Datenbank Spektrum
DOI 10.1007/s13222-012-0112-8

SCHWERPUNKTBEITRAG

Compilation of Query Languages into MapReduce

Caetano Sauer · Theo Härder

Received: 26 November 2012 / Accepted: 20 December 2012

Abstract The introduction of MapReduce as a tool for Big Data Analytics, combined with the new requirements of emerging application scenarios such as the Web 2.0 and scientific computing, has motivated the development of data processing languages which are more flexible and widely applicable than SQL. Based on the Big Data context, we discuss the points in which SQL is considered too restrictive. Furthermore, we provide a qualitative evaluation of how recent query languages overcome these restrictions. Having established the desired characteristics of a query language, we provide an abstract description of the compilation into the MapReduce programming model, which, up to minor variations, is essentially the same in all approaches. Given the requirements of query processing, we introduce simple generalizations of the model, which allow the reuse of well-established query evaluation techniques, and discuss strategies to generate optimized MapReduce plans.

1 Introduction

The popularity of MapReduce arises mainly from two factors: (i) distribution transparency, which enables users to specify only the sequential, independent parts of the computation, while the framework takes care of the execution in parallel; and (ii) flexibility to implement specific data formats, serialization schemes, partitioning functions, etc. Because of these characteristics, MapReduce hits an abstraction sweet-spot, serving as a general tool for easily scaling out data-intensive computations.

This paper focuses on efforts to push the programming abstraction further by providing higher-level data processing languages which resemble query languages like SQL and XQuery. This allows users to specify computations using data processing operations such as selection, projection, grouping, aggregation, join, etc.—operations which are cumbersome to implement directly in the MapReduce programming model. Besides providing more specific operations, such languages also specify richer data models to more conveniently represent user data and provide more expressive power to operations.

The flexibility of MapReduce aims to support some of the requirements of *Big Data*, which involves a broader set of computation patterns and data types than those provided by relational databases. Therefore, SQL is considered too restrictive for such scenarios. In Sect. 2, we elaborate on these restrictions by defining a list of desired characteristics for MapReduce query languages and provide a qualitative analysis of existing approaches according to such criteria.

Having defined the desired characteristics of a higher-level query language, we go back to the lower layer of MapReduce and discuss how its computational model could be adapted to improve the support for such languages. In Sect. 3, we specify a slightly generalized model, which is, up to implementation aspects, equivalent to that of Hadoop, the popular open-source implementation of MapReduce. The discussion of the generalized model serves mainly the purpose of providing an informal but precise specification of the actual features and tunings of MapReduce being used in real applications. It aims to fulfill the gap between the basic MapReduce programming model [5] and the one actually used in practice. Furthermore, the specified model serves as basis for the following sections.

C. Sauer · T. Härder (✉)
University of Kaiserslautern, Kaiserslautern, Germany
e-mail: haerder@cs.uni-kl.de

C. Sauer
e-mail: csauer@cs.uni-kl.de

The process of compiling an instance of a query programming model into the generalized MapReduce model is discussed in Sect. 4. As we shall see, the process is essentially the same for all query languages presented in Sect. 2. It depends only on an abstract definition of operators and their composition as well as on their classification into blocking or non-blocking.

In Sect. 5, we discuss the main aspects which impact the performance of query evaluation in MapReduce. These fall into two categories: (i) general query-processing optimizations, and (ii) distribution aspects of the MapReduce environment. Because the first category fits in the widely known domain of query optimization, we will focus on the second class of problems. We show how some of these problems can be solved within MapReduce, either by customizing the framework or by injecting knowledge about the data, while others are inherent to the computational model.

Finally, Sect. 6 draws our conclusive remarks, focusing on future work and related approaches.

2 Query Languages Reconsidered

The emergence of MapReduce alone does not justify the introduction of new languages for data processing. Because it is nothing but a generic infrastructure for data-intensive parallel computations, the choice of a higher-level language depends only on the targeted application scenarios. If we consider business-oriented data processing, which is the traditional use case of relational databases and OLAP/OLTP, SQL is perfectly suitable, and we could move directly to the discussion of compiling SQL into MapReduce.

The context of Big Data, however, is not only about technical requirements such as scalability, availability, or elasticity—it is mainly about different application scenarios than those for which scalable databases are designed. Given their relevance in current technology trends, we identify two major classes of applications. First, there is the new context of Web 2.0, in which the rate of data production reaches the petabyte-per-month scale and the technical requirements widely vary from those of relational databases and ACID transactions. Second, there is the context of scientific computing, in which the amounts of data being processed were traditionally small when compared to the computation and communication loads of typical tasks, but the increasing frequency of data-intensive loads narrows this gap, posing new architectural challenges.

In the case of scientific computing, the challenges of Big Data arrive in two opposite directions. From one side, we have tools for statistical computing and data analysis such as R and MATLAB, which are primarily designed for single-machine use and are thus incapable of scaling with ever-increasing amounts of data. On the other side, we have high-performance computing (HPC), which deals with scalable infrastructures, but whose focus lies primarily at computation-intensive tasks. Thus, HPC models lack the support for data-oriented workloads such as query processing.

Because the Big Data challenges have a deeper technology impact in HPC, the progress is obviously slower, and no significant approach for languages or programming models has been established yet. Therefore, our language analysis will focus on the Web 2.0 context, which in fact has more in common with database research and consequently receives more attention within the community.

2.1 Analyzed Languages

Our analysis will focus on four languages: PigLatin [15], HiveQL [18], Jaql [4], and XQuery [19]. We provide a qualitative analysis on how each of these languages overcomes deficiencies of SQL, based on a list of criteria which are relevant for query processing in Big Data environments. We refer to this list as our query language *wish list*. Because introducing each language is out of the scope of this paper, we assume that the reader is familiar with their basic syntax and data models.

Systems that implement these languages are beyond the scope of this discussion, since they basically rely on MapReduce as the execution engine. Furthermore, as we point out in Sect. 4, we can discuss the compilation and execution processes based on a dataflow graph model which is common to all the considered approaches.

