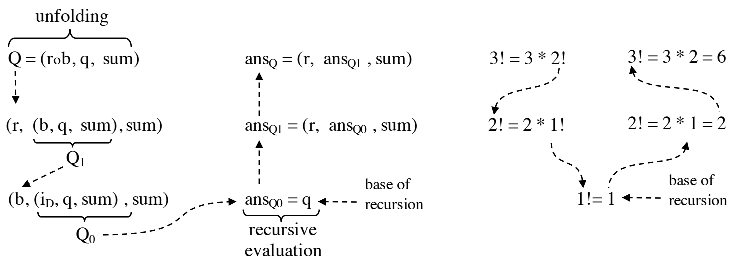

Fig. 9. Recursive evaluation of the analytic query "totals by region"

Clearly, if Q_1 has already been evaluated and its answer stored (e.g. in a cache) then ans_{Q_1} can be used in evaluating Q (thus accelerating the evaluation of Q).

Note how the identity function ι_D is indispensable in defining the base query Q_0, based on the following equality: $b = b \circ \iota_D$. Indeed, it follows from this equality that:

- $Q_1 = (b, q, sum) = (b \circ \iota_D, q, sum) = (b, (\iota_D, q, sum), sum)$

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 [16]).

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

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

In other words, each of the factors of $f \wedge g$ can be reconstructed from $f \wedge g$ 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:

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

To see how this rewriting rule for pairings works, refer to Figure 4(a) and suppose that the query $Q = (b \wedge p, q, sum)$ 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:

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

We end this section by noting that rewriting analytic queries actually boils down to decomposing the grouping function "losslessly". Indeed, let us look again at our basic rewriting rule:

- $(g \circ f, m, op) = (g, (f, m, op), op)$

What this rule actually does is to decompose the grouping function $g \circ f$ into two functions, g and f that are subsequently used as grouping functions in two different (and simpler) analytic queries whose results are combined to produce the result of the original query. This remark holds also for query rewriting based on pairing.

6 Case Study: Rewriting in Map-Reduce and in SQL

In this section we use rewriting of HiFun queries to define rewriting rules for map-reduce jobs and for SQL group-by queries. We proceed as follows:

1. We abstract the map-reduce job or the group-by query as a HiFun query.
2. We rewrite the HiFun abstraction (using the rules of the previous section).
3. We encode the rewritten HiFun abstraction in map-reduce or in SQL.

6.1 Rewriting Map-Reduce Jobs

Let J be a map-reduce job and $Q_J = (k, v, op)$ its HiFun abstraction, where k and v are the functions used by the mappers to extract key-value pairs from data items during the mapping step; and op is the operation applied by the reducers.

As we have seen in the previous section, if k is the composition of two other functions, then we can rewrite Q_J by applying the basic rewriting rule of HiFun. In other words, if $k = g \circ f$, then Q_J can be rewritten as follows: $Q_J = (g, (f, v, op), op)$. This implies that the map-reduce job J can be rewritten as a sequence of two other jobs, J_f and J_g defined as follows: $J_f = (f, v, op)$ and $J_g = (f, ans_{J_f}, op)$.

Therefore an alternative way to run J is the following:

1. run J_f on the data set D to obtain the answer ans_{J_f}
2. store ans_{J_f} (e.g. in the HDFS)
3. run J_g on the dataset ans_{J_f}

We illustrate this process, assuming that the data set D is a set of Web access events from different IP addresses, and that we want to run a map-reduce job returning the total number of Web accesses by city. Let f_I be the function that associates each event identifier with an IP address, and let g_C be the function that associates each IP address with a city name. Suppose f_I can be extracted from log files in HDFS, and g_C can be implemented as an external geolocalization service which is relatively slower than data access to HDFS. Clearly, running two jobs, J_{f_I} then J_{g_C} instead of $J_{g_C \circ f_I}$ is more efficient because it applies g_C to every distinct IP address instead of applying it to every event identifier. However, note that, to perform such staged evaluation of map-reduce jobs, we need to export intermediate results produced by one job into HDFS to pass it to another job. New generations of distributed computing frameworks, including Spark, can handle it more efficiently by using in-memory data cache. However, implementation details lie outside the scope of the present paper.

