XQuery Processing in the MapReduce Framework

by

Caetano Sauer

in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science

Supervisors:
Dipl.-Inf. Sebastian Bächle
Prof. Dr.-Ing. Dr. h. c. Theo Härder

Submitted to:
Department of Computer Science
University of Kaiserslautern

June 6, 2012

Abstract

This master's thesis addresses the mapping of declarative query processing and computation to the programming model of MapReduce. MapReduce is a popular framework for large-scale data analysis on cluster architectures. Its popularity arises from the fact that it provides distribution transparency, which is based on the computation patterns of functional list processing. This work proposes a translation mechanism which maps XQuery expressions to MapReduce programs, allowing the transparent, parallel execution of data-processing queries in a distributed environment. Our approach reuses the compilation logic of an existing XQuery processor and extends it with the MapReduce programming model and distribution awareness. This thesis describes the computational model of MapReduce and introduces the major aspects of data processing in XQuery, including an overview of the compiler organization. The mapping mechanism is described in three major steps: the transformation of the query logical plan into a distribution-aware representation; the compilation of this representation into MapReduce job descriptions; and their execution in the distributed framework. Furthermore, this work analyzes the performance of the approach in terms of execution time and provides a critical analysis with respect to existing approaches.

Zusammenfassung

Diese Masterarbeit beschäftigt sich mit der Abbildung von deklarativen Anfragen und Berechnungen auf das Programmiermodell von MapReduce. MapReduce ist ein beliebtes Framework für die Analyse großer Datenmengen in Cluster-Architekturen. Seine Beliebtheit ergibt sich aus seiner Verteilungstransparenz, welche auf den Prinzipien der funktionalen Verarbeitung von Listen basiert. In dieser Arbeit wird ein Übersetzungsmechanismus vorgestellt, der XQuery-Ausdrücke auf MapReduce-Programme abbildet, um Anfragen transparent in einer parallelen, verteilten Umgebung auszuführen. Der vorgestellte Ansatz erweitert dazu einen bereits existierenden Anfrageprozessor um das MapReduce-Programmiermodell. Die Arbeit beschreibt das Berechnungsmodell von MapReduce, führt die wichtigsten Aspekte der Datenverarbeitung in XQuery ein, und erläutert die interne Arbeitsweise des Compilers. Der Abbildungsmechanismus wird in drei Schritten beschrieben: die Transformation des logischen Anfrageplans in eine verteilungsfähige Darstellung; die Kompilierung dieser Darstellung in MapReduce in Jobs; und deren Ausführung im verteilten Framework. Abschließend, wird der Ansatz in Bezug auf seine Leistungsfähigkeit evaluiert und mit Blick auf bereits existierende Ansätze kritisch untersucht.

Ich versichere hiermit, dass ich die vorliegende Masterarbeit mit dem Thema "XQuery Processing in the MapReduce Framework" selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Die Stellen, die anderen Werken dem Wortlaut oder dem Sinn nach entnommen wurden, habe ich durch die Angabe der Quelle, auch der benutzten Sekundärliteratur, als Entlehnung kenntlich gemacht.

Kaiserslautern, den 05.06.2012

Caetano Sauer

in memory of my father, Iberê Goulart Sauer

Acknowledgements

I am deeply thankful to Professor Theo Härder for providing me with funding and excellent conditions to study and work, as well as constant support in educational and research matters. He gave me the freedom to pursue my study goals and work on interesting research projects in his group, and it is only because of his support to the Brazilian exchange program that I had the opportunity to study in one of the top universities in Germany. My gratitude also goes to Sebastian Bächle for his constant dedication in supervising my work and for teaching me a lot in all the interesting discussions we had. His magnificent work on the Brackit project was fundamental for this thesis, and it has been a great pleasure working together with him. Furthermore, I am thankful to Karsten Schmidt and José de Aguiar Moraes Filho, who guided me through my first two years doing research in Kaiserslautern. I also thank Professor Stefan Deßloch for his support in acknowledging my previous studies for a Bachelor degree and his guidance during the Master's course of studies.

On the personal side, I am foremost thankful to Irina for her love and care during these four years in Kaiserslautern. It is very hard to live away from home, family, and friends, but her company turned it into a happy and fulfilling experience. I thank my brother Jacob and my uncle Milton for always being sources of support and inspiration, especially after I lost my father. I am grateful to my mother, Maria Cristina, for supporting and understanding my endeavors abroad, with constant care and unconditional love. Finally, I am very thankful to all the great friends I made in Kaiserslautern, with whom I had great fun, exchanged many interesting ideas, and shared both the hard and the joyful moments. I hope we can keep in touch for many years to come.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

Recent trends in data management research have, to a significant extent, focused on the challenges of *big data*. What "big" means in this context is not only a matter of interpretation, but also heavily depends on the technological context. A database of 1GB, for instance, would have been considered big by any standards back in 1970, but in 2012 it easily fits in a fingernail-sized memory chip. In the search of a proper and timeless definition, Adam Jacobs [25] stated that big data is "*data whose size forces us to look beyond the tried-and-true methods that are prevalent at that time*". Thus, research on big data is concerned with the problem faced by data-management applications which cannot rely on established technologies simply because of their large size.

But where does such big data come from? In short, this question can be answered with a single word: *analysis*. A common trend in scientific communities and enterprises is that the key to gain insights, be they business decisions or scientific discoveries, is to collect as much data as possible and perform some kind of analysis on it [23]. Telescopes and particle accelerators produce data at a petabyte-per-year rate, in order to discover properties of sub-atomic particles and explore distant galaxies, while Web applications collect every mouse movement of millions of users to find out where ads should be placed. The phenomenon of big data arises from the fact that these data collection processes have a much higher throughput than what can be handled by the analysis stage of the pipeline.

Database systems have been developed for most business applications of big data, and they are commercialized by successful companies such as Vertica, Teradata, Greenplum, etc. However, the problem persists in many scenarios which are less business-oriented, including the fields of computational science and Web applications. There are two major factors that contribute to this. The first is related to the commercial aspect, since such specialized business solutions are often prohibitively expensive for the aforementioned scenarios. The second factor, and perhaps the most important one, is that many solutions are too restrictive for applications outside the business domain, because they are purely based on the relational model and do not allow for easy customization.

Because of the lack of cheap and flexible solutions for big-data analysis, the data management and processing layers of a system have to be developed from scratch. This phenomenon is fundamental to understand the context behind most technologies that arrive in the field of

big data, and we call it "*data management for programmers*". Developers of such systems need a lot of skill and creativity to integrate raw files and low-level IO libraries with scientific development environments such as R and MATLAB and code written in Java, Python, or C. The resulting systems are not only extremely complex and hard to maintain, but also likely to be inefficient, since they do not leverage advanced algorithms and techniques developed in databases. What we see, therefore, is a technological gap which needs to be filled by developing systems that implement common data management tasks without the restrictions of relational databases.

Because the most cost-effective way to scale storage capacity and processing power is by employing a cluster-computing architecture, i.e., a standard network of cheap off-the-shelf machines, it should be the key strategy to overcome the technical challenges of big data. Hence, the major step in filling the technological gap is to provide a middleware solution to abstract the aspects of data distribution and process communication. This is precisely where MapReduce comes into play. It is both a programming model and a software framework for the analysis of distributed datasets, which allows the developer to focus exclusively on the analytical algorithm, hiding the technical aspects of its execution in a cluster environment.

Despite being an important step towards solving the challenges in the analysis of big data, MapReduce still leaves a lot of room for improvement, not particularly in modifying the middleware itself, but mostly in developing additional tools that enhance the abstraction on different levels. This thesis investigates concepts and techniques for the abstraction in one of such levels, namely the language used to express MapReduce computations.

1.2 Contribution

In the past years, MapReduce has gained immense popularity in big-data research and applications, and the main reason for that is its simplicity. The framework seems to hit an "abstraction sweet-spot" in the scenario of data management for programmers: it alleviates developers from the tedious tasks of network programming and fault tolerance, but gives them full flexibility to specify their own data formats and implement algorithms in a programming language of their choice. There are, nevertheless, certain classes of MapReduce programs that exhibit a significant level of similarity, where the abstraction can be pushed even further. Our focus is on the large class of query-processing tasks, such as aggregating, sorting, filtering, transforming, joining, etc. Implementing such tasks in a general-purpose programming language is cumbersome and unintuitive, especially in imperative languages like Java and C. Hence, higher-level data-processing languages like SQL are an important step towards filling the technological gap we mentioned earlier, and we present arguments in favor of employing XQuery for that goal.

Our work analyzes the requirements and characteristics of query-processing tasks which are carried-out in MapReduce. A query is essentially a tree, or in some cases a directed acyclic graph, of operators, which can be modelled as higher-order functions. As it turns out, the MapReduce programming model is also based on higher-order functions, and therefore we specify a mapping mechanism which essentially translates the query programming model to that of MapReduce. For the execution of such queries in the MapReduce framework, we exploit an already existing query processing engine by extending it with distribution-aware op-

erators, providing an easy strategy to scale out a single-machine query processor to distributed datasets.

The remainder of this thesis is organized as follows. Chapter 2 introduces the computational model of MapReduce, focusing on the establishment of a dataflow-based representation to which a query plan can be translated. Chapter 3 briefly introduces the XQuery language and presents its benefits against traditional languages like SQL. Furthermore, we present the Brackit XQuery Engine, whose logical query representation is the basis of our mapping mechanism, and whose runtime environment is used to execute queries in the MapReduce context. Once MapReduce and XQuery have been introduced, Chapter 4 brings them together by describing a technique to translate XQuery expressions into the MapReduce representation and execute them transparently in the distributed environment. This chapter represents the core ideas of this thesis. Despite not considering performance aspects as part of our work, we provide in Chapter 5 a brief overview of some fundamental techniques which could optimize query execution in MapReduce. Furthermore, we perform simple measurements of query execution times to show the applicability of our solution. Finally, Chapter 6 concludes this thesis, providing a brief overview of the pros and cons of our approach, comparing XQuery to alternative languages, and discussing open challenges for future research.

Chapter 2

The MapReduce Model

2.1 Contextualization

In order to understand what MapReduce is about and most importantly for what kind of work it is suitable, it is important to study the context in which it was conceived and what goals its creators had in mind. The system was developed at Google, and as such its primary focus was to solve problems related to Web search, particularly the technical problems that arise when performing *large-scale analytical computations*. In practical terms, "large-scale" means distributed and scaled up to thousands of machines, and "analytical" means batch jobs that scan the complete dataset and perform some aggregation to extract meaningful information, typically orders of magnitude smaller than the original data.

The canonical example—and perhaps the "holy grail" task for which MapReduce was introduced—is the *PageRank* computation [13], which assigns a measure of relevance to a specific Web page when compared to a whole set of pages. The algorithm is an instance of sparse matrix-vector multiplication, a classical problem of parallel algorithms. The matrix represents the *Web graph* (a graph where vertices are Web pages and edges are hyperlinks between them), and hence the dimensions are in the order of tens of billions. This example is representative not only because of its large scale and analytical nature, but also because it widens the spectrum of data management problems beyond the traditional business scenarios.

Distributed systems that partition data and perform large computations in parallel were long established by the time Google came to existence, but the big-data phenomenon requires much higher levels of scalability. Traditional systems such as parallel databases were not able to manage data at this scale, since the maintenance of ACID properties in a distributed scenario quickly becomes a bottleneck. Such scalability is in fact rarely necessary, since in the usual OLTP/OLAP scenario the rates at which data is generated are much smaller.

The context of big data also brings more general problems than those of traditional data analytics or business intelligence. It is part of the recently emerged *data science* discipline [29], which considers the whole pipeline of collecting, preparing, analyzing, evaluating, and presenting data, incorporating aspects from data management, statistics, machine learning, data cleansing, information visualization, etc. Data science is a field inherently inter-disciplinary, and the key skills for its practitioners are creativity and versatility to deal with various raw data sources and transform them into relevant *data products*. MapReduce is a key technology

in applying data science, as the core component of data processing, and the main reasons for its adoption are scalability, simplicity, and versatility. Despite MapReduce being suitable for traditional data analysis, the more general context of data science should be considered when analyzing its characteristics and applicability.

There is also a technological context behind the design of MapReduce, as Google adopts a cluster computing architecture [6]. Systems that run on such architectures must minimize the amount of centralized control and information, take high latencies into account, and employ robust fault tolerance mechanisms. The latter is important because despite the mean time to failure of a single machine being high, it drops significantly when we consider a large collection of machines.

Given all these contexts that drove its development, MapReduce can be seen as a *middleware* for large-scale analytical computations, serving as a layer of abstraction between programs and their parallel execution. Like any middleware, it offers a programming abstraction and a computing infrastructure that provide distribution transparency, unburdening the developer from implementing common tasks such as communication, control flow, or serialization. The user provides only a pair of Map and Reduce functions, while the system takes care of the data flow through the network and the distributed control of execution[1]. Furthermore, it provides transparent fault tolerance, guaranteeing the completion of a task even in the presence of individual node failures.

The rest of this chapter introduces the MapReduce computational model, which is of greater relevance than the infrastructure to the purpose of this thesis. Certain aspects of the runtime system will be touched when necessary, but kept mainly at an abstract level.