We chose these four languages not only because there exist MapReduce-based implementations for them (for XQuery, see [17]), but also because they have basic characteristics of a query language: declarative, dataflow-oriented, targeted at factual databases, etc. Therefore, we do not discuss simple scripting languages for scientific computing, such as Sawzall [16].

As a kind of wish list, we illustrate the properties of such languages in the following.

2.2 Semi-declarative Nature

A common criticism to SQL is that, because of its declarative style, complex queries are often hard to express and understand. An SQL query is a monolithic entity, whose simplification into smaller parts is not very intuitive. For a query composed of SELECT, FROM, and WHERE clauses, for instance, the more advanced user understands that first the data is fetched from the tables as specified in the FROM clause, then it is filtered according to the WHERE predicate, and finally the columns of the result set are extracted as defined in the SELECT clause. For complex queries, this step-by-step division is crucial for human understanding, and the monolithic style of SQL can become a burden. Therefore, we believe that a *semi-declarative* style is better suited. Even though the "declarativeness" of a language is hard to measure, it is clear that expressing queries in a step-by-step style sits in between the abstraction levels of SQL and lower-level procedural access paths. Hence, we make use of the term *semi-declarative*.

Note that explicitly expressing the order in which operations are applied does not inhibit the query optimizer from reordering them to produce more efficient execution strategies. If anything, it introduces a decision criterion for when it is hard to choose an optimal strategy, namely the order of operations preferred by the user.

In the semi-declarative model, queries are composed in a manner which corresponds closer to that of the algebraic representation or logical plan. Consider, for example, the following Jaql query, which counts the number of persons over the age of 18 in each city:

```
read(hdfs("person.txt"))
-> filter $.age >= 18
-> group by c = $.city as grouped
   into { city: c, count: count(grouped) }
-> write(hdfs("output.txt"))
```

Note how the "`->`" syntax, which represents function composition, emphasizes the sequence of operations applied to the input data. After being read from a file, the records are fed into the `filter` operator, and then to the `group by` operator, and so on. This compositional characteristic is used in the algebraic foundations of query processing, as well as in the logical query plan representation.

The semi-declarative style is incorporated in PigLatin by means of variable assignments. The output of each operator is assigned to a variable, and these variables are then used as input to other operators. A query plan is then built by tracking the dependency of these variables. This variable-based composition is also available in Jaql, as an alternative to the function composition syntax, and it represents a very flexible and powerful way of enabling the user to compose arbitrary dataflow graphs of operators.

In XQuery, the semi-declarative style is present in FLWOR expressions, which are composed of a sequence of clauses which can occur multiple times and in any order. The semantics dictates that each clause in the sequence consumes a tuple stream from the previous clause, thus resembling a query plan. FLWOR expressions, however, have the same monolithic characteristic of SQL queries, because the clauses do not exist as an independent language construct, such as a function or an operator. Instead, a whole FLWOR block is treated as a single expression, and the flow of tuples from one clause to another is hidden in its internal semantics. This means that XQuery is less flexible and elegant than PigLatin and Jaql when it comes to composing queries as dataflow graphs.

We believe that the graph-oriented approach is more powerful, because the output of an operator can be fed into different processing streams. Such functionality, which resembles the UNIX command `tee`, is useful for large complex queries, which apply an initial transformation step (e.g., to clean the data), which multiple (not necessarily independent) computations rely on. The only restriction is that such graphs are acyclic (i.e., DAGs), because cycles introduce complex termination issues into the evaluation process.[1]

HiveQL, which is based on a relational model and is designed to be similar to SQL, suffers from the same drawbacks as SQL. Unlike XQuery, it does not emphasize the sequence of clauses and does not allow multiple clauses. Hence, it is a purely declarative language.

2.3 Nested Data Model

Several Big Data applications prefer to store data in a denormalized manner. From the technical perspective, there are well-known trade-offs from database literature in storing data in normalized vs. denormalized form, and such discussions are out of the scope of our study. From the perspective of data modeling, the decision depends only on the workload and data relationship patterns of a particular application. Hence, a query language should provide thorough support for both formats.

The problem which we identify with SQL is that, even though its latest versions support nesting through lists, tuples, and multisets, the language has poor support for operations on nested data. Furthermore, database engines are not designed to efficiently store and process nested data. This means that users are forced into choosing a normalized design, even when the denormalized approach would be more applicable. Therefore, we consider proper support for a fully nested data model and its transparent integration in the language's operations an important point in our wish list.

HiveQL follows the SQL approach. It deals with data stored in tables, whose column types can be atomic values as well as the typed data structures *list*, *map*, and *struct*. A list is an arbitrarily long sequence of values of the same type; a map is an associative array; and a struct is a fixed-length set of attribute names and values. These types are composable, which allows combinations like, for example, a list of maps of structs.

The drawbacks of HiveQL are essentially the same as in SQL, namely that there is no query support for data contained in these data structures—the only access method is

[1]Although cycles are necessary to model more "exotic" operations such as recursive queries.

through *get* operations. A further problem is that the structures used to store and manipulate data—tables or collections of tuples—are not available as types to the programmer. This characteristic is related to the over-declarative nature of queries, and it also has an impact on the wish for transparent UDF support, as we discuss later on.

One important characteristic, which is present in both PigLatin and Jaql, is that the types of data items and their collections are explicitly defined in the language and made available to the user. We refer to these types as the *element* type and the *bulk* type, respectively. In SQL and HiveQL, we have tables as the bulk type and tuples as the element type. It is not possible, however, to define the value of a column or a function argument as a tuple or table.[2] If the types were available in the data model, it means that an operator is nothing but a function on the bulk type, and this is a key characteristic that enables the flexible, graph-oriented, and semi-declarative style mentioned earlier. In PigLatin, a *bag* is used as bulk type and a *tuple* as element type. In Jaql, we have, correspondingly, *arrays* and *records*.