6.2 Rewriting SQL Group-by Queries

Let G be the following group-by query:

G: `Select` X`,` $op(Y)$ `As` *Result*
`From` T
`Group by` X

As explained earlier, G can be abstracted as a HiFun query $Q_G = (proj_X, proj_Y, op)$. If $proj_X$ is the composition of two other functions, then we can rewrite Q_G by applying the basic rewriting rule of HiFun. It turns out that this is possible if T satisfies a functional dependency whose right-hand side is X.

The following proposition explains how functional dependencies can be incorporated in query rewriting, and therefore how they can help improve the query evaluation process.

Proposition 5. *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 $h'/range(f) = h/range(f)$).*

The proof is given in the appendix. As an immediate corollary we obtain the following proposition.

Proposition 6. *Let $X \to Y$ be a functional dependency over T. 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 the attribute sets X and Y).

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

– $(proj_Y, proj_Z, op) = (h, (proj_X, proj_Z, op), op)$

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 $Q'' = (h, ans_{Q'}, op)$.

As an example, consider a database containing two tables: $T_1(Emp, Dep, Sal)$ and $T_2(Dep, Div)$. The table T_1 contains employees, their departments and their salaries, while the table T_2 contains departments and the divisions they belong to. Moreover, suppose that Emp is the key of T_1 and Dep is the key of T_2.

Consider the following group-by query (call it G), asking for the total salaries by division:

```
G: Select Div, sum(Sal) As TotDiv
   From join(T1,T2)
   Group by Div
```

The HiFun abstraction of G is the query $Q_G = (proj_{Div}, proj_{Sal}, sum)$. As we have the dependency $h : Dep \to Div$ we can rewrite Q_G as follows: $Q_G = (h, (proj_{Dep}, proj_{Sal}, sum), sum)$. Now, let G' be the SQL encoding of the nested query $(proj_{Dep}, proj_{Sal}, sum)$, which returns the total salaries by department:

G': `Select` Dep`,` $sum(Sal)$ `As` $TotDep$
`From` T_1
`Group by` Dep

We can now rewrite G as the SQL encoding of Q_G, using G':

Rewritten G: `Select` Div`,` $sum(TotDep)$ `As` $TotDiv$
`From Join(`$ans_{G'}$`,` T_2`)`
`Group by` Div

Remember that $ans_{G'}$ (the answer of G') is a table containing departments and their totals and observe the following:

1. Evaluating G requires joining T_1 and T_2, of which T_1 is potentially very large (it contains as many tuples as there are employees).
2. Evaluating the rewritten G requires joining $ans_{G'}$ and T_2, of which $ans_{G'}$ is usually small (it contains as many tuples as there are departments)
3. If G' has already been evaluated and its answer stored (e.g. in a cache) then $ans_{G'}$ can be used in evaluating G (thus accelerating the evaluation of G).

Summarizing our discussion so far, it is clear that Proposition 6 above allows for the seamless integration of functional dependencies in the rewriting of SQL group-by queries, thus improving performance during query evaluation. Moreover, this proposition, combined with a well known result from relational database theory, helps answer a basic question regarding the rewriting of group-by queries.

Indeed, consider a group-by query G' over a table T and its HiFun abstraction $Q_{G'} = (proj_X, proj_Z, op)$. Then a basic question is: what is the set of all group-by queries G with HiFun abstraction of the form $Q_G = (proj_Y, proj_Z, op)$ that can be rewritten using G'. It follows from Proposition 6 that G can be rewritten using G' only if a functional dependency of the form $X \to Y$ holds in T.