2.2 Background: Functional List Processing

MapReduce was inspired by list processing in functional programming languages, which is a very elegant and powerful way of performing declarative data processing. The idea is to focus on the data flow rather than the control logic, representing computations as a composition of higher-order functions applied to lists of data items. These functions represent a general pattern of data processing, such as transforming or filtering. The specific behavior is determined by another function which is given as parameter (hence the higher-order nature). For example, a relational join can be seen as a higher-order function with three parameters: the two input relations (i.e., lists of tuples) and a binary predicate (i.e., a boolean function), which is applied to the input tuples to indicate whether they match the join criterion.

In order to introduce the higher-order functions that represent such processing patterns, we need to define lists. A list is an ordered sequence of elements denoted as $[a_1, \ldots, a_n]$, where $a_1, \ldots, a_n \in T$. Hence, all elements must have the same type T, and the type of the list itself is $[T]$. Relational tables, for instance, can be represented as lists of equal-size tuples whose items have atomic types like strings or integers, such as $[(Int, String, String)]$. As in mathematics,

[1]Such characteristics were already present in *TP-Monitors* [21], being referred to as control and communication independence. The user initiates a transaction from a terminal, and the distribution aspects are kept transparent. Unlike TP-Monitors, however, MapReduce does not provide data independence, because it is only concerned with the task execution, and not with the management of the stored data.

we use the parentheses to denote ordered tuples.

The most basic processing pattern is the `map` function. It takes a list $[a_1, \ldots, a_n]$ of type $[T]$ and a function $f : T \to S$ and produces the list $[f(a_1), \ldots, f(a_n)]$ of type $[S]$. The `map` function can be used to emulate procedural `for` loops that iterate over a list and modify each item independently. It is thus a *transformation* pattern. It can be used, for instance, for relational projection if we provide a function like $f : (A, B, C) \to (A, C)$.

Another simple but important function is `filter`, which of course provides a *filtering* pattern. Given a list $[a_1, \ldots, a_n]$ and a predicate P, it filters out the elements a_i which do not satisfy the predicate, delivering a list $[a_{(1)}, \ldots, a_{(m)}]$ where $P(a_{(i)}) = True$ and $m \leq n$. Many other processing patterns can be defined in this manner. The `sort` function, for instance, produces a list $[a_{(1)}, \ldots, a_{(n)}]$ with the property that $o(a_{(i)}) \geq o(a_{(i+1)})$, where $o : T \to Number$ is an *ordering* function which is given as parameter[2]. Similar definitions exist for `group`, `join`, `union`, etc.

A very powerful function, which is expressive enough to represent all the others, is `fold`. It comes in two flavors: `foldl` and `foldr`, representing the left and right directions of the folding process, respectively. Without loss of generality, we concentrate on the left version, `foldl`, which takes three arguments: a list $[a_1, \ldots, a_n]$ of T, a binary function $g : S \times T \to S$, and an initial value c of type S. Its semantics can be understood as an iterative process which initializes an *accumulator* `acc` with the value of c and then traverses the list, updating the accumulator at each step[3] with $\texttt{acc} \leftarrow g(\texttt{acc}, a_i)$, as illustrated in Figure 2.1.

function FOLDL($g, c, list$)
  $acc \leftarrow c$
  for $i \leftarrow 1$ **to** $size(list)$ **do**
    $acc \leftarrow g(acc, list[i])$
  end for
  return acc
end function

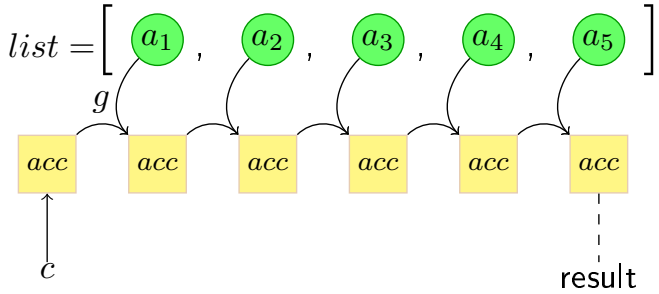


Figure 2.1: Iterative accumulation process that describes a `foldl` operation

As part of a functional programming environment, the actual definition of `fold` is in fact a general recursive pattern on the constructor of a datatype, and lists are thus only one instance of such datatypes. Nevertheless, the iterative semantics serves our purposes without going into much detail about recursion and datatypes. We refer to [10] for a proper functional treatment.

The sum of a list of numbers, for instance, can be implemented with a `foldl` with 0 (the neutral element of addition) as initial value and the $+$ operator as the g function. A more data-processing-oriented example is depicted in Figure 2.2, where `foldl` is used to implement the `group` function. Note that it simply groups successive elements which are equal, which means the list must be sorted in order to produce the behavior equivalent to an SQL `GROUP BY`. In this example, the function g takes a list of lists and an element, concatenating the element to the

[2]More general versions allow a *comparator* function as parameter, which generalizes the $\geq$ operator
[3]This is the reason why some textbooks prefer to call the function `accumulate`

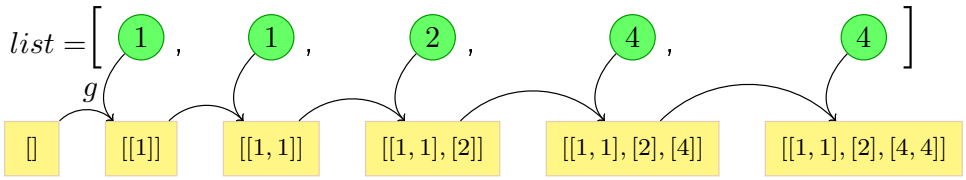


Figure 2.2: Using `fold` to implement the `group` function

last list if it is equal to one of its members, or appending a new singleton list with the element otherwise.

The pattern of *folding* may seem unusual, but all the other patterns can be expressed as a combination of folding patterns, as illustrated in the implementation of `group` above. Note that it must not necessarily condense the input list, producing a single value at the end, since the function g may as well produce lists as output. Nevertheless, the function is sometimes referred to as `reduce`, given its behavior of reducing the whole list to a single value.

These higher-order patterns are at the heart of any data processing system. Despite the purely declarative nature of SQL, for instance, the relational algebra is based on such patterns, which are referred to as *operators* like σ, π, and $\bowtie$, and these patterns are combined through composition, which imposes a strict order of application. The same ideas are present in XQuery FLWOR expressions, but in contrast to SQL, the order of application is defined explicitly by the programmer, which is why XQuery is sometimes referred to as a *semi-declarative* query language.

2.3 Programming Model

The original MapReduce paper [16] emphasizes the technical aspects and provides only an informal description of the programming model, which can in some points be misleading from a formal perspective. The Map and Reduce functions described on the paper, for instance, are *not* equivalent to the functional programming counterparts `map` and `reduce`. Instead, they are the functions which are written by the user and given to the framework for execution. Hence, they constitute the *parameters* of the corresponding higher-order patterns. Given this discrepancy, the distinct notation between `map` and Map must be emphasized, because despite having the same name, they have completely different semantics.

The programming model described in the original paper has been properly formalized in [28] by making use of the type system of Haskell, a popular functional programming language. The authors clarify the informal ambiguities of the original paper and provide an implementation of the MapReduce model as Haskell functions, which together with their type signatures, provide a semantical description of the framework.

The formalized description of the model, however, does not take practical aspects into account. The most popular implementation of MapReduce—the open-source project *Hadoop* [36]—allows the developer to customize and extend the behavior of the framework using object-oriented mechanisms such as subtyping and interfaces. The model implemented by

Hadoop, therefore, alleviates some of the strict assumptions of the formal model in [28].

Because our implementation is based on the Hadoop system, we provide a description of the programming model which makes a compromise between the Haskell-based formal specification and the one which is mostly used in practice. We start by providing a description of the Map and Reduce functions, which parameterize the execution of the higher-order processing patterns that we discuss later on.

Map and Reduce

The Map and Reduce functions which are provided by the user have the following signature:

$$\text{Map: } (K_1, V_1) \to [(K_2, V_2)]$$
$$\text{Reduce: } (K_2, [V_2]) \to [(K_3, V_3)]$$

Here K_i and V_i are, respectively, *key spaces* and *value spaces*, for $i = 1, 2, 3$. Hence, the Map function takes a key-value pair from one domain (or space) and produces a list of key-value pairs in another domain, performing a more general transformation than the parameter f of the `map` pattern, in the sense that a list is produced instead of a single value. The fact that Map produces a list allows the transformation to not produce any output by returning an empty list (which allows implementing a filtering pattern), as well as to produce multiple key-value pairs from a single one.

The Reduce function takes a key and a list of grouped values which belong to that same key. The grouping is performed implicitly by the framework, in a process known as *shuffle*, which will be discussed later. The input pair $(K_2, [V_2])$ is then transformed into a list of output pairs $[(K_3, V_3)]$. The Reduce function is thus used to perform a *transformation*, and hence it is equivalent to the behavior of Map, with the additional restriction that the domain of input values must be a list. We clarify this equivalence later when we discuss the higher-order patterns of MapReduce.

Despite Reduce being in fact a specialized version of Map, and therefore serving as parameter to a transformation pattern, there is usually a reduction process being carried out, because of the analytical nature of typical MapReduce jobs. Reduce is normally used to perform an aggregation, by transforming the list $[V_2]$ into a single value V_3, which corresponds to a condensed information such as a sum or an average. What happens in this case, is that a higher-order folding pattern is used as Reduce function[4]. Thus, there are three levels of nesting: a transformation pattern is parameterized with a Reduce function which is in fact another higher-order pattern, namely a folding parameterized by a sum or average function.

Consider the example of Figure 2.4: A Grouping of the sales records in Figure 2.3 by the state in which the store branch is located, followed by the computation of the average sale price in each state. The Map function must produce keys in the space of states, so $K_2 = \{NY, CA, WA, \dots\}$. The input key space K_1 is irrelevant here, so we use arbitrary values k_i,

[4]Note that given the higher-order nature of our functional setting, there is no distinction between higher order functions and simple (first-order) ones; the parameter f of `map` may be in fact another `map` or (as in the Reduce case) a `fold`.

and for the value spaces we have $V_1 = \{\text{tuples of the sales relation}\}$ and $V_2 = \mathit{Currency}$, which corresponds to the extracted `total_price` field from the sales tuple. The Reduce function then computes the average total price given a list of tuples which have the same state, and thus we have $K_2 = K_3$ and $V_3 = \mathit{Currency}$.

sale_id	total_price	state
4711	59.90$	NY
4713	142.99$	CA
4714	72.00$	NY
4715	108.75$	NY
4718	19.89$	WA
4719	36.60$	CA

Figure 2.3: Sample table of sales

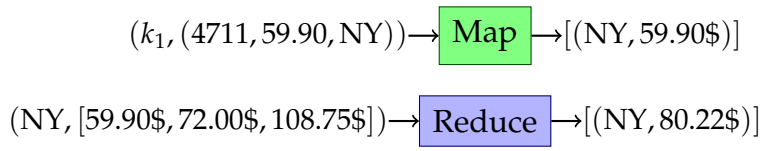


Figure 2.4: Map and Reduce functions used to calculate average sale price by state

Processing Patterns

So far, we have concentrated on the Map and Reduce functions, ignoring the higher-order patterns that execute them behind the scenes. Analogously to the patterns of functional list processing given in Section 2.2, we present the corresponding patterns used by the MapReduce framework, in the sequence in which they are applied during a computation.

The first step of the execution is the Mapper pattern, which invokes the user-provided Map function on each key-value pair of an input list $[(K_1, V_1)]$. Since each invocation itself produces a list $[(K_2, V_2)]$, they are all concatenated to form the complete output list of the Mapper. Again, we emphasize the misleading nomenclature introduced by the MapReduce paper [16]: Our Mapper pattern is analogous to the `map` higher-order function, while our Map is analogous to the function f which is parameter to `map`.

The list produced by the Mapper is then fed into the Shuffle pattern, which groups all pairs with the same key k into a single pair of the form $(k, [v_1, v_2, \dots])$, where $v_1, v_2, \dots$ are all the values associated to key k. The output is a list of the form $[(K_2, [V_2])]$ which is sorted in increasing order on K_2. The sort order is determined by a *comparator* function given as parameter, which specifies the total order between any pair of values from a specific datatype. Implementations like Hadoop provide built-in comparators for standard types such as strings and numbers.

Even though it happens behind the scenes, the sort and group processes that are executed in a Shuffle are crucial to typical data processing tasks such as joins and groupings.

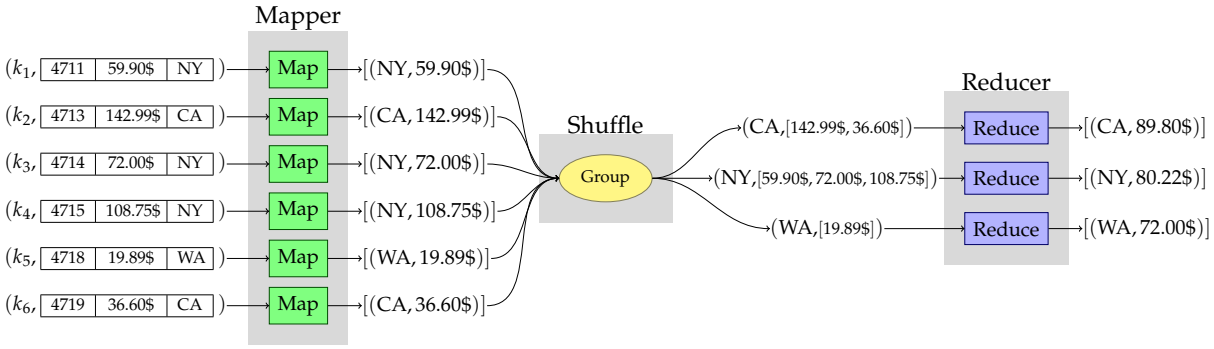


Figure 2.5: General structure of a MapReduce computation

The next step in a MapReduce computation is the Reducer pattern, which, as already discussed, is simply a Mapper pattern with the restriction that input values are lists. The Reduce function is invoked on each pair of the Shuffle output list, usually iterating over the list of V_2 values to produce an aggregate value in V_3. Some computations, which only require a transformation or a sort, may skip the Reducer pattern, producing directly $[(K_2, [V_2])]$ as result.