The XQuery data model is based on the idea that everything is a *sequence*. A sequence is an ordered, arbitrarily long list of *items*, and a single item has no distinction to a singleton sequence containing it. A sequence may not contain another sequence as one of its elements, but an item may be an XML *node*, which enables nesting by creating trees of nodes. The use of XML as basis for the data model has advantages and disadvantages. Despite being an elegant and general format for representing various shapes of user data, it is cumbersome from the programming perspective, because there is no notion of data structures or abstract data types. This means that there is no convenient way of implementing composite types like lists, dictionaries, matrices, and such. Instead, the only way to build composite nested structures is by creating a tree of XML elements. Even though it allows to simulate a wide range of data structures, the tree representation destroys any property of the data structure, restricting the access method to the much slower path navigation. Furthermore, XQuery hides its element and bulk types—tuples and tuple streams—as internal structures.

Nevertheless, the generic approach of modeling data using just trees has some advantages. First, because XML nodes have an identity, it is suitable for object-oriented programming and data management. Second, because of the recursive nature of trees and the composability of XQuery, there is no restriction on the kind of queries that may be executed on nested data.

[2]Newer revisions of SQL actually include the types MULTISET and TUPLE, but they are not used transparently as the type of stored tables and intermediate results.

2.4 Transparent UDF Support

Given the wider applicability of query processing in Big Data scenarios, it is crucial for a query language to allow a seamless integration of user-defined functionality. In Big Data Analytics, users must often integrate functions implemented in languages like R or MATLAB, in order to accommodate complex data analysis algorithms. One example in XQuery is the following query, which filters a collection of emails based on the *isSpam* UDF:

```
for $e in collection("emails")
where !isSpam($e)
return $e
```

Internally, *isSpam* probably applies a machine learning algorithm, which would be cumbersome to implement in XQuery itself. Hence, it is crucial for the query language to support extensibility by allowing the definition of external UDFs and treating them transparently like any other (built-in) function.

If the user prefers to have a more homogeneous environment by implementing all functions in the query language itself, the expressive power of the language should not be a limitation. This point is where SQL falls short, because it defines a separate language (SQL/PSM) for procedural code, which has a rather “exotic” syntax when compared to popular procedural and functional languages. This implies that the database system must provide two separate evaluation mechanisms: one for procedural code of UDFs, and one for dataflow graphs of queries. We believe this approach is not only redundant, but also prone to inefficiencies, because query optimization logic may not be directly applicable to procedural code. XQuery, in contrast, offers full-fledged support for recursive functions, and its compositionality allows FLWOR expressions to be used not only for bulky user-data processing, but also for simple iterative computations. This approach relies on the foundations of functional programming to unify the query and UDF environments, providing transparent integration in the query language as well as optimization potential to UDFs.

Jaql provides the same functionality by means of higher-order functions, but it goes one step further by exposing bulk and element types. Hence, UDFs can also be used as operators, because they are nothing but higher-order functions. The integration of user-defined operators and its impact on query optimization, however, is a complex problem which must be researched on its own. We refer to work in the *Stratosphere* project [10] for UDF optimization techniques in the context of dataflow models more general than MapReduce.

HiveQL and PigLatin have limited expressive power, because they support neither recursive nor higher-order functions, and also have limited procedural capabilities, because of the lack of constructs for control flow, such as looping and conditional statements. Therefore, they do not suffice as a general-purpose (but data-centric) programming language.

2.5 Partial Schema Definition

The definition of a schema allows data to be processed more intuitively, by referring to structures by name, as well as more efficiently and type-safely, by specifying types. Similar to other points in this wish list, SQL is too strict when it comes to schema definition, because it enforces the complete schema specification before enabling any processing on the data. The specification must be precise, which is a limitation for ad-hoc scenarios where the exact schema of a particular dataset cannot always be known in advance. For Big Data scenarios, which have a strong ad-hoc nature and often deal with self-describing data, it is crucial to allow data processing to start right away, freeing the user from the tedious task of defining schemas for data which is already available.

This does not mean, however, that the language should not support schema at all, because it has major advantages for the user. As in other points of this wish list, it is important to give the user the freedom to specify an amount of schema information which he judges convenient. Using a partial schema, the user is allowed to specify the basic structure of the data, representing unknown types with a wildcard such as the `*` symbol. One may wish to specify, for example, a table with four well-defined columns and one fifth column for miscellaneous data, which can occur in various shapes or which the user simply has no knowledge about. The task of the query processor in such cases is to take advantage of as much type information as possible to speed up query execution.

Jaql and XQuery are the only languages with support for partial schemas. In XQuery, the types of items are organized in a single hierarchy, in which `anyType` is the root. This allows unknown schema information to be specified using general types, based on type substitution. A schema can then be refined simply by specifying more specific types. Item types are then combined with a cardinality to build *sequence types*. Cardinalities can be *zero*, *one*, *zero or one*, *zero or many*, and *one or many*. Because every instance of the XQuery data model is a sequence, a schema in XQuery is simply a collection of sequence types. Furthermore, because it is based on XML Schema [20], it provides powerful rules to specify various shapes of documents in different levels of rigor.

Jaql has a much simpler approach, based on a pattern language similar to regular expressions. Atomic types are specified by their names, and composite structures are specified using a straight-forward syntax. For example, a record with `id` as its first attribute, a second optional `url` attribute, and arbitrary fields after that can be specified as `{id: string, url?: string, *}`.

The disadvantage of Jaql is that there are no user-defined types, which means that schema information is only a *constraint* on possible values of a domain, but it does not specify a domain on its own. Thus, it is not possible for users to specify types that occur multiple times within an application and refer to them by name. Furthermore, the schema facility of Jaql basically relies on dynamic typing, which has the downside that there is no type safety in the query language, which is in fact an expected feature when defining a schema.