Now, from relational database theory we know that: $X \to Y$ 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 [14]. 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 ([14]).

We end this section by noting that the rewriting of map-reduce jobs and group-by queries is just one example demonstrating the expressive power of HiFun as a formal framework for studying analytic queries. Other examples not presented in this paper because of lack of space include the visualization of query results and the definition and generation of query execution plans (see [20] for more details).

7 Concluding Remarks

We have seen HiFun, a high level functional query language for expressing analytic queries over big data sets. The main features of HiFun are as follows:

1. Hifun considers the attributes of the data set as functions and uses a functional algebra that combines attributes to form analytic queries. HiFun queries are easy to express (in the form of triples) and can be evaluated by encoding them either as map-reduce jobs or as SQL group-by queries.
2. HiFun offers a clear separation between the conceptual level, where analytic queries are defined and the physical level where analytic queries are evaluated. Users of HiFun express their queries against a set of attributes arranged in a graph (and called a context).
3. HiFun offers a formal approach to query rewriting which is directly applicable to the rewriting of map-reduce jobs and of SQL group-by queries. In the latter case, it also allows the seamless integration of functional dependencies in the rewriting process.

Future work includes two research items. The first 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.

The second item concerns the extension of the functional algebra on which HiFun is based so that it strictly includes the relational algebra. The important implication of such an extension would be to unify usual SQL queries of the form select-from-where, group-by queries and map-reduce jobs under the same formal framework, thereby unifying the data management platform.

References

1. Data science institute / institut pour la science des données. `http://www.datascienceinstitute.org/`, accessed: 2015-12-19
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. Apache Software Foundation: Apache project list - category: Big data. `https://projects.apache.org/projects.html?category` (Feb 2017)
4. Beyer, K.S., Ercegovac, V., Gemulla, R., Balmin, A., Eltabakh, M.Y., Kanne, C., Özcan, F., Shekita, E.J.:
5. 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`
6. Cutting, D., Cafarella, M.: Hadoop. `http://hadoop.apache.org/` (2005)
7. Drineas, P., Huo, X.: Theoretical foundations of data science. `http://www.cs.rpi.edu/TFoDS/` (Apr 2016)

8. Erez Aiden, J.B.M.: Cthe predictive power of big data. http://www.newsweek.com/predictive-power-big-data-225125 (Dec 2013)
9. 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
10. 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
11. 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
12. 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)
13. 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)
14. Maier, D.: Theory of Relational Databases. Computer Science Press, New York (1983)
15. 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
16. Pierce, B.C.: Basic Category Theory for Computer Scientists. MIT Press, Cambridge, MA, USA (1991)
17. Sadalage, P.J., Fowler, M.: NoSQL distilled: a brief guide to the emerging world of polyglot persistence. Addison-Wesley Professional, Boston (Aug 2012)
18. 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)
19. 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, pp. 11–20. Springer Berlin Heidelberg, Heidelberg (2013), http://dx.doi.org/10.1007/978-3-642-40140-4_2
20. Spyratos, N., Sugibuchi, T.: A high level query language for big data analytics. Tech. Rep. 1575, CNRS – Université Paris Sud LRI, Orsay, France (2014), https://www.lri.fr/bibli/Rapports-internes/2014/RR1575.pdf
21. Swanstrom, R.: Colleges with data science degrees. http://101.datascience.community/2012/04/09/colleges-with-data-science-degrees/ (Apr 2012)
22. 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
23. 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)

24. Zaharia, M.: An Architecture for Fast and General Data Processing on Large Clusters. Association for Computing Machinery and Morgan &#38; Claypool, New York, NY, USA (2016)

Appendix

Proof (of Proposition 1). Let A and B be two nodes of $\mathcal{C}$, and suppose there is a cycle on A consisting of two functions: $f : A \to B$ and $g : B \to A$. 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.

Proof (Proof: Proposition 2). 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:

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

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

Proof (of Proposition 3.1). 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.