The composition of MapReduce patterns and the effect they have on input lists are illustrated in Figure 2.5. Again we use the example of computing average sale prices by state, using the same data as in Figure 2.3. Note how the Group function has a shape different than that of Map and Reduce. This emphasizes that they have different functionality: Group is a built-in function of the framework, which is applied over all input pairs at once, while Map and Reduce are user-specified and only applied to single key-value pairs.

2.4 Distributed Execution Model

The MapReduce computations introduced earlier have a high degree of *data-parallelism*, i.e., computations can be run on partitions of the dataset independently, but the model discussed so far does not consider any parallel execution strategy. Data-parallelism can be achieved by exploiting the transformation pattern of Mapper and Reducer, which allows each key-value pair to be processed independently, and that is exactly the strategy employed by the MapReduce framework.

This Section presents an execution model for the given programming model, which is tailored for cluster computing architectures [6]. The execution consists of applying the higher-order patterns Mapper, Shuffle, and Reducer in sequence, forming a composition of patterns. We refer to each application of a pattern in a distributed setting by the name *phase*.

Before discussing the parallelization strategy of each phase, we enhance the model introduced so far with aspects of the cluster architecture on which MapReduce operates. Individual computations such as applications of the Map and Reduce functions take place in *worker nodes*, while the control logic behind the composition of Mapper-Shuffle-Reducer is managed by a single *master node*[5]. Furthermore, data can be stored either in a *distributed filesystem*, whose

[5]In Hadoop terminology, we have *task trackers* and a *job tracker*, respectively.

files are available to all nodes, or in the *local filesystem* of each worker node. The contents of local filesystems are also available to all nodes, but the address of the node must be known (i.e., there is no location transparency).

Input data is stored in the distributed filesystem, in the form of *collections*, which correspond to the physical incarnation of lists. Collections are thus large files which contain elements of the same type, and their organization allows for the parallelization of the Mapper phase. Before job execution, the input collection is divided into M *splits* containing a subset (small enough to fit in a local filesystem) of the data, and each split is to be processed by a corresponding Mapper task.

The master node then dynamically assigns the M Mapper tasks to available workers, which read elements from the corresponding split sequentially, and invoke the Map function on each of them. As the Map function requires a key-value pair, there are two possibilities: Either the stored elements are already key-value pairs, or a key is assigned dynamically by the worker, such as the byte offset in the collection. The result of the Map invocations is then stored in a collection in the worker's local filesystem, and the master node is notified about the task completion.

Consider the example of Figure 2.6, which illustrates the execution of a job whose input collection was divided into four splits, each to be processed by a Mapper task $m_i, i = 1, \ldots, 4$. Note that two of these tasks were processed by the same node $worker_3$, and hence it has two intermediary result files in its local filesystem.

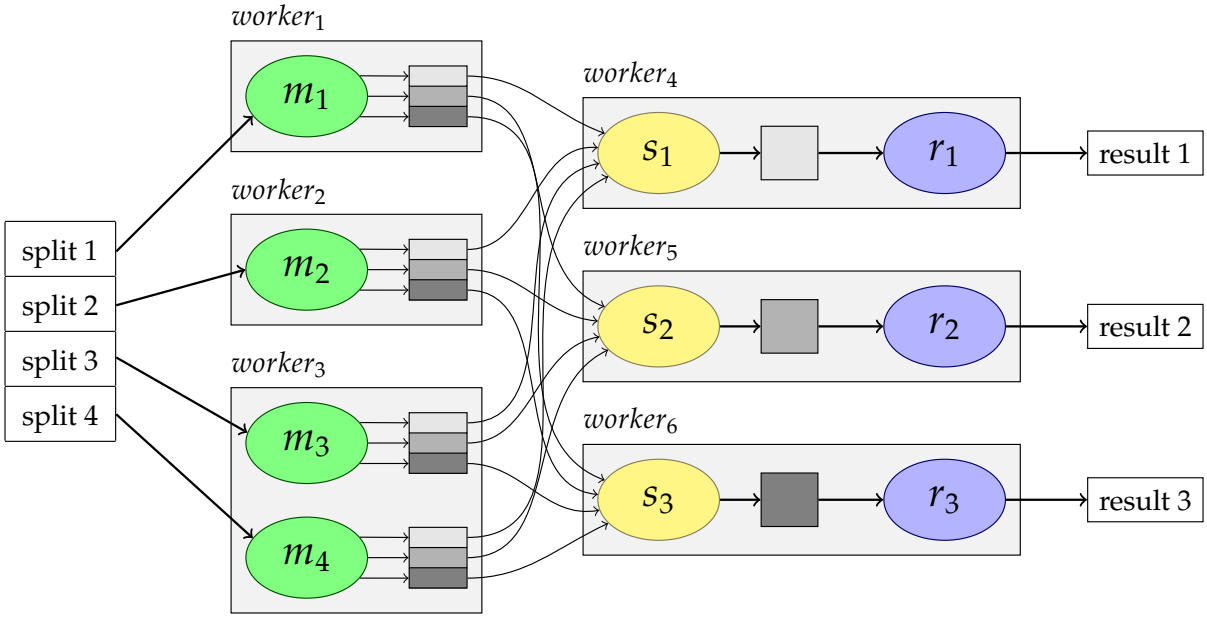


Figure 2.6: Distributed execution of a MapReduce job in a cluster

Once the complete input file is processed, the Mapper phase has finished, and its output is spread across M files in the local filesystems of the workers. The intermediary results are not stored in the distributed filesystem for efficiency and fault-tolerance reasons [16]. To allow for the parallelization of the Shuffle and Reducer phases, the key space K_2 is partitioned in R

subsets, and each Mapper worker must have partitioned its local output accordingly, allowing for a remote reader to fetch only the pairs that belong to a specific partition. In Figure 2.6, the key space was partitioned in three subsets, which are represented by different shades of gray.

Each partition of K_2 is to be processed by a Shuffle task s_i, and hence R Shuffle tasks are created. These tasks are again assigned dynamically to available workers, and their job is to fetch the pairs belonging to the task's partition from each of the M files that contain the output of the Mapper phase[6]. Once the data from all Mapper workers has been collected, the Shuffle workers start grouping by key and write the results to the local filesystem. Note that since the key space was a partitioned, all pairs that belong to any particular key are guaranteed to be processed by the same Shuffle worker.

In Figure 2.6, the three Shuffle tasks were assigned to three different workers. Note, however, that these workers may in fact be the same workers that executed Mapper tasks, so it may be possible that $worker_6$ is the same physical machine of $worker_2$. Here they are separated just to visually distinguish the phases.

After the Shuffle phase is completed, the same R workers that executed it also execute the Reducer phase. Since each partition of K_2 is completely contained in the corresponding worker node, there is no need to read data from remote workers, and hence the Reduce function can be directly invoked on each grouped key-value pair. This is the reason why a Shuffle task s_i and its corresponding Reducer task r_i are confined within the same worker node in Figure 2.6.

Results of the Reduce function are written to the distributed filesystem, in the form of R files, each corresponding to a Reducer task and thus to a partition of the key space K_2. Note that in contrast to the input collection, the results reside in different files of the distributed filesystem. In practical scenarios, this restriction is usually not an obstacle, since distributed applications can easily handle multiple files. Otherwise, the output files can be efficiently concatenated by the distributed filesystem.

The established parallel model is expressive enough to describe a translation mechanism that generates MapReduce jobs from typical database queries. Mapping to the MapReduce model is an alternative to implementing a distributed query processor from scratch, as normally done in parallel databases [17], allowing developers to easily scale-out existent single-threaded query processors, relying on the MapReduce middleware for the distribution aspects. This is exactly the contribution of this thesis: a distributed XQuery engine is achieved by scaling out using MapReduce. The next chapter introduces the XQuery language, focusing on the data-processing aspects, and describes the basic architecture of the Brackit engine, which served as basis for our work.

[6]Because the Shuffle task workers must each contact all the previous Mapper task workers, the pattern was named *Shuffle*, since it somehow resembles shuffling a deck of cards.

Chapter 3

The Brackit XQuery Engine

3.1 Why XQuery?

XQuery arose from a combination of efforts to create a query language for XML documents. Based on XPath and inspired by SQL, the goal was to shift the paradigm of query languages from the well-specified world of relational databases and their schemas to XML and its key characteristic: *flexibility*.

XML is an absolute success in both science and industry, and the only reason behind that is the flexibility it provides for the representation of data. It is a format that accepts data without schema, allows the representation of highly unstructured data, and fosters exchangeability through the use of a Unicode-based, self-contained, and human-readable encoding. These characteristics are the main background of XQuery—a language that couples with XML in providing flexibility to both data representation and processing.

Inheriting the flexibility features of XML, XQuery is a language that is able to process completely untyped data, which may be progressively enhanced with schema information, in a "data first, schema later" approach. XQuery can also cope with various degrees of structuredness in data, from normalized relational tables to arbitrary Web documents, without sacrificing the efficiency of highly structured, schema-supported data access.

Over the course of its development, however, XQuery was enhanced with features that bring it closer and closer to a universal programming language, tailored for data-oriented applications. The current XQuery specification [33] and its extensions bring a whole new level of flexibility into play, enabling the use of XQuery in much broader scenarios than simple data querying [5].

The expressive power of XQuery, however, is usually not acknowledged in scientific and technical communities, which are still attached to the querying roots of the language. This "curse" that lays upon advocates of XQuery is further harmed by the language's own name, which is not worthy of its true power: "X" comes from XML, but XQuery has an abstract data model [34] which is also suitable for data in many flavors such as relational, object, and graph; and "Query" is also too narrow to represent its expressive programming-language capabilities.

Nevertheless, the flexibility provided by XQuery is remarkable in the context of information management: it can be used not only in the database layer, as a replacement of SQL, but also in Enterprise Information Integration [11], in stream processing and sensor networks [12],

in data analytics [8], in the development and infrastructure of Web applications [26], in the presentation of content [18], and so on. Such high degree of flexibility also provides a kind of architectural independence, since code can be easily moved between tiers, like moving programming logic from application code to database stored procedures, for instance. The typical impedance mismatch problem, which takes place when relational data needs to be mapped to the data model of the application layer, is also completely absent in XQuery-based systems, since data is modeled in the XQuery data model across all layers.

Given such an expressive power, it is no wonder that XQuery is also perfectly suitable for expressing MapReduce computations. The following sections provide a brief introduction to the main constructs of the XQuery language, as well as a description of the Brackit XQuery engine and its internal computational model, including a proposed extension of the engine to support relational data.

3.2 A Quick Tour of The Language

This section provides a brief overview of the XQuery language, focusing primarily on FLWOR expressions, which form the basis for MapReduce computations. Rather than a detailed explanation, we provide a sample query whose individual components are discussed from an abstract perspective. For a proper introduction, we refer to the W3C XQuery standard [33].

Data Model

The XQuery language is defined in terms of the *XQuery and XPath Data Model* [34], or shortly, XDM. Every data item in XQuery can be treated as an ordered *sequence* which contains *items*, and there is no distinction between a sequence of length one and the individual item it contains. Note that this definition forbids nested sequences, i.e., sequences that contain another sequence as one of its items, which means that some operations must perform implicit unnesting. An item is either a *node* or an *atomic value* like integer or string. The types of nodes and atomic values are specified by XML Schema [35], which is the standard that defines types in an XML document.

A node can be one of seven kinds: document, element, attribute, text, namespace, processing instruction, or comment. A document is an abstract node (i.e., does not have a serialized representation), which represents an entire XML document and contains its root node as child. Elements are the basic node kind. They can contain other nodes as children, and hence are the ones that give structure to an XML document. Attributes are key-value pairs associated to elements, where a key consist of a *QName* (the XML-equivalent of an identifier) and a value is an instance of an atomic type. As the name indicates, text nodes contain plain character data, usually corresponding to the "user data" in a document. Text nodes do not necessarily represent string values, as the type is determined by schema information. If no schema is attached, text nodes represent instances of the special type `xs:untypedAtomic` [35]. We ignore the remaining node kinds as they are of smaller relevance for the purposes of *data-centric* XML, which corresponds to the "database" perspective of XML as opposed to the *document-centric* nature of office documents, Web pages, images, etc.