HiveQL, like SQL, enforces the use of schema, while PigLatin uses an all-or-nothing approach—bags are either completely untyped or have a full SQL-like schema. One disadvantage of PigLatin towards HiveQL, though, is that data structures are untyped. Hence, it is not possible to specify a map from strings to integers, for instance—all that can be specified is a map. This is a rather primitive and inflexible approach when compared to the other languages.

2.6 Generality

Apart from XQuery, all the languages that we analyzed were developed with the primary goal of expressing MapReduce jobs at a higher level of abstraction. Despite also aiming, rather as a secondary goal, at being applied in more generic data processing scenarios, in practice they have found no use outside MapReduce processing. Because of such a MapReduce-driven development, features that support the use case of traditional database systems are not included. For example, there are no statements for inserting or updating stored data, no data definition sub-language, no transaction-related statements, etc. In essence, MapReduce query languages are designed specifically for large-scale batch processing, when in fact the same data model and semantics can be applied to wider data management scenarios. It is therefore a relevant point in our wish list to require generality, in order to support heterogeneous environments in a versatile manner.

3 A Generalized MapReduce Model

The programming model of MapReduce has become widely known, and hence we assume that the reader is familiar with its basic structure. Nevertheless, for the sake of completeness, we provide a quick overview of a parallel job execution, which is illustrated in Fig. 1.

The execution starts by exploiting data parallelism in the input dataset, dividing it into partitions called *splits*. Each split is processed independently by a *map task*, which is assigned to run in one of the *worker* nodes. A map task iterates over all *key-value pairs* of its input split, calling a user-specified *map* function on each pair to produce a list of output pairs. The pairs resulting from all map invocations are

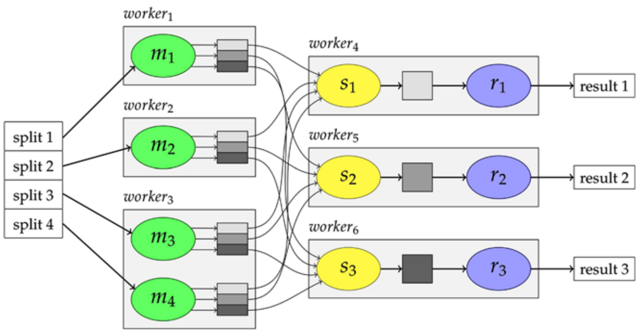

Fig. 1 Job execution in a MapReduce cluster

then concatenated, sorted, and divided into R partitions. In Fig. 1, we have $R = 3$, which is represented by three shades of gray coming out of the map tasks m_i. These partitioned outputs are stored in the local storage of the worker node which executed the map task.

For each of the R partitions, a *shuffle* task is scheduled for execution in a worker node. Its job is to fetch its corresponding partition (e.g., task s_2 fetches the second partition) from all the workers that completed the map tasks. The fetched key-value pairs are merged into a single sorted list and then grouped by key, which produces a list of keys associated to lists of values. This list serves as input for a following *reduce task*. Each reduce task works on a single partition. For this reason, it may execute in the same worker node of the corresponding shuffle task. Similar to map tasks, a reduce task executes a user-provided *reduce* function on each grouped key-value pair. The pairs produced by *reduce* invocations are finally concatenated to form the job's output, which is partitioned into R output sets.

The general structure of a MapReduce computation can be organized into three *phases*: Mapper, Shuffle, and Reducer. As noted in [13], the Mapper and Reducer phases are essentially equivalent, in which they simply provide the functionality of the well-known `map` function of functional programming languages—namely applying a function to each item of an input list. The only difference between them lies at how they deal with inputs and outputs: Mapper reads from splits and writes to partitioned lists; Reducer reads from local Shuffle output and writes to output lists on distributed storage. The first generalization we propose comes from abstracting the input and output aspects away from the actual phases. This allows us to model a MapReduce computation as a sequence of Mapper-Shuffle-Mapper phases, where the way in which each phase consumes input and produces output is specified separately.

This elimination of the distinction between Mapper and Reducer phases directly allows us to apply a further generalization: multi-stage jobs. The Shuffle phase essentially performs a global group-by-key operation. In case, the task to be executed involves a subsequence of two or more of such operations, it is necessary to execute multiple jobs. The problem with this approach is that, in concatenating the phases executed in a sequence of two jobs, we have a Mapper phase at the end of the first job (it replaces the Reducer phase in our first generalization) followed by another Mapper at the beginning of the second job. These two Mapper phases are obviously redundant, and, what is worse, it technically means that data is completely written to distributed storage, only to be read again in the following phase, in a blocking manner. This is clearly a waste of time and bandwidth. Therefore, we propose a generalization in which a single job is composed of n Mapper phases interleaved by $n - 1$ Shuffle phases.

A further generalization arrives if we look at what is executed sequentially inside a single map task. In order to implement the functional `map`, the task iterates over each key-value pair of its input, calls the user-provided map function[3] on each pair, and concatenates the resulting pairs into a single output list. We may allow the user to provide the task code itself, which we refer to as *task function*, so that the processing of a whole split or partition is abstracted. In the query processing scenario, for example, this model enables us to apply the iterator model to the input list, allowing efficient pipelined evaluation. Given this generalization, the finest granule of parallel computation is a task, and we no longer speak of the original map and reduce functions.

The last generalization we propose allows multiple inputs at each phase, which is needed to implement binary operations such as joins. The key observation here is that a Shuffle phase fetches scattered key-value pairs from the previous Mapper phase to produce global groups. This means that, as long as the keys belong to the same domain, it does not matter whether the key-value pairs were generated from a single input or from multiple inputs. This also implies that the tasks themselves may execute different functions, as long as the produced keys are in the same domain. Therefore, we allow Mapper phases to run on k inputs, where a task function is specified for each input. The Shuffle phase following a multiple-input Mapper may then either combine any subset of the k outputs produced, or execute on a single output as in the original model.