Figure 3.1 illustrates a basic XML document. Its equivalent structure as an XDM instance is shown in Figure 3.2, where each node shape in the tree corresponds to an XDM node kind. The document represents part of the catalog of a library or book store, and thus it contains a sequence of `book` elements.

```
<bib>
  <book year="1994">
      <title>TCP/IP Illustrated</title>
      <author><last>Stevens</last><first>W.</first></author>
      <publisher>Addison-Wesley</publisher>
      <price>65.95</price>
  </book>
  <book year="2000">
      <title>The C++ Programming Language</title>
      <author><last>Stroustrup</last><first>Bjarne</first></author>
      <publisher>Addison-Wesley</publisher>
      <price>77.95</price>
  </book>
  <book year="1992">
      <title>Advanced Programming in the UNIX Environment</title>
      <author><last>Stevens</last><first>W.</first></author>
      <publisher>Addison-Wesley</publisher>
      <price>65.95</price>
      </book>
  <book year="2000">
      <title>Data on the Web</title>
      <author><last>Abiteboul</last><first>Serge</first></author>
      <author><last>Buneman</last><first>Peter</first></author>
      <author><last>Suciu</last><first>Dan</first></author>
      <publisher>Morgan Kaufmann Publishers</publisher>
      <price>65.95</price>
  </book>
  <book year="1999">
      <title>The Economics of Technology and Content
             for Digital TV</title>
      <editor>
         <last>Gerbarg</last>
         <first>Darcy</first>
         <affiliation>CITI</affiliation>
      </editor>
      <publisher>Kluwer Academic Publishers</publisher>
      <price>129.95</price>
  </book>
</bib>
```

Figure 3.1: A sample XML document

FLWOR Expressions

The core of data processing in XQuery lies on *FLWOR expressions* (commonly pronounced "flower"). They consist of a sequence of *clauses*, which can be one of `for`, `let`, `where`, `order by`, or `return`. Note that the initials of the clauses form the acronym FLWOR. The version

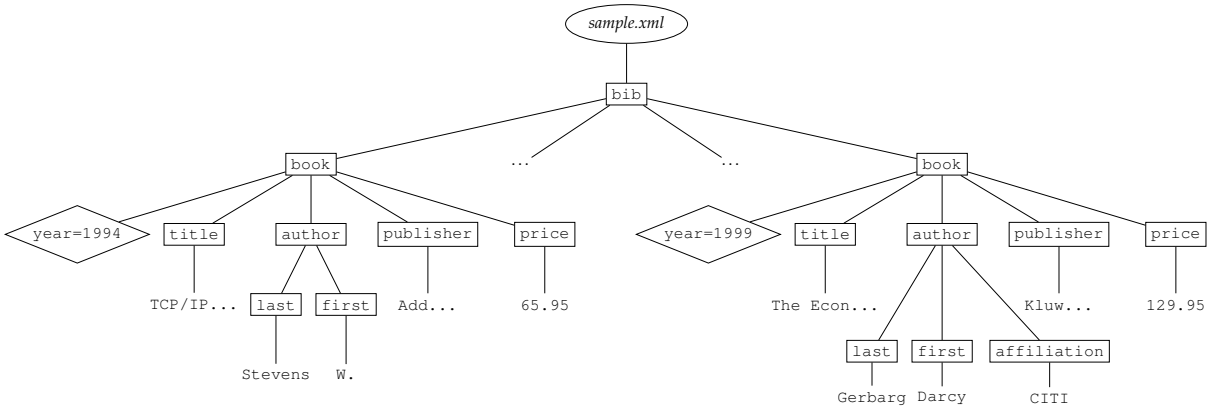


Figure 3.2: XDM instance for the sample document

3.0 of the XQuery standard further defines the `group by`, `window`, and `count` clauses. In order to form a valid FLWOR expression, the first clause must be either a `for` or `let`, and the last one must be a `return`. Furthermore, a `return` may not occur in any other position than the last. Figure 3.3 shows an example of FLWOR expression, which we use as basis for the following discussions.

```
for $b in doc('sample.xml')/bib/book
let $y := $b/@year
where $y > 1995
group by $y
order by $y
return
  <result>
    <year>{$y}</year>
    <avg-price>{avg($b/price)}</avg-price>
  </result>
```

Figure 3.3: Sample FLWOR expression

The semantics of FLWOR expressions is given by a sequential evaluation of the clauses, where each clause receives *tuples of bound variables* (which we refer to simply as *tuples*) as context and generates one or more tuples as context to the next clause. Variables are denoted by the initial symbol `$`. At the start of the evaluation there are no variables bound, hence an empty tuple is used as initial context. Because each clause receives and produces tuples which are passed to the following clause, the process can be thought of as a *tuple stream* being manipulated by a chain of clauses. This behavior is equivalent to tables (which can also be interpreted as streams of tuples) being manipulated by a chain of relational operators, and hence FLWOR expressions are analog to `SELECT` statements in SQL.

A `for` clause evaluates the expression given to its `in` parameter and, for each item in the sequence that results from the evaluation, the current tuple is extended with a new variable bound to that item. The resulting tuple is then fed to the next clause and the process is repeated

until all items in the sequence have been consumed. In the example of Figure 3.3, the variable `$b` is bound to each book in the library catalog, and the following clauses are evaluated once for every book.

The `let` clause serves the simple purpose of introducing a name to refer to a given expression, just like in functional programming languages. Hence, it simply attaches a new variable to the context tuple. If a sequence containing multiple items is returned by the evaluated expression, the whole sequence is bound at once, and thus there is no iteration.

After the `for` and `let` clauses of the example in Figure 3.3 are evaluated, we have the following tuple stream:

[$b = b_1, $y = "1994"]
[$b = b_2, $y = "2000"]
[$b = b_3, $y = "1992"]
[$b = b_4, $y = "2000"]
[$b = b_5, $y = "1999"]

Here $b_1, \ldots, b_5$ represent the book elements in the document as XDM instances. We emphasize that they do not contain strings with the XML text `<book>...</book>`, as all XQuery expressions work with XDM instances. The internal representation of b_i is implementation-defined.

The `where` clause filters out tuples for which the evaluation of the given expression returns the boolean value *False*. In our example, the following tuple stream remains after the `where` clause:

[$b = b_2, $y = "2000"]
[$b = b_4, $y = "2000"]
[$b = b_5, $y = "1999"]

The `group by` clause groups all tuples which have the same value on a given variable, condensing the other variables in a sequence. Unlike the other clauses, its argument must be a variable reference, and in order to group by an arbitrary expression, it must first be bound to a separate variable with a `let` clause, as it was done with `$y` in the example. A `group by` also supports multiple variable references, which causes groups to be formed according to distinct values on every given variable. The grouped tuple stream of our example is the following:

[$b = ($b_2$, b_4), $y = "2000"]
[$b = b_5, $y = "1999"]

The syntax `(`b_2, $b_4`)` represents a sequence containing the items b_2 and b_4.

The `order by` clause simply reorders the tuple stream according to the expression given. In our example, it will simply swap the two tuples in the stream, so that the tuple with year "1999" comes before the one with year "2000".

Finally, the `return` clause finishes the evaluation of the FLWOR expression and produces the final result by evaluating the given expression for each tuple in the stream. Because of the

```
<result>
  <year>1999</year>
  <avg-price>129.95</avg-price>
</result>
<result>
  <year>2000</year>
  <avg-price>71.95</avg-price>
</result>
```

Figure 3.4: Result of the example query

unnesting semantics of XQuery, if the expression results in multi-item sequences, these are all concatenated to form a single sequence which is the result of the whole FLWOR expression. In our example, the `return` clause is given a *constructor* expression, which we explain later. The final result is shown in Figure 3.4. Note that the two XDM instances that represent the `result` elements are shown serialized as XML strings, but they would remain internally as XDM instances if the result were to be given as input to another expression.

Additionally, XQuery provides the `count` clause, which binds a variable to an integer corresponding to the tuple position in the stream. It could be used, for example, to restrict the evaluation of the FLWOR expression only to the first n tuples in the stream, if used in combination with a `where` clause. Because the operation performed by `count` depends on a strictly sequential evaluation, and hence cannot be parallelized, we ignore it for the remainder of this thesis. Likewise, we ignore the `window` clause, whose semantics also have a certain dependence on the sequentiality of items. The parallelization of window patterns, such as moving averages, is a non-trivial task, which is out of the scope of this work.

Since XQuery is a functional language, everything is treated as an *expression*, which is in contrast to the *statement* nature of SQL. Because a FLWOR[1] is nothing but an expression (which can be evaluated to deliver a sequence of XDM items), note that the expressions used as argument to a clause may also be FLWORs. This behavior is equivalent to the use of *sub-queries* in SQL.

Note how the clauses of a FLWOR expression closely resemble the higher-order patterns of list processing discussed in Section 2.2. They can be seen as patterns whose inputs and outputs are tuples of variable bindings, except for `return`, which delivers a sequence as output. The functional parameters of these patterns correspond then to the expressions which belong to the clauses. An expression can be interpreted as a function on the variables to which it refers, as it is done in *lambda abstractions* of functional programming [10], and thus they are equivalent concepts.

[1]Given the often usage of FLWOR expressions throughout this thesis, we alternatively make use of the name FLWOR as a noun from now on.

Path Expressions

The predecessor of XQuery, which still consists of a sub-language defined entirely inside it, is XPath, a language whose purpose is to locate a subset of nodes within a document. It allows the user to specify a particular set of nodes of interest, such as all elements with a particular name, or all nodes whose value satisfies a given predicate. The syntax is similar to that of paths in a UNIX filesystem. The expression `doc("sample.xml")/bib/book` in the example of Figure 3.3, for instance, delivers all books in the bibliography document of Figure 3.1.

A path expression consists of a sequence of *steps*, each of which is evaluated according to a *context node*, producing a sequence which is then used as context to evaluate the following step. If the context consists of multiple items, the expression is evaluated once for each item, and the results are concatenated. The result of a path expression is defined as the result of evaluating its last step. Additionally, a step may contain a *predicate*, which filters the generated sequence before it is passed to the next step or delivered as result of the whole path expression.

In the scope of this thesis, we consider path expressions from an abstract perspective, simply as expressions which deliver a sequence of items. For a thorough description, see [33].

Constructors

XQuery is an extension of XPath in which it provides means to not only locate nodes in XML documents, but also to transform them, producing new nodes as result, as well as to create nodes which do not necessarily originate from existing ones. For that purpose, *constructor expressions* are defined.

The expression `<avg-price>{avg($b/price)}</avg-price>` of the example query, for instance, constructs an element named `avg-price`. It contains a sequence of nodes which results from the evaluation of the expression `avg($b/price)`. If the returned sequence would contain atomic values, these would be converted to text nodes. The constructed element is then used as content to the parent element constructor for `result`.

Similar to elements, there are constructors defined for each kind of XDM nodes. Furthermore, an alternative syntax is provided for defining names of nodes as XQuery expressions as well. Again we refer to [33] for a complete specification.

Atomization, Node Identity, and Document Order

In favor of a cleaner syntax, XQuery allows basic constructors such as arithmetic or boolean operators to work directly with any kind of sequence, without requiring the programmer to explicitly extract atomic values. This process is known as *atomization*, and its rules are defined as following:

- If the argument is a single atomic value, it is used directly.
- If the argument is a single node, the *typed value* of the node is extracted.
- If the argument is a sequence of multiple items, atomization is applied to each item, which results in sequence of atomic values.

The typed value of an attribute is directly its value, which is already of an atomic type. The typed value of an element, in the simple case where it only contains a single text node,

corresponds to the atomic value of that text node, as the value "65.95" under the element price of the first book in our example. In that case, when the element price is used inside a comparison or an arithmetic expression, it actually delivers the atomic value "65.95". As another example, the attribute @year is atomized in order to be compared with the literal "1995" in the where clause. For a full description of typed value extraction of all node kinds, refer to the standard [33].

Note that atomization does not necessarily result in a single atomic value, since multi-item sequences generate sequences of multiple atomic values. Thus, operations that require a single atomic value, such as arithmetic expressions, must not only atomize their arguments, but also check whether the atomized sequence contains at most one value.

Further concepts of XQuery, which require special care when implementing optimization rewrites in a query engine, are node identity and document order. Every XDM node possesses an implicit identity, which makes it distinct from any other node with the same name and contents. This means that constructor expressions cannot be freely moved around or replicated by the compiler, as they undergo an implicit identity assignment.

Document order is also an implicit characteristic of nodes which must be respected by the compiler. Nodes that belong to the same document or XML fragment have a certain order which must be kept by the FLWOR execution unless an explicit order was defined by an order by clause. This requirement may be alleviated if the user specifies the *unordered* property to the given query. Otherwise, the query processor must execute additional sorts, e.g., at the end of a path expression, to ensure preservation of document order, which despite being simple, can be an expensive computation depending on the size and distribution of the data. For that reason, we assume the unordered property in the discussions of Chapter 4.

3.3 Query Processor Architecture

A Versatile Query Engine

The Brackit XQuery Engine distinguishes itself from most available XQuery processors in which it provides a high degree of versatility. Typical XQuery solutions are offered either as part of a database system (native XML database or XML-relational hybrid) or as a standalone interpreter for small XML files. In the former case, the engine is tailored towards efficient processing of large amounts of external data, usually falling short on actual programming capabilities such as a standard library and function definitions. Moreover, it is usually part of a heavy and bulky system which is installed in a dedicated machine, and whose managed data is kept only in the internal storage, hence lacking the flexibility to perform simple tasks such as processing a small input encoded as a string. In contrast to this database scenario, interpreter solutions are tailored to the fast evaluation of parsed XML trees, usually employing stream techniques [27]. Therefore, they are lightweight and flexible enough to work with common XML documents in string representation, and are aimed at supporting a broad spectrum of the XQuery standard. However, given this distinct focus, such engines usually lack database capabilities such as efficient external storage, transaction processing, scalability, indexes, etc.