An instance of our fully generalized model is illustrated in Fig. 2, which depicts a multi-stage job composed of four Mapper phases $M_1, \ldots, M_4$. A phase is constituted of a series of task functions t_i, each of which will spawn multiple tasks when executed in a parallel setting. The linkage between these functions, which specifies the data flow, is defined by input and output specifications which are attached to each task function. We do not provide a precise specification of how such inputs and outputs can be defined, but

[3]Note that the MapReduce authors ambiguously refer to map as the first-order function that, in the functional programming setting, is actually a parameter to the higher-order function `map`.

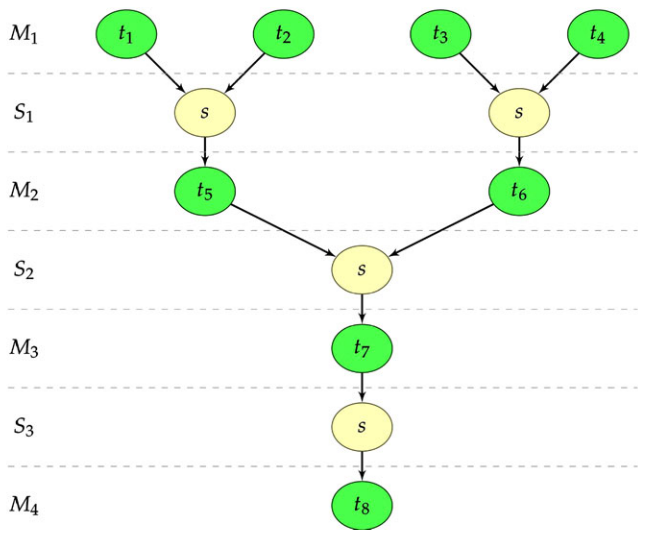

Fig. 2 Generalized phase-oriented model

it suffices to say that data may be read from or written to files on the distributed storage as well as on local disks, and that it should include options like sorting and partitioning, in order to properly simulate the original MapReduce model.

The job depicted in Fig. 2 corresponds to the same phase structure that would be generated for a bushy four-way join followed by a group-by operation, for example. In Sect. 4, we explain how such a job is generated from the actual query plan. Note that Shuffle phases always execute the same built-in function s, which is the generalized version supporting one or more Mapper inputs and producing a single grouped output. This function always precedes a user-provided task function in a Mapper phase. Hence, we could also illustrate a job in our generalized model as a graph of task functions t_i, omitting the implicit Shuffle phases.

Despite being a generalization of the original MapReduce model introduced in [5], our proposed model can be fully implemented in Hadoop. Task functions and multiple inputs are already supported, whereas the other generalizations, namely elimination of Reducer and multi-stage jobs, can be simulated by sequences of multiple jobs with empty, or identity, map/reduce functions. This simulation is obviously inefficient, but this is exactly the reason why the original MapReduce model is not well suited for executing complex queries. As we discuss in Sect. 5, there are alternative approaches where our model can be simulated efficiently.

4 Translation Mechanism

The generalized version of the MapReduce computational model, which we from now on refer to as GMR, serves as target of the compilation mechanism. Before explaining the actual compilation process, we establish an abstract model for the logical representation of a query—the source format of the compilation. Any of the four languages discussed so far can be fully expressed in terms of this model, which makes use only of general assumptions that are relevant for the parallel evaluation. As we established earlier, the popular MapReduce implementation Hadoop can be used to simulate the GMR model. Hence, we do not further distinguish the GMR model from the original MapReduce model when discussing the compilation process.

4.1 Query Plans

We assume that a query plan is a tree of operators. Later on, we discuss techniques to generalize this assumption into directed acyclic graphs. For now, the tree assumption simplifies the discussion. Operators can be interpreted as higher-order functions that consume and produce instances of a *bulk type* (e.g., array, bag, or table). The operators in the leaf nodes of the tree are special *scan* operators, which have the purpose of fetching elements from stored datasets. Similarly, the root is a *sink* operator, which delivers the result of the query into a file or console.

The bulk type, which is the input and output type of operators, contains instances of an *element type*, which are the individual items fetched by operators. No assumption is made as to what the structure of an element is. It may be an associative list, a fixed-length set of attributes and values, or a single complex item such as an XML node—from the MapReduce perspective, it is only important to identify keys and values, regardless of what the actual content of these keys and values are.

To ease the nomenclature in the following discussions, we assume, without loss of generality, that the bulk type is *array* and the element type is *tuple*.

4.2 Compilation into GMR Model

In the GMR model, we have the concept of a task function as the equivalent of a query operator. Like operators, task functions are also composed into a tree in order to form a job specification. Therefore, it becomes clear that the compilation process is based on a mapping between operators and task functions. Given this mapping scheme, the input/output specifications of the task functions, i.e., the edges in the graph, can be derived directly from the composition of operators.

The mapping is based on the classification of operators into blocking and non-blocking. The former are operators which allow pipelined execution—as supported by the iterator model (i.e., open-next-close)—by operating on one tuple at a time [8]. This characteristic enables a data-parallel execution, which is the key to mapping into MapReduce. In fact, one may argue that "non-blocking" and "data-parallel"

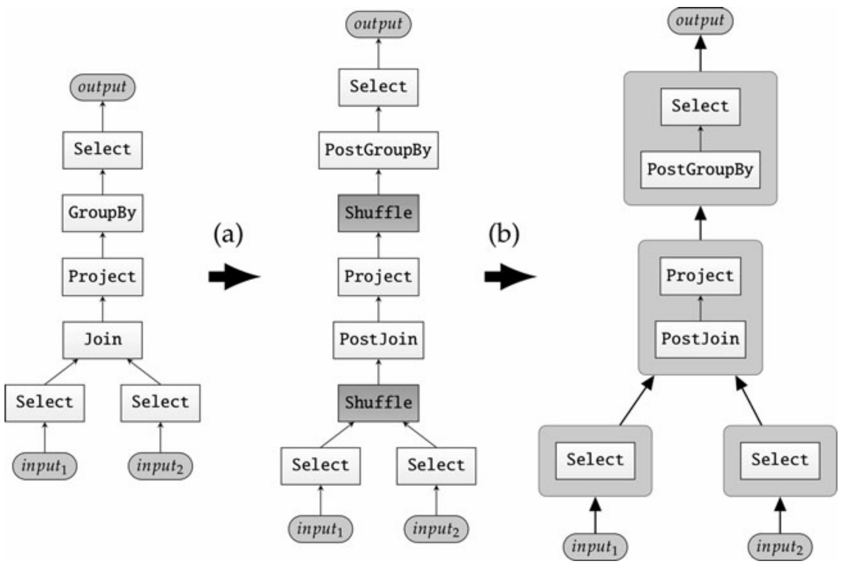

Fig. 3 Rewrite of query plan (**a**) and its translation into task functions (**b**)

are equivalent definitions, in the sense that they imply each other.

In the blocking category, we have operators for which the value of at least one output tuple depends on the value of more than one input tuple. Note that the common informal definition as an operator which buffers the complete input before producing any output is more strict than the one given. However, our definition simplifies the discussion, because it basically implies that we cannot generally assume that a data-parallel execution is possible. Based on this definition, a more formal definition for non-blocking operators can be derived as the converse statement.

In the GMR model, the task functions which are executed in Mapper phases are essentially non-blocking, because they only operate on independent partitions of the input. Hence, whatever operation is implemented by the user inside a task, it can only be of non-blocking nature. Note that the definition of blocking and non-blocking depends on the granule of the element type in consideration. In the GMR model, we have datasets which are composed of partitions, which, in turn, contain key-value pairs. If we consider the partition granule, task functions are non-blocking, because their input is a single partition. If we step down to the level of key-value pairs, then we may have *partition-wise* blocking operators, which are blocking with respect to pairs being processed by a single task, but non-blocking with respect to the whole phase input. The *combine* operation defined in the original MapReduce model [5] is an example of a partition-wise blocking operator, because it pre-aggregates the key-value pairs of a single partition to speed up a blocking global aggregation later on.

The shuffle function, which is executed by tasks of a Shuffle phase (function *s* in Fig. 2), is a blocking operation, because it implements a global sort and merge of the key-value pairs generated by the previous Mapper phase. It is the only blocking operation provided in GMR, and because, contrary to a task function, it is fixed by the framework, other blocking operations must be somehow simulated by a shuffle. Therefore, the first step in the translation process is to rewrite the query plan so that there is only one kind of blocking operator, namely one that is equivalent to a shuffle. In our query model, we make use of a Shuffle operator to simulate the functionality of the GMR shuffle function.

The strategy to convert blocking operators into a Shuffle is to make use of a post-processing non-blocking operator which, based on the sorted Shuffle output, rearranges the tuples to produce the correct result. This is only possible because a Shuffle essentially implements a multiple-input sort-merge, which is a basic algorithm that can be used to implement the Join, GroupBy, and Sort operators, for instance. Figure 3a illustrates how a query plan with Join and GroupBy operators is translated into sequences of Shuffle-PostJoin and Shuffle-PostGroupBy, respectively.

Based on the rewritten query plan, we specify the mapping between operators and task functions. The Shuffle operator, which is the only blocking operator in the rewritten plan, is trivially mapped into a shuffle function. Non-blocking operators, on the other hand, have the property that their composition is also non-blocking. Hence, we can map a sequence of adjacent non-blocking operators into a single task function. If we omit the Shuffle phase from the graphical representation and interpret the boxes on the right-hand side of Fig. 3b as task functions, the resulting graph can be interpreted as an instance of the GMR model.

Technically, this idea of "packing" a sequence of non-blocking operators into a task function has the interesting property that the iterator model may be used to evaluate them. In fact, we may execute these "partial pipelines" using a standard database query execution engine, resulting in a very efficient evaluation at the task level. This is exactly why it pays off to abstract away from map/reduce functions and consider tasks as the finest computation granule.

4.3 Implementation of Common Blocking Operators

The basic operator to which all blocking operators must be converted to is Shuffle. Figure 4 illustrates, at the bottom, a Shuffle on two arrays of tuples. Here, we assume that tuples are records with atomic values, as in the relational model. The value of the first column of each input is used as grouping key. In our general model, we assume that a function *extractKeys* is provided along with each task function that precedes a Shuffle phase. This function takes a tuple as input and delivers a sequence of atomic values, which are used as keys in the GMR key-value pairs. In the example, *extractKeys* would be, for both inputs, a projection of the first column.

Note that the output is not grouped, as normally assumed in the original MapReduce model. In a distributed setting, the Shuffle operation performs a global rearrangement of the key-value pairs by merging, sorting, and repartitioning. The grouping operation is trivial to perform in the sorted output, and, because it is not necessary for joining and sorting, we abstract it from the task function.

The Sort operator is equivalent to a Shuffle with a single input. Hence, no post-processing operators are needed. A GroupBy is implemented by a single-input Shuffle followed by a PostGroupBy, which simply groups adjacent tuples that have the same key. This behavior corresponds to a sort-based implementation of GroupBy. Note that, in this case, it is assumed that the language supports a nested data model, which means that groups can be formed within a tuple without any implied aggregation. This is in contrast to the relational model, where grouping is always attached to an aggregation specification for each non-grouped column.