Brackit, on the other hand, uses a hybrid execution mechanism, which provides efficiency

comparable to the top performers in both usage scenarios [4]. The architecture of the system itself also fosters the versatility of usage, by employing a pluggable design, which allows (i) customization of the compiler for specific applications; (ii) incorporation of storage modules such as a native XML database; and (iii) interoperability with other languages by exposing the logical query representation. The versatility of Brackit was heavily exploited in the context of this thesis, since we extended the compiler for producing MapReduce jobs. Furthermore, as detailed in Section 3.5, we plugged-in a simple CSV-based relational store, which makes Brackit behave transparently like a relational engine.

Figure 3.5 illustrates the basic pluggable architecture of the Brackit engine. Note how Language, Engine, and Store are separated modules, which can easily be extended or replaced. The glue that connects Language and Engine is a parser, which understands the corresponding syntax and generates the logical query representation which is further analyzed, optimized, and executed by the Engine. The Engine interacts with data stored in documents and collections, which are provided by the Store module, and the glue between them is an XDM mapping mechanism. The standard store of Brackit, for example, accesses documents parsed from XML strings or files, and thus the XDM mapping is performed by a standard SAX parser. Usually, a complete extension package provides not only a new store module, but also extends the compiler by incorporating storage-specific optimizations such as indexes, evaluation push-down, tree navigation algorithms, etc.

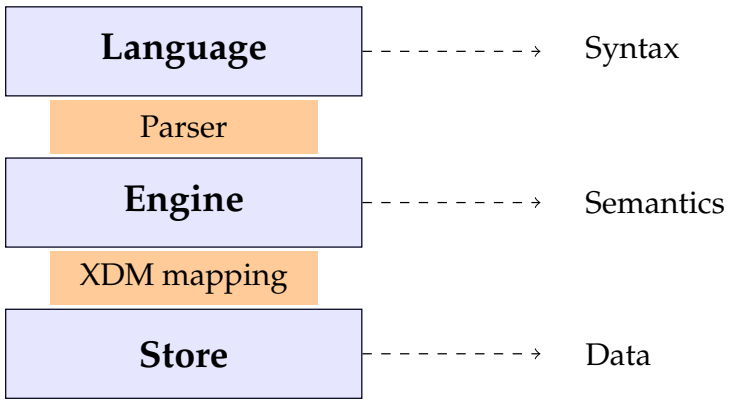


Figure 3.5: Basic modules of the Brackit Engine architecture

To illustrate the extension possibilities, Figure 3.6 shows three scenarios for which Brackit can be tailored: (a) represents the native XML database *BrackitDB*, (b) a hybrid XML-relational engine, and (c) our distributed MapReduce engine.

Compilation Process

The compilation process in Brackit is illustrated in Figure 3.7. The Parser module at the top generates an abstract syntax tree, short AST, which is used throughout the compilation phases as logical query representation. At the end of compilation, the final AST is then passed on to the Translator module, which generates a tree of executable physical operators. The steps

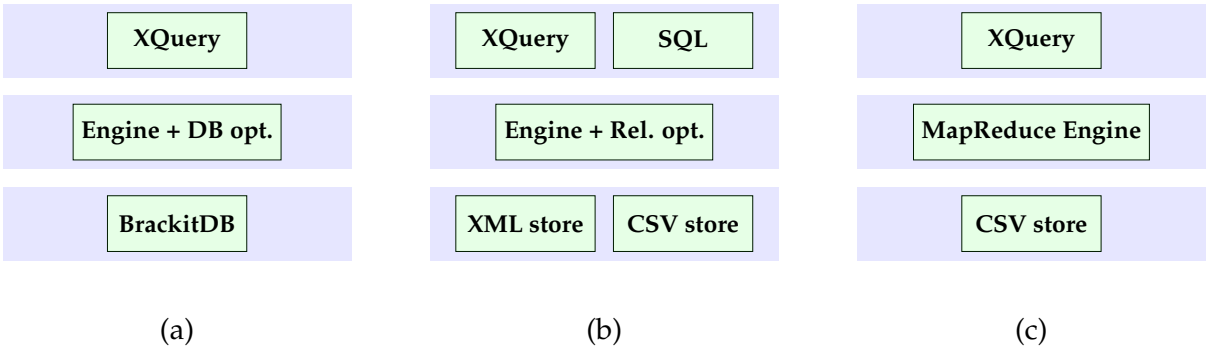


Figure 3.6: Three possible extensions of the Brackit XQuery Engine

in between the Parser and the Translator are what constitute the Compiler module, and they correspond to sequences of rewrite rules which are applied to the AST.

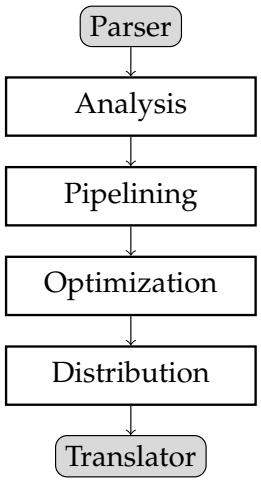


Figure 3.7: Overview of compilation process

The first phase of the Compiler module is Analysis. Its goal is to perform static typing and annotate expressions with typing information, as well as to perform simple rewrites such as constant folding and introduction of let bindings that simplify optimization. The next phase is then Pipelining, which transforms FLWOR expressions into pipelines—the internal, data-flow-oriented representation of FLWORs, which we discuss in Section 3.4. Pipelines are subject to several optimization rules (such as push-downs, join recognition, unnesting, etc.), which are applied at the Optimization phase. The Distribution phase is specific to distributed computing scenarios, which is where knowledge about the partitioning of documents and collections is applied to the query. In our MapReduce approach, certain pipeline operators are replaced by distribution-aware operators, as discussed later in Chapter 4.

3.4 FLWOR Pipelines

The tuple-stream semantics of FLWOR expressions is more naturally modelled as a pipeline of operators which manipulate tuples in a set-oriented style of processing, making use of the open-next-close paradigm [19].

FLWOR pipelines are generated from a FLWOR expression tree, during the Pipelining phase of the compilation process. The expression tree representation corresponds to an AST node `FlowrExpr` which contains all clauses as children. In the pipeline view, however, each clause is translated into an *operator*, and the operators are arranged in a top-down sequence, as illustrated in Figure 3.8.

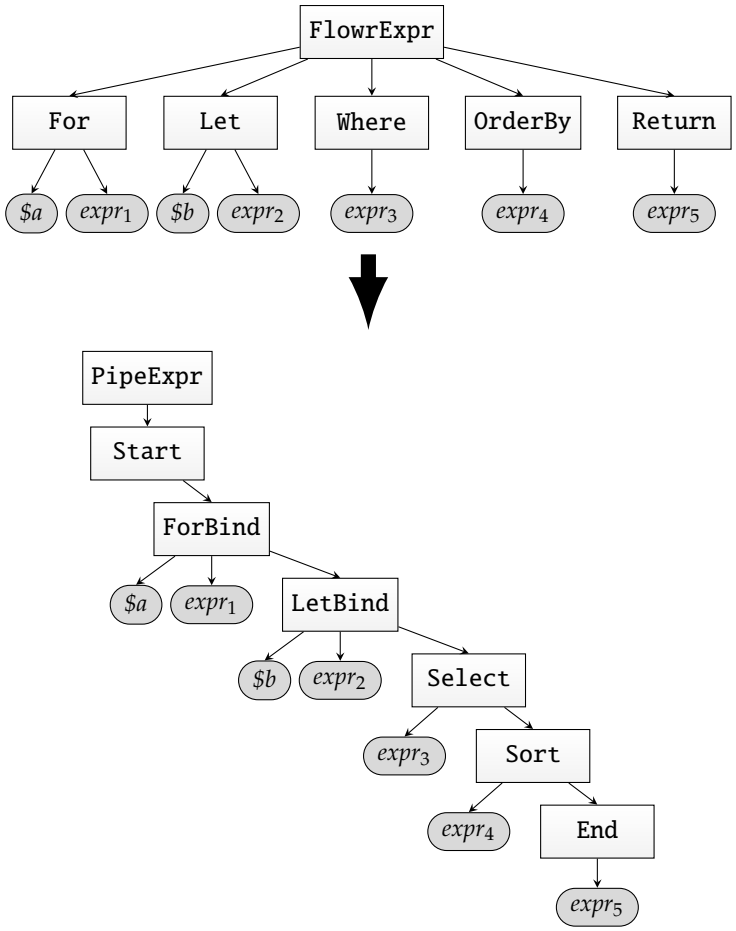


Figure 3.8: Translation of a FLWOR expression tree into a pipeline

A standard open-next-close pipeline is represented bottom-up, since a tuple is requested from the final operator, which requests a tuple from the previous operator and so on. The bottom-up style is actually used in the physical query representation of Brackit, but the top-down representation is preferable for analysis and optimization of the AST. Many pipeline rewrite rules must consider variable scopes to decide upon the dependence between operators, and a top-down representation allows for a direct computation of a scope, which is simply the complete subtree under the binding operator.

We make use of the example in Figure 3.8 to explain how FLWOR pipelines work. First, since every XQuery program is constituted of a tree of expressions which deliver sequences when evaluated, there must be an expression node which represents the evaluation of the complete FLWOR pipeline, and this is `PipeExpr`. In short, a `PipeExpr` triggers the execution of the pipeline under it, delivering the result of the last operator (always an `End`) to the containing expression. Thus, it represents a control point for switching between the bottom-up logic of expressions and the top-down logic of pipelines.

The pipeline execution starts with a `Start` operator, which simply generates an initial empty tuple as context to the initial binding operator. In the top-down representation, the next operator in the pipeline is always the rightmost child, into which the generated tuples are fed. The other children are either expression nodes which must be evaluated to produce the desired tuple output, or additional operator parameters such as the direction of sort order and whether a `ForBind` should produce tuples for empty sequences.

The operators `ForBind`, `LetBind`, `Select`, `Sort`, and `Group` correspond directly to the `for`, `let`, `where`, `order by`, and `group by` clauses. Their effect on the tuple stream is exactly the same as the semantic description of the clauses given in Section 3.2.

The `End` operator has a special purpose, contrasting to that of `Start`. It represents the `return` clause, and as such, it must take the tuple input, evaluate the given expression according to it, and deliver a sequence as result. Therefore, an `End` does not produce output tuples and finishes the pipeline execution. Its results are fed directly to the `PipeExpr` at the top of the pipeline.

Join Processing

A FLWOR pipeline in which two subsequent `ForBind` operators occur ends up generating a cartesian product between their respective sequences. As in relational processing, this causes an explosion on the number of tuples which are generated within the pipeline, unless there is a predicate which filters the pairs generated by the product, which gives opportunity to a `Join` operator. Hence, a `Join` can be used to replace a subsequence of two `ForBind`s and a `Select`, as long as the following conditions hold:

1. The expression in the `Select` is a comparison between two expressions e_1 and e_2
2. The intersection of the transitive closures of the variable references in e_1 and e_2 may only contain the variables which were available before the first `ForBind`

Condition 2 defines *independence* between the two join inputs. The transitive closure of a variable reference is the variable itself together with all variables on which it transitively depends. Therefore, the intersection of transitive closures of two expressions is simply the set of variables which they both (direct- or indirectly) refer to. Being independent, hence, means that all variables to which both expressions refer are present in their scopes regardless of the order in which their variables are bound. Note that the variable-binding nature of FLWOR expressions requires this independence analysis, which makes our `Join` operator more general than a relational join.

Figure 3.9 shows the resulting `Join` operator in a pipeline. *Input* represents the pipeline section which occurs before the first `ForBind` involved in the join. The variables bound in

Input may be referenced by both join branches, since they correspond to previous context available to both branches, and thus do not make them dependent. The Θ symbol represents a comparison operation (such as $=, <, >, \ldots$), and the expressions e_1 and e_2 satisfy the condition 2 stated above.

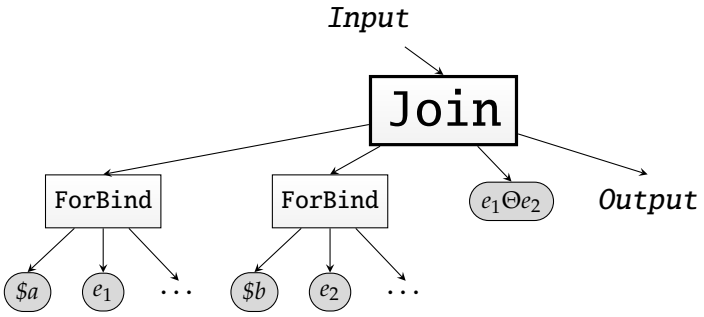


Figure 3.9: The `Join` operator

Note that the `Join` operator has in fact 3 inputs: the standard left and right inputs plus a common input which is available to both of them.

3.5 Processing Relational Data

As mentioned in Section 3.3, we developed an extension of the Brackit engine that incorporates a simple CSV-based storage module together with an appropriate XDM mapping mechanism. Other than demonstrating the versatility of the Brackit engine, this extension was motivated by the simplicity of relational data for our MapReduce mapping mechanism. Since encoding relational data is straight-forward using CSV files and partitioning rows of a table is a trivial task, we may focus primarily on the query processing aspects, without considering complex XML encodings and partitioning strategies. However, since the basic Brackit engine abstracts the store module by means of XDM mapping, enhancing the system with a MapReduce-suitable XML store does not interfere with our query translation mechanism.

XDM Mapping

The basic XDM mapping strategy can be divided into two parts: (i) the construction of a relational tuple from a serialized representation; and (ii) the mapping of XDM operations to operations on a relational tuple. The former is an abstraction of the physical representation or encoding, and because it can easily be abstracted and encoding schemes are not the focus of this work, we opted for implementing a CSV-based encoding, which corresponds simply to the string representation of each field in a tuple separated by a special character such as a comma. The latter part, however, is of greater interest, because it actually represents the ability to process relational objects using XQuery. The goal is to wrap a relational tuple in an object that implements the interface of an XDM instance, namely an *element*. The mapping scheme is visually represented in Figure 3.10.

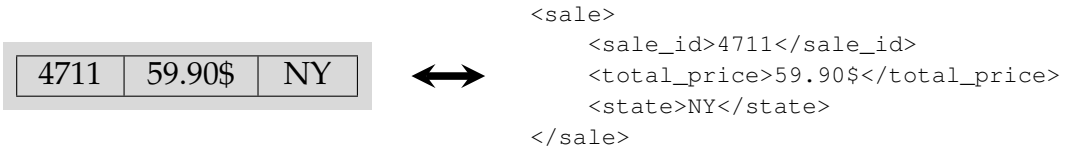


Figure 3.10: Mapping between a relational tuple and an XDM instance

A row[2] is encoded as an element whose name is simply the name of the table to which it belongs. Fields are encoded as child elements, and the actual value of a field is contained in a single text node inside it. One could argue that fields could be more naturally encoded as attribute nodes, since they are simply key-value pairs where the key is the column name. The element approach, however, gives more flexibility, since we may use attribute nodes exclusively to encode metadata information such as type, provenance, or constraints. Figure 3.11 shows an example of FLWOR expression which accesses the sample table of Figure 2.3.

```
for $s in doc('sales')//sale
where $s/total_price > 5.00
let $state := $s/state
group by $state
return
  <result>
    <state>{$state}</state>
    <avg-sale>{avg($s/total_price)}</avg-sale>
  </result>
```

Figure 3.11: A sample FLWOR expression on a relational table

Rows as Nodes

We describe our relational-XDM mapping mechanism using object-oriented techniques. The wrapper that provides node behavior to a row is a `RowNode` object, which must implement the operations described in a `Node` interface. The interface specifies mostly navigational operations such as *getChildren*, *getNextSibling*, and so on; structural comparisons like *isParentOf*, *isDescendentOf*, etc.; and XDM-related functionalities like *getIdentity*, *getNodeName*, *atomize*, etc.

In order to simulate node navigation in a relational tuple, a `RowNode` must keep track of which *level* it is currently pointing to. Possible levels are: (i) *document*, which corresponds to the document node that represents a whole table; (ii) *root*, which is the root element that contains all rows as children; (iii) *row*, which is the element that encodes a single row; (iv) *field* which corresponds to the element of a particular field inside a row; and (v) *value*, which is the

[2]Alternatively, we use the short notation "row" to refer to a tuple of the relational model, avoiding confusion with tuples of bound variables that occur in FLWOR expressions.

text node that contains the value of a field. Additionally, for the 2 lower levels, the object must keep track of the column index that corresponds to the current field. This additional information provides an XML node structure to relational tuples, allowing a `RowNode` to properly implement the XDM operations defined in the `Node` interface.

Figure 3.12 illustrates the structural relationship between `RowNode`s. Each box represents a different instance of a `RowNode` object, annotated with their *level* and *column* attributes. A `RowNode` with the levels *row* or below points to an instance of a relational tuple, which means that the same tuple can be shared among different `RowNode` instances that are related to the same row. Hence, `RowNode`s *per se* are very lightweight objects, which makes navigation efficient because generating copies of nodes at different levels is an inexpensive operation.

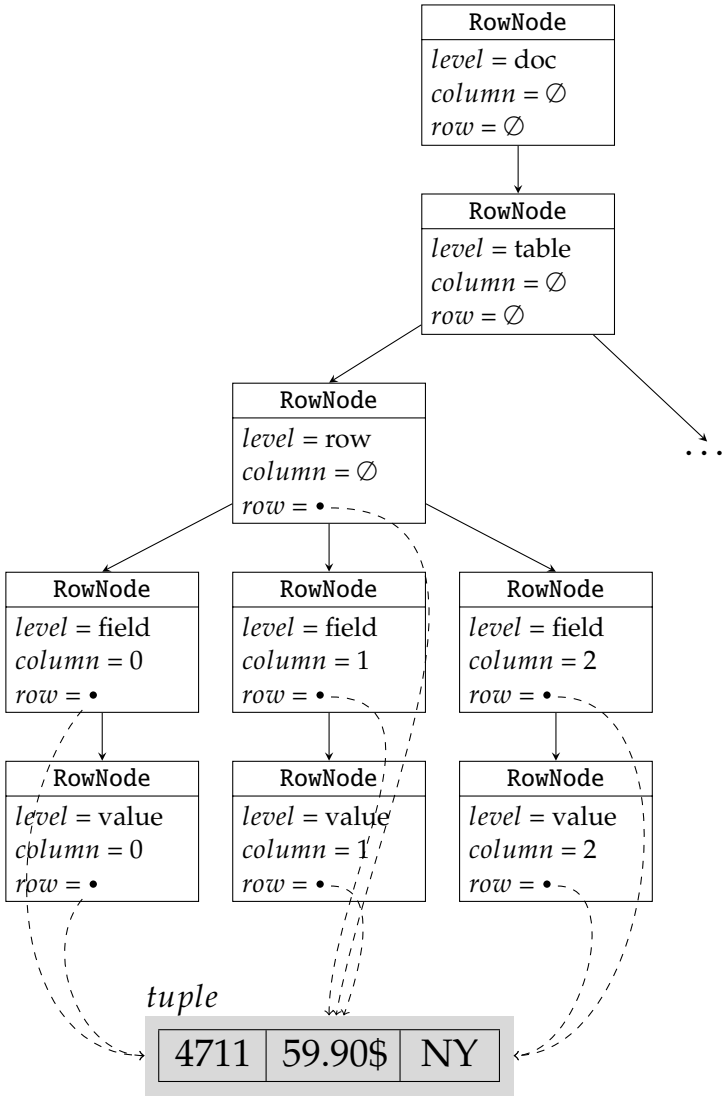


Figure 3.12: The XDM structure implemented by `RowNode`s

Metadata Catalog

The `RowNode` scheme provided so far requires a mapping between column names and field indexes. This information is kept centralized in a *metadata catalog*, which contains not only physical mapping information such as field indexes and location of tables, but also schema information like defined in SQL DDL statements.

The catalog provides a list of table definitions, each one identified by a name which is used to access rows through the XQuery functions *doc* and *collection*. The former function returns a `RowNode` with level *root*, while the latter returns a sequence of `RowNode` objects at the *row* level, allowing a direct scan through the tuples in the tables. Each table definition contains a list of column definitions, which contain the name, field index, and type of each column. Not only is this metadata information required for implementing the above *RowNode* schema, but it also allows optimizations to be done at a query level, as described in the following section.

Relational Access Optimization

`RowNode` behaves transparently like an XDM element, and as such it can directly be processed by the XQuery engine, without any further modification. However, to provide efficiency comparable to a relational engine, we must extend the compiler to recognize nodes which are actually rows and make use of faster access methods than element navigation.

The technique we use is to replace path expressions which start on a `RowNode` with a customized column access expression. Using standard XDM processing, a path expression like `$s/state`, where `$s` references a `RowNode` with level *row* and `state` is one of its fields, would be evaluated by calling the *getChildren* operation. It delivers a list of `RowNode` objects with level *field*, each with increasing values for the *column* attribute. This list would then be iterated, and all nodes with the QName `state` would be returned as a result sequence. A more efficient evaluation is performed by a column access, which consults the table metadata to retrieve the field index of the column `state`. The index can then be used to access the value of the field directly, eliminating the need to create superfluous field nodes and iterate over them.

To allow path expressions to be recognized as column accesses, however, a previous analysis of the FLWOR pipeline must identify `ForBind` operations which bind to a call to a XQuery `doc` or `collection` function, and whose argument is identified in the metadata catalog as a relational table. Path expressions that start on that variable and contain only forward axes can be identified as column accesses. Figure 3.13 shows an example pipeline where the *sales* table is accessed as a collection, binding nodes with level *row* directly to the variable $*s*. The path expressions $*s*/*state* and $*s*/*sale_id* are then substituted by the `ColumnAccess` AST node.

Another significant optimization step which can be performed is *projection*. Once all relevant path expressions have been replaced by column accesses, an additional analysis step identifies which columns are accessed in the query. When constructing rows from the physical representation, the unused columns may be skipped, which saves significant IO time. This optimization is particularly useful in analytical computations, since tables tend to have many columns, but each query only accesses a small subset of them. Our CSV-based storage, however, does not support skipping, since the field boundaries in a row are not known beforehand.

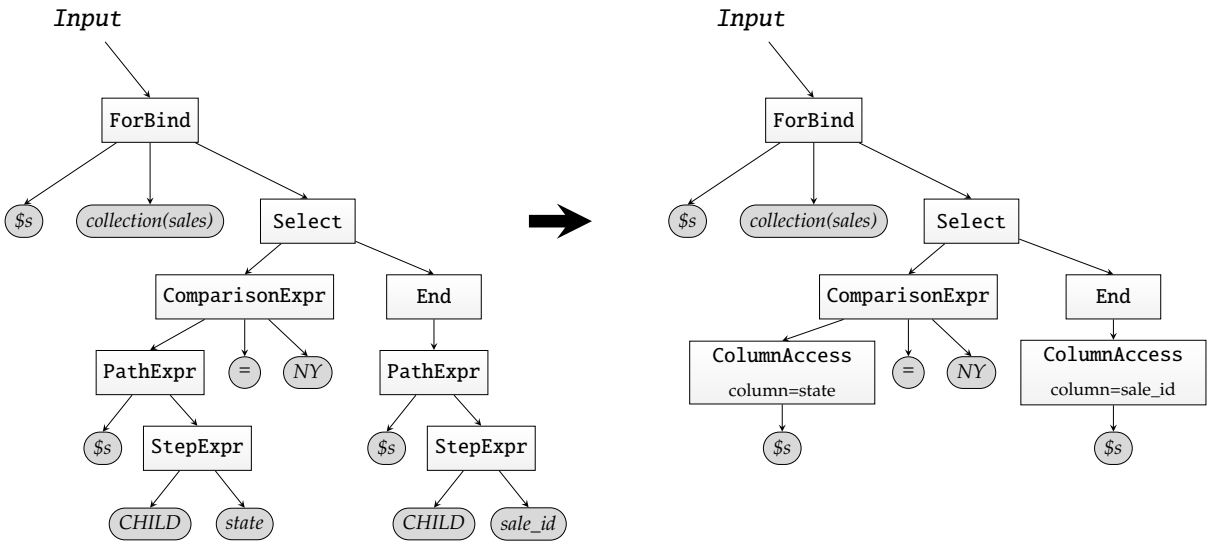


Figure 3.13: Transformation of path expressions into column access

Nevertheless, in order to allow projection, we developed a binary variation of CSV files, which makes use of a length-field in the encoding.

The final optimization step which we implemented in our relational extension is the elimination of atomization, which also delivers a significant performance boost. When used inside expressions, a `RowNode` at field level must usually be atomized. Hence, in a comparison like `$s/total_price > 5.00`, for instance, the column access expression would return a `RowNode` on which the comparison expression would call the *atomize* operation. Hence, the construction of an intermediary `RowNode` is completely superfluous, since the column access already has access to the atomic value of the `total_price` field. These situations can be recognized and the column access expression annotated with an *atomize* flag, which causes the expression to deliver directly an atomic value when evaluated.

Chapter 4

Mapping XQuery to MapReduce

Now that the computational models and execution environments of both MapReduce and XQuery are introduced, we proceed to the kernel concept of this thesis: the mapping framework for execution of XQuery programs in the MapReduce middleware.

First, for a contextualization, we discuss the typical scenario for which our framework is designed. The data to be processed is stored in collections in the distributed filesystem. Because these collections are plain files, which may be used by other systems than MapReduce, our framework cannot control the way in which these files are encoded and managed. This is in contrast to a database system, which has the freedom to create indexes and use efficient record encodings that allow fast random access, byte-level comparisons, and projection. Since we do not assume that our framework exercises exclusive control over the collections, such storage-level optimizations must be abstracted. Therefore, our framework serves only as a query processing engine, and not as a complete database stack. Considering a layered model of DBMS architecture [22], this means that we only implement the topmost layer, namely the set-oriented interface. The layer below it would be represented by MapReduce, which provides the basic record (i.e., key-value-pair) operations. Because the framework has no control over the data and performs only read operations, the MapReduce layer sits directly on top of the file services layer—in our case represented by the distributed filesystem.

In the plain MapReduce model, i.e., at the record-interface layer, computations are described by means of Map and Reduce functions, which are usually implemented in a general-purpose programming language such as Java, C++, or Python. This calls for a language abstraction layer, which allows to program MapReduce jobs using a higher-level, data-processing language such as SQL or XQuery. The goal of our framework is to abstract not only the language which is used to describe the computations, but also the programming model of MapReduce. Hence, the complete MapReduce back-end is transparent to the user, and all computations are described using plain XQuery.

Being the core of data-intensive computations in XQuery, we concentrate our efforts on the mapping of FLWOR pipelines, which is the subject of Section 4.1. Despite our approach being based on XQuery, the mapping concepts presented also work for general query processing, because the characteristics of each operator are similar in languages like SQL as well. Furthermore, FLWOR pipelines may actually be compiled directly from SQL syntax, in which case our mapping can be directly applied to execute SQL on MapReduce.

Given the early development stage of our prototype, we do not support arbitrary pipelines. The restrictions which must be observed are presented in Section 4.2, where we also discuss rewrite rules that can transform a wide spectrum of queries into this restricted form. Sections 4.3, 4.4, and 4.5 describe the processes of translation into the MapReduce model, compilation to executable job specifications, and execution in a Hadoop environment, respectively.

4.1 MapReduce as a Query Processor

A query in MapReduce is represented by a sequence of jobs, each of which realizes a section of an operator pipeline, and hence corresponds to a composition of higher-order patterns that consume and produce tuple streams. The mapping process consists of first breaking down the pipeline into sections to be run in Map and Reduce functions, using the technique of *Pipeline Splitting* discussed in Section 4.3. The second step is to apply *Job Generation* to group these patterns and form MapReduce job specifications, a process discussed in Section 4.4.

The basic idea behind Pipeline Splitting is the distinction between two kinds of patterns: *blocking* and *non-blocking*. Blocking operators apply an operation on the complete input, and hence they must block the pipeline of *next* calls in the open-next-close evaluation and store all tuples in a local buffer. Non-blocking operators, on the other hand, operate one tuple at a time, maintaining the pipeline flow. Despite the nomenclature being based on technical implications, the distinction is related only to the nature of the processing pattern, which means we can classify them into these two classes regardless of any evaluation strategy.

In Section 2.4, we established that the Mapper and Reducer patterns are essentially applications of a transformation function, with the restriction that Reducer values must be lists. Hence, because they individually and independently apply a transformation to key-value pairs, both Mapper and Reducer belong to the class of non-blocking operations. The Shuffle pattern, on the other hand, reorganizes the complete input, by sorting and grouping key-value pairs. It is thus a blocking operation.

Based on this classification, we provide a mapping technique for each class of operators in the following discussion. The techniques introduced here aim to provide a basic understanding of the mapping strategy on a operator level. The actual mapping rules on the pipeline level will be algorithmically described with Pipeline Splitting in Section 4.3

Non-blocking Operators

A non-blocking operator, or a composition thereof, fits the transformation pattern of a Mapper or Reducer phase, since it manipulates one tuple at a time. This means that a sequence of one or more non-blocking operators can be executed inside a Map or Reduce function, in a technique which we refer to as *wrapping*. From the computational model perspective, this can be understood as using the query processing patterns as parameters to the patterns of MapReduce, meaning that each invocation of Map or Reduce is in fact evaluating a pipeline section on a (usually small) subset of the tuples.

The first kind of non-blocking operators are *data access operators*, which scan data items from some input collection and provide them as tuples to the succeeding operators. Logically, they

have a unique abstract representation, regardless of the underlying physical structure being accessed, such as tables or indexes. In our case, data access is logically represented by a `ForBind` operator parameterized with a *doc* or *collection* function. Despite the function being the actual data source, the `ForBind` is responsible for introducing the data items in the pipeline context as tuples, and hence we consider it as the primitive data access entity. A `ForBind` that binds to a sequence which is not originated from external documents or collections is not considered a data access operator.

In the MapReduce model, input data is accessed in the Mapper phase, where each item is passed to an invocation of the Map function. As discussed in Section 2.3, the Mapper pattern is equivalent to a procedural for-loop that applies a function to each element. Hence, the iteration semantics of a `ForBind` is covered implicitly by the input mechanism of the MapReduce framework, and the only task which must be executed inside the Map function is binding each item to the corresponding variable. Therefore, we apply an *instantiation* technique, replacing the `ForBind` operator with a `LetBind` and the iterated sequence with the individual item being scanned. For example, a `ForBind` which binds a variable `$b` to a sequence of `book` elements is replaced by a `LetBind` on a particular `book` already retrieved from the collection, as illustrated in Figure 4.1. Note that the instantiated pipeline is executed once for each book in the collection, and thus it is equivalent to the `ForBind` version.

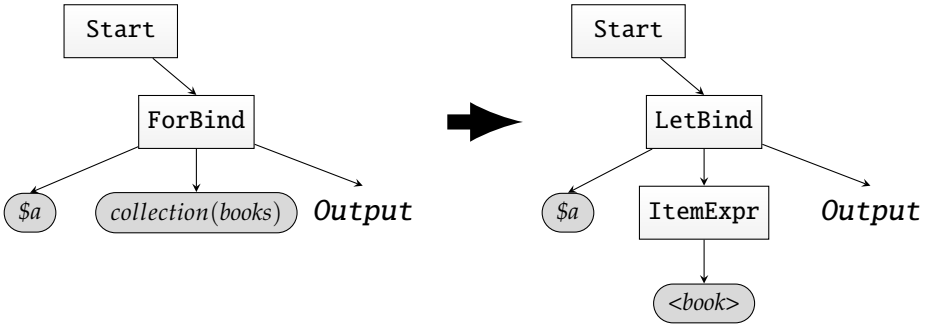


Figure 4.1: Instantiation of a `ForBind`

On the other end of the pipeline, execution is always finished with an `End` operator, which corresponds to the `return` clause of XQuery. In contrast to data access operators, which fetch items from input collections and thus are executed in a Map function, the `End` operator must be wrapped in a Reduce function, so that its results are written to the distributed filesystem as specified by the MapReduce execution model.

The generation of results currently does not take advantage of the XDM mapping techniques used for reading nodes, which would allow the writing of XDM data back to native formats such as CSV. The options for implementing this feature would be to either set some environment variable or to extend the syntax of constructor expressions to specify a particular format. Since this is not of primordial importance to our query mapping purposes, our current prototype relies on standard Brackit in-memory nodes, which are serialized to output files as plain XML strings.

The remaining non-blocking FLWOR operators are `LetBind`, `ForBind`[1], and `Select`. A contiguous sequence of such operators can be composed and executed inside a single Map or Reduce function, by applying the wrapping technique mentioned earlier. A precise specification of the wrapping technique is given later in Section 4.3, where we consider operators in the context of a pipeline, instead of isolated. As we shall see, the decision of whether to wrap operators in a Map or in a Reduce function depends on the context provided by the translation of the previous operators in the pipeline.

Blocking Operations

The blocking nature of the Shuffle pattern makes it a candidate for realizing the blocking operators `Group`, `Sort`, and `Join`. However, it is only because of the generality of the sort and group operations performed inside it that these operators can in fact be implemented using a Shuffle. The idea is to apply a *translation* technique, which maps the operator to a semantically equivalent operation using a Shuffle together with pre- and post-processing steps, in the following manner:

1. Generating the correct key on the previous Mapper or Reducer phase
2. Sorting and grouping during the Shuffle phase
3. Prepare the grouped tuples for further execution in the Reducer phase

Since step 2 is carried out implicitly by the MapReduce framework, our translation mechanism must take care of steps 1 and 3. For that purpose, we introduce two auxiliary operators: `PhaseOut` and `PhaseIn`.

The `PhaseOut` operator corresponds to step 1, and thus it is executed as the last operator of a previous Mapper or Reducer phase. It has no effect on the tuple stream, as it simply extracts from the tuple the variable bindings which must be used as key to the following Shuffle phase. Note that we do not require the phase before a Shuffle to be a Mapper, as in the MapReduce model, because the restriction is not necessary in our scenario. As explained later in Section 4.3, we can simulate a Shuffle following a Reducer by introducing a so-called *identity Mapper*, which produces its input directly as output without any transformation.

The AST representation of a `PhaseOut` contains multiple variable-reference expressions as children. Given an input tuple, the `PhaseOut` extracts the fields corresponding to the specified variable references and generates them as a compound key. The remaining variables are then used as value. Figure 4.2b illustrates a `PhaseOut` operator which generates a key composed of variables `$a` and `$c`, resulting from the translation of the `Group` in Figure 4.2a.

Once the Shuffle fetched all pairs produced by the `PhaseOut` and generated the grouped tuples, these are read in the Reducer phase, where an additional finalization step is carried out by the `PhaseIn` operator, illustrated in Figure 4.2c. The first task is to rebind the variables in the tuple stream for the evaluation of the following operators. This is necessary because the previous phase delivers only the raw tuple data, without the proper information of which index corresponds to which variable name (i.e., the dynamic context). For that purpose, the

[1]Note that here we are not concerning `ForBind`s that perform data access, since these are instantiated into a `LetBind`. Thus it is safe to treat a `ForBind` as a non-blocking operator.

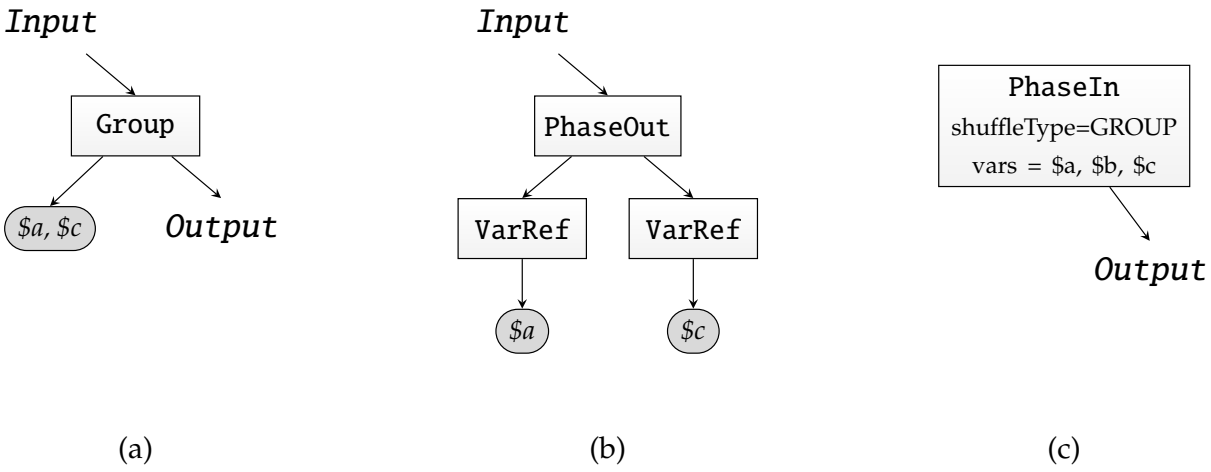


Figure 4.2: A `Group` (a), and its corresponding `PhaseIn` (b) and `PhaseOut` (c) operators

`PhaseIn` AST node contains a list of variable names—ordered according to the respective tuple indexes—as a property.

Furthermore, the `PhaseIn` operator performs a reorganization of the tuple stream, depending on the *shuffleType* property, which specifies which blocking operator is to be translated. `PhaseIn` is thus a polymorphic operator. In the case of a `Group`, a reorganization is necessary because the Shuffle only delivers all tuples that have the same key as a list of values. To correctly realize the semantics of the `group by` clause, we must merge these tuples into a single one, by concatenating all non-grouped variables in a single sequence and leaving the grouped variables with their unique values. This process corresponds to step 3 above, and it is executed in a `PhaseIn` operator in the Reducer phase which follows the Shuffle.

To implement `Sort`, we take advantage of the sorting operation executed by the Shuffle. Each partition of the key space K_2—produced by the Shuffle and processed by a Reducer task—is guaranteed to be sorted by the key. The Reduce function is invoked sequentially for each key, with a list of tuples which have the same value on the sorting key in the $[V_2]$ parameter. These tuples can be passed on to the further operators without any merging step. Because of the sequential invocation, these operators are also evaluated sequentially and in the order specified by the sorting, hence correctly implementing an `order by` clause.

The translation of a `Join` operator is not so straight-forward as the other blocking operators, because the MapReduce framework was not designed to directly support operations with multiple inputs. Nevertheless, a simple extension of the execution model—which is implemented in Hadoop—allows a `Join` to be realized using a Shuffle that gathers output from many Mappers. Because each Shuffle task simply fetches key-value pairs of a specific partition from M Mapper tasks, it makes no difference if, for example, one half of the M tasks executed a different Map function than the other half. The extended model, therefore, supports the specification of multiple Mappers, which must all generate output in the same space (K_2, V_2). The key idea to implement joins is to process the left and right inputs using different Mappers and then combine them using a single Shuffle that groups their tuples based on the join value. The Reduce function then receives all tuples from both inputs that have the same join value, and

the post-processing step is used to separate the inputs and build the join results.

The key generated by each `PhaseOut` operator (one for the left input Mapper and another for the right input Mapper) is a composite of the value which is to be used as join key and a *tag* which identifies whether the associated tuple comes from the left (tag = 0) or the right (tag = 1) input.

The Shuffle phase must then group all values from both inputs which have the same key, but we must ensure that the comparator function used to form groups ignores the tag value. Otherwise, tuples with the same join value but from different inputs will end up on different Reduce function invocations. As mentioned in Section 2.4, the Shuffle can be parameterized with a comparator function, the default one being an equality on the complete key. Hence, we parameterize using the *ignoreTagCompare* function, which provides the desired behavior.

Once the customized Shuffle is executed, each invocation of Reduce will obtain a list of all tuples from both inputs which have the same join key. At this point, the join matching has already been done, and the only remaining step is to separate tuples from the left and right inputs using the tag, and combine them using a cartesian product. This logic is executed by the `PhaseIn` operator.

Currently, we only support equality comparator functions, and thus only equi-joins can be translated. The extension to different join types, however, is a simple matter of providing additional comparators. In the XQuery context, we currently only support value comparisons, which are based on atomic variables. General comparisons, which operate on sequences, would require a more complex key generation mechanism and are thus not supported. Further restrictions of our current `Join` translation are discussed in Section 4.2.

This technique of executing joins with multiple Mappers and tags is known in the literature as *Reduce-side Join* [36], because the join is executed partially during the Shuffle and finalized in the Reduce function. This is the most general join algorithm for MapReduce, but if certain assumptions are satisfied, it is more efficient to use alternative algorithms, such as a *Map-side Join*. This is just one of the cases in which an optimizer module could be employed to find efficient MapReduce translations for query pipelines. Such optimization steps are currently not supported by our prototype, but the initial requirements are sketched later in Section 5.1.

The use of the special `PhaseOut` and `PhaseIn` operators represents a compromise between the computation models of query processing and MapReduce. In the former case, there is a manifold of operators with well-defined semantics and a fixed signature (consume and produce tuples), and little to no parameterization is required beyond the expressions evaluated by each pattern. The MapReduce approach, on the other hand, consists of generic and versatile patterns, whose behavior is only loosely specified, and the actual semantics of operations is given by parameters. The `PhaseIn` and `PhaseOut` operators fit in the query processing model by behaving like operators in a pipeline, but additionally they are aware of a MapReduce evaluation context, enabling a smooth and well-defined mapping mechanism to take place.

4.2 Normal-Form Pipelines

Due to the characteristics of the MapReduce execution model and mostly to the early development stage of our prototype, we do not support the mapping of the complete specification of FLWOR expressions introduced in Section 3.2. This section provides a restricted form of FLWOR pipelines for expressions which can be run in MapReduce using our prototype, as well as an explanation for the restrictions which must be applied.

The basic restriction is that the top-level expression must be a FLWOR. If the program consists, for example, of a node constructor whose contents are delivered by one or more FLWOR expressions, it is currently not suitable for execution in MapReduce. However, our prototype could easily be extended to support such scenarios by either: (i) executing top-level non-FLWOR sections in the local machine, submitting only the FLWOR parts to be executed in MapReduce; or (ii) normalizing any top-level expression into a FLWOR using rewrite rules. A further restriction is that we currently only consider the body of an XQuery program, ignoring preamble declarations such as user-defined functions or external variables.

For expressing the restrictions on FLWOR expressions, we choose a structural description of the pipelines which are accepted by our prototype, instead of providing a syntactic description of the supported expressions with a grammar. The reason is that our MapReduce mapping is already based on the logical query plan represented by FLWOR pipelines, without any consideration for language syntax. Hence, it would be cumbersome to describe our mapping rules using a pipeline representation and the accepted normal form using the query language syntax. Furthermore, the same syntactic expression may have multiple pipeline realizations, due to optimization or simplification rules, which would make our system reject expressions that could be supported once rewritten to an equivalent normal form.

Before discussing each restriction in detail, we enumerate them below for reference:

§1. A `ForBind` which performs data access must have a `Start` as parent.
§2. A `Join` must have a `Start` or another `Join` as parent.
§3. A nested `Join` may not have a non-empty left input.
§4. A `Sort` may only have variable references as argument.
§5. Expressions inside FLWOR operators may not contain a `PipeExpr`.

The first restriction involves data access operators, represented as `ForBind`s with an invocation of the `collection` function, which must occur after a `Start` operator. This is an obvious requirement in traditional query languages like SQL, because processing is applied to some external collection of data items, such as a table or an index. In this scenario, data access operators always occur at the "leaves" of the bottom-up query plan, because the evaluation must start with fetching data and thus no previous input tuples have to be considered. The `ForBind` operator, however, is a generalized version, which attaches items from a source sequence to a currently existing tuple stream by using a nested-loops semantics. It is thus not required to occur at the beginning of a pipeline. One can, as in the example of Figure 4.3, bind some variables with a `let` clause and start a `for` loop on the numbers from 1 to 10 before accessing data in a collection. MapReduce, on the other hand, has the same characteristic of traditional query languages, because a computation always starts by fetching data items in the

Mapper phase. Therefore, we must enforce data access to occur immediately after a `Start`.

A simple rewrite rule can be applied to try to bring the data access operator to the top of the pipeline, as shown in figure 4.3. Given our restricted normal form and the unordered assumption, the only prerequisite for the rule to work is that there cannot be a blocking operator between the `Start` operator and the `ForBind` which is to be pulled up.

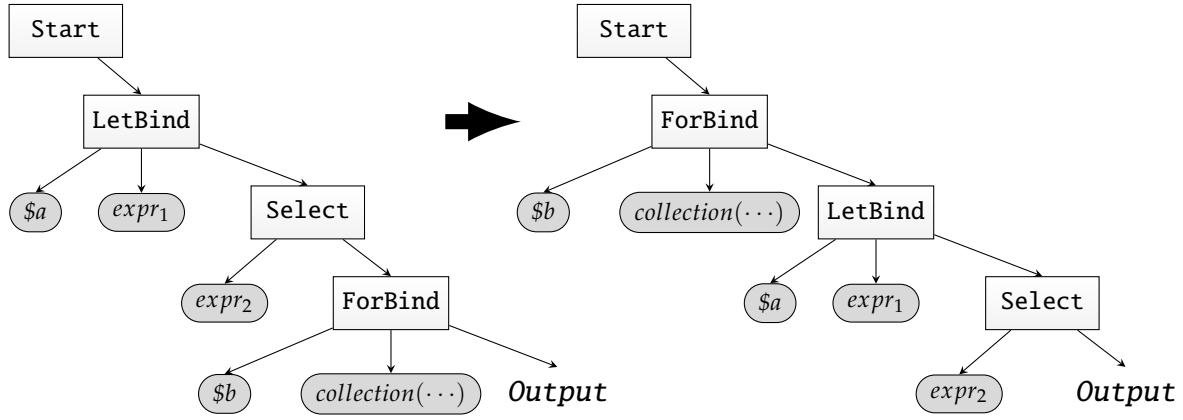


Figure 4.3: Pipeline with irregular `ForBind` and its normalization

The restriction above implies that a sequence of two data access operators is also not supported, since the second `ForBind` would not have a `Start` as parent. Hence, the only way to combine multiple data sources is via a `Join` operator. The same rule of having a `Start` as parent applies for a `Join`, and the reason is again data access. Recall that the pipeline section above a `Join` corresponds to the common input which is available to both left and right branches. Having a common input other than the trivial `Start` means that there is a non-trivial tuple stream which is already available to both branches. However, each branch is an independent pipeline which must perform data access as its first operator. But because there already exists a non-trivial common input, the `ForBind` operators at the beginning of each branch cannot simply fetch and deliver items from a collection—they must combine them with the common input. Therefore, this situation also represents the occurrence of pipelines which do not start with a data access operator.

Allowing only `Start` as parent of a `Join`, however, does not allow the combination of three or more data sources, since it would require a nesting of `Join` operators, a situation in which only the topmost `Join` would have a `Start` as parent. Note, however, that if an inner `Join` is placed directly under a parent `Join`, it still obeys the general rule of having data access operator in the first position. That is because the join itself does not perform any data access, and hence a nesting of n joins corresponds to an independent execution of $n - 1$ pipelines. Therefore, the data access rule still holds as long as the $n - 1$ pipelines also confirm to it, and we alleviate the restriction by allowing another `Join` as parent as well. Note that the argument holds for right-deep, left-deep, and bushy join trees as well.

Similar to the `ForBind` pull-up, a rewrite rule is also available for bringing `Join`s to the beginning of the pipeline. The rule is illustrated in Figure 4.4, where two nested `Join` operators have parents which are neither `Start` nor `Join`. It is applied if the common input is

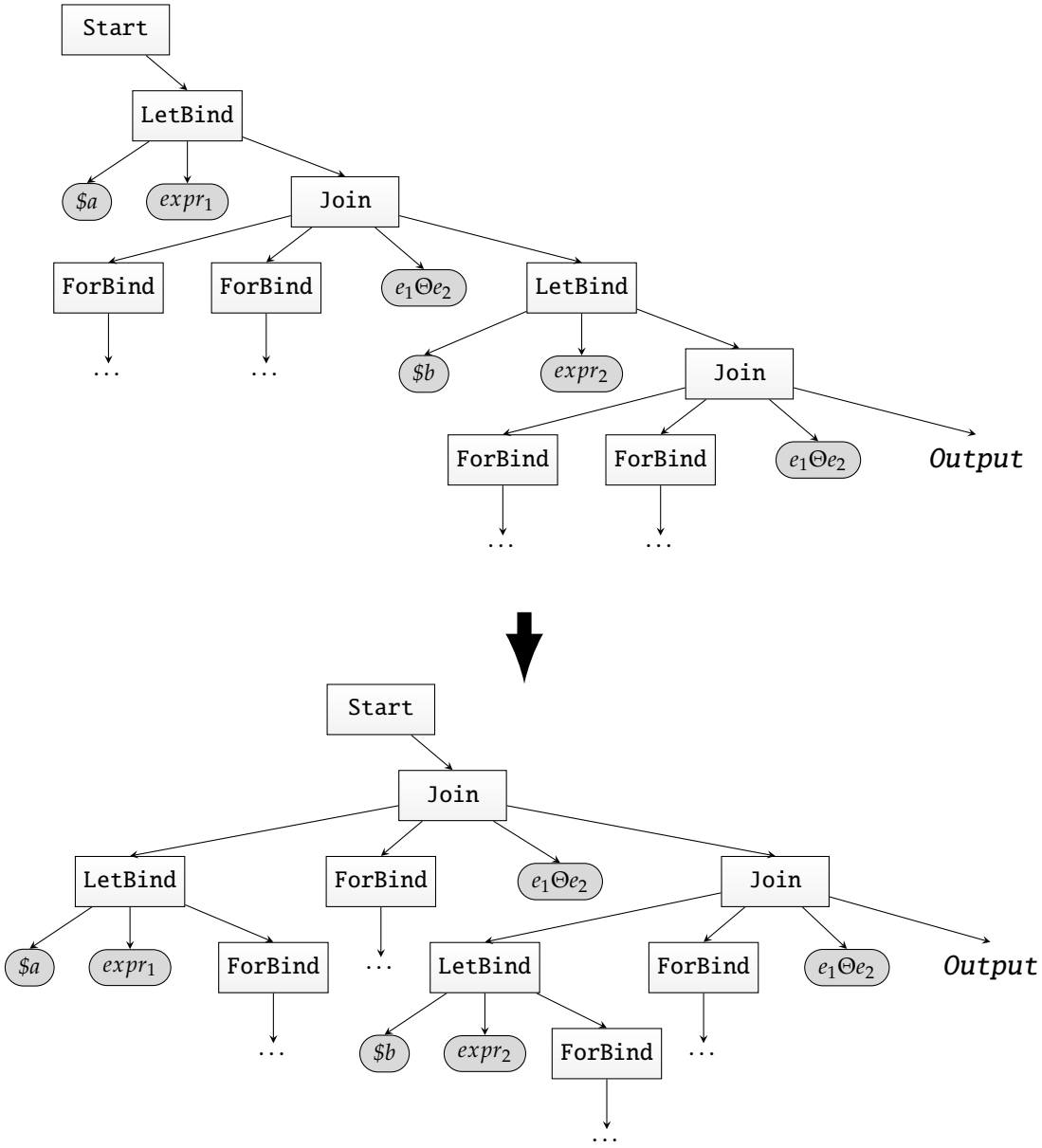


Figure 4.4: Pull-up of `Join` operators for normalization

independent of one of the join branches, which means the complete pipeline section between the `Start` and the `Join` can be pushed down to the other branch.

In the case of a nested join, i.e., whose parent is another `Join`, we impose the restriction that its left input branch must be empty. In the standard Brackit compilation of a nested join, the common input provided by the parent would be fed into the left branch, which is equivalent to having the whole left branch as common input on the top. In our MapReduce scenario, however, precautions need to be taken to avoid breaking restrictions §1 and §2, and this would require complex rewrite rules which deviate from the main focus of this work. Nevertheless, this restriction is rarely a problem, since the join recognition process by default generates empty left branches, having only the common and the right inputs. The left branch of nested

joins is only populated in cases of complex `LetBind` unnesting rules, which we currently do not support anyways.

The fourth pipeline restriction applies to `Sort` operators. Their translation to a Shuffle pattern requires emitting the evaluated order expression as a key in K_2, but as explained later in Section 4.5, our framework uses exclusively a subset of the bound variables as grouping keys. Hence, in order to emit an arbitrary expression as grouping key for a Shuffle, it must first be bound to a separate variable using a `LetBind`. Note that this can be realized automatically with a simple rewrite rule. This generates a semantics equivalent to a relational sort operator, which sorts the rows in a table based on the value of a column. In our case, the rows are the tuples of bound variables, and the columns are variable references. Note that despite grouping keys being generated, no actual grouping of the tuples is performed, because the `PhaseIn` operator that is generated from a `Sort` does not merge the tuples as it is done for a `Group`. The grouping key in this case is used exclusively for sorting.

The fifth restriction concerns the expressions which are used as parameters to the pipeline operators. Because a pipeline is in fact part of a `PipeExpr`, nested FLWORs may occur in any context where an expression is expected, including the parameters of an operator. If a nested FLWOR contains a data access operator, it would have to be evaluated in MapReduce, which would result in a nested-loops execution of MapReduce jobs. Given the large amounts of data, the latencies involved in a distributed execution, and the fact that intermediary results are written to disk after a Mapper phase, nested jobs are prohibitive. Hence, our framework rejects any pipeline which contains a nested `