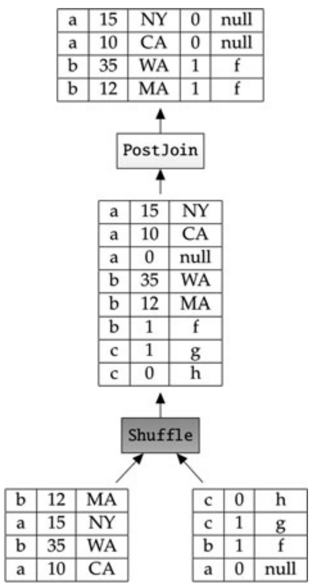

Fig. 4 Implementation of Join using Shuffle and PostJoin

To implement Join, a Shuffle of two inputs is used. This basically implements a kind of sort-merge join algorithm, where the two inputs are "shuffled" together into a single one before performing the actual join. The PostJoin operator then buffers all input tuples with the same join key, i.e., all adjacent tuples that deliver the same value for *extractKeys*. These buffered tuples must then be separated according to which input they come from and finally combined with a Cartesian product. This behavior is illustrated in Fig. 4. Note that the number of matches for a single join argument value is expected to be small, so the PostJoin computation can usually be carried out in main memory.

The separation of tuples in PostJoin is based on a *tag* value which must be attached to the key-value pairs before entering the Shuffle phase. This technique, referred to as *reduce-side join* [21], may be implemented either by attaching the tag to the tuple or by incorporating it into *extractKeys*. To keep the discussion at an abstract level, we assume the existence of a function called *extractTag*, which identifies whether a given tuple comes from the left or right input. Note that the PostJoin execution can be optimized by using the tag as a secondary sort key, which means that only tuples from one input must be buffered.

4.4 Query Plans as DAGs

So far, we have only considered operators which produce a single output, which results in tree-shaped plans. Multiple outputs, that result in DAG-shaped plans, can be implemented using a Split operator, which can be explicitly generated from demultiplexing commands in the query language (such as Jaql's *tee*), or automatically by identifying common sub-expressions in the query optimizer. In a traditional query evaluation mechanism based on the iterator model, a Split operator introduces buffering concerns when output branches consume tuples at different rates or points in time. In the GMR scenario, however, the implementation of a Split depends on whether or not blocking operators are in the output branches.

For the output branches of a Split which contain only non-blocking operators, the evaluation is carried out using the task-level query evaluation mechanism, which has to deal with the buffering issues mentioned above. Because it stays at the task level, this kind of splitting does not involve GMR shuffles, and hence the job graph will be tree-shaped. This mechanism also works if the non-blocking branches are all combined together later on by means of a join. In this case, the (de)multiplexing logic is again hidden from the GMR model by executing inside task functions.

In the case of blocking operators, we rely on the synchronous execution of Mapper and Shuffle phases. Because it is a blocking operation, a Shuffle cannot start until all

Mapper tasks have finished.[4] Therefore, there are no buffering issues when demultiplexing, and the Split operator can be implemented by multiple task functions whose input specifications are based on the same source. The simulation of this behavior in Hadoop may require the use of identity map and reduce functions, which, just like in multi-stage jobs, introduce a performance bottleneck. We give an overview of such issues in the following section.

For an overview of how the Split operator is implemented in the Pig query engine, we refer to [7].

5 Performance Considerations

5.1 Query Optimization

Because the starting point for our compilation mechanism is the query plan, we assume that standard query optimization techniques such as predicate push-down, join reordering, projection, etc., have already been applied when being submitted for translation. Nevertheless, there are issues to be considered when dealing with the evaluation in MapReduce. We give a brief overview of some of these issues in the following discussion.

The first issue concerns the provisioning of statistical information for cost-based decisions. Because the MapReduce environment does not assume exclusive control of the data, it is not possible to keep and maintain statistics as it is done in database systems. On top of that, the arbitrary complexity of nested data, combined with the potential lack of full schema information makes it difficult to select the domains on which statistical information is collected. A simple approach to solve these problems, which is only viable in batch-oriented scenarios like MapReduce, is to run a preliminary sampling job on a small (but hopefully representative enough) subset of the complete dataset. Such a sampling technique provides us with estimations not only for cardinalities of input datasets, but also for the selectivity and running time of each operator.

An alternative to sampling arrives when we consider the synchronous execution of phases in MapReduce. It enables the query plan to be optimized on-the-fly. As each phase completes, it may report its statistics back to the compiler, which may then reorganize the plan sections in the following phases. As an example of application, this techniques provides an elegant solution for implementing join reordering. Regardless of the shape of the join tree (left-deep, bushy, or right-deep), a task function is eventually executed for each join input. Therefore, in an n-way join, we may execute the n task functions independently. Then, based on the cardinalities collected, the optimal shape of the join tree may be derived. Join optimization in MapReduce has been discussed in several publications [1, 14, 22].

If we step down into the MapReduce (or GMR) level, cost-based optimization techniques can be applied to find optimal values of distribution-related parameters such as the number of tasks and partitions. Such techniques are introduced in [9], which also provides relevant related work on MapReduce optimization. Additional approaches, such as Manimal [12] and Hadoop++ [6], try to optimize data access on raw MapReduce programs (i.e., on the user-provided map and reduce functions instead of on a query plan). Optimization of dataflow operator graphs and MapReduce programs is a topic which is currently under heavy research, and an investigation of these techniques is out of the scope of this paper.

5.2 GMR Simulation in Hadoop

The GMR model proposed in Sect. 3 can be implemented in Hadoop, but, as we already eluded to, there are significant performance bottlenecks in such a less general model. The first, and perhaps most critical problem is the implementation of plans containing multiple blocking operators. Because the framework does not allow multi-stage jobs, we must simulate it using multiple jobs where all but the first have empty Mapper phases. This implies that there are, for each but the first phase, additional costs from reading and writing the complete intermediate result one additional time. This superfluous read-and-write operation is performed by the empty tasks. Thus, the only solution for this problem is the multi-stage generalization.

A further problem of MapReduce, which is also reflected in our GMR model, concerns its inflexibility to implement blocking operators. The translation into a Shuffle operator requires cumbersome post-processing steps, especially in case of the Join operator. Furthermore, it does not enable the implementation of hash-based algorithms, restricting all operators to sort-based strategies.

More general computational models based on parallel dataflow graphs, such as Dryad [11] and Nephele [3], are better suited to implement multi-stage jobs and efficient blocking operators. These models not only provide higher flexibility of building dataflow graphs—beyond the phase-based approach of GMR—, but also offer a richer set of input/output specification patterns to connect the nodes in the graph. The flexibility to build arbitrary graphs with different edge connection patterns enables the implementation of a wide range of blocking operators, whereas in MapReduce there is only Shuffle. This allows the implementation of more efficient join operators, which come close to the performance of parallel database systems.

[4]Note that if we consider the implementation of Shuffle as a two-phase step—first sorting each map task output locally and then globally merging all partitions—, the first phase can start before all map tasks are completed. Nevertheless, because the merge phase still needs to wait, the whole process is itself synchronous.

6 Conclusion and Outlook

This paper provided an introduction to abstract query compilation techniques for the MapReduce computational model. The presented wish list of query language capabilities gives an overview of the expressive power required by Big Data applications. To fulfill the wish list, we need data processing languages with more expressive power and flexibility. We introduced four competing languages which attempt to fill these needs, but as our qualitative analysis shows, none of them is fully applicable, and there is still room for improved proposals. Based on our evaluation criteria, we believe that Jaql is currently the best suited data processing language for Big Data.

The efficient support for the required capabilities introduces new challenges for query optimization and execution. Such languages better suited for Big Data Analytics significantly push the boundaries of the well-organized, closed world of relational data. Based on the versatile and extensible query evaluation infrastructure of the Brackit project [2], we have implemented a prototype for the XQuery language. It currently provides comparable performance to the systems which implement the other three evaluation languages [17]. More comprehensive measurements, including parallel database systems and general parallel dataflow systems such as Dryad [11] and Nephele [3], would be a valuable contribution to the field.

On the MapReduce side, our generalized specification aims to provide a proper level abstraction to discuss the mapping of query plans. A direct implementation of our GMR model allows a simplified compilation process, because there is no need to introduce Hadoop idiosyncrasies such as empty tasks. On the other hand, our model does not generalize too much, since it maintains the main characteristics of the MapReduce programming model and can be fully simulated by Hadoop. Therefore, we believe that we achieved a proper compromise between the simplicity of MapReduce and the power and flexibility of general parallel dataflow models.

References

1. Afrati FN, Ullman JD (2011) Optimizing multiway joins in a map-reduce environment. IEEE Trans Knowl Data Eng 23(9):1282–1298
2. Bächle S (2012) Separating key concerns in query processing—set orientation, physical data independence, and parallelism. PhD thesis, University of Kaiserslautern, Germany
3. Battré D, Ewen S, Hueske F, Kao O, Markl V, Warneke D (2010) Nephele/PACTs: a programming model and execution framework for web-scale analytical processing. In: SoCC, pp 119–130
4. Beyer KS, Ercegovac V, Gemulla R, Balmin A, Eltabakh MY, Kanne CC, Özcan F, Shekita EJ (2011) Jaql: a scripting language for large-scale semistructured data analysis. Proc VLDB Endow 4(12):1272–1283
5. Dean J, Ghemawat S (2010) MapReduce: a flexible data processing tool. Commun ACM 53(1):72–77
6. Dittrich J, Quiané-Ruiz JA, Jindal A, Kargin Y, Setty V, Schad J (2010) Hadoop++: making a yellow elephant run like a cheetah (without it even noticing). Proc VLDB Endow 3(1):518–529
7. Gates A, Natkovich O, Chopra S, Kamath P, Narayanam S, Olston C, Reed B, Srinivasan S, Srivastava U (2009) Building a high-level dataflow system on top of MapReduce: the pig experience. Proc VLDB Endow 2(2):1414–1425
8. Graefe G (1993) Query evaluation techniques for large databases. ACM Comput Surv 25(2):73–170
9. Herodotou H, Babu S (2011) Profiling, what-if analysis, and cost-based optimization of MapReduce programs. Proc VLDB Endow 4(11):1111–1122
10. Hueske F, Peters M, Sax M, Rheinländer A, Bergmann R, Krettek A, Tzoumas K (2012) Opening the black boxes in data flow optimization. Proc VLDB Endow 5(11):1256–1267
11. Isard M, Budiu M, Yu Y, Birrell A, Fetterly D (2007) Dryad: distributed data-parallel programs from sequential building blocks. In: EuroSys, pp 59–72
12. Jahani E, Cafarella MJ, Ré C (2011) Automatic optimization for MapReduce programs. Proc VLDB Endow 4(6):385–396
13. Lämmel R (2008) Google's MapReduce programming mModel—revisited. Sci Comput Program 70(1):1–30
14. Okcan A, Riedewald M (2011) Processing theta-joins using MapReduce. In: SIGMOD conference, pp 949–960
15. Olston C, Reed B, Srivastava U, Kumar R, Tomkins A (2008) Pig Latin: a not-so-foreign language for data processing. In: SIGMOD conference, pp 1099–1110
16. Pike R, Dorward S, Griesemer R, Quinlan S (2005) Interpreting the data: parallel analysis with Sawzall. Sci Program 13(4):277–298
17. Sauer C, Bächle S, Härder T (2012) Versatile query processing in the MapReduce framework based on XQuery (submitted)
18. Thusoo A, Sarma JS, Jain N, Shao Z, Chakka P, Zhang N, Anthony S, Liu H, Murthy R (2010) Hive—a petabyte scale data warehouse using Hadoop. In: ICDE conference, pp 996–1005
19. W3C (2011) XQuery 3.0: an XML query language. http://www.w3.org/TR/xquery-30/
20. W3C (2011) XQuery and XPath data model 3.0. http://www.w3.org/TR/xmlschema-11-1/
21. White T (2011) Hadoop—the definitive guide: storage and analysis at Internet scale, 2nd edn. O'Reilly, Sebastopol
22. Zhang X, Chen L, Wang M (2012) Efficient multi-way theta-join processing using MapReduce. Proc VLDB Endow 5(11):1184–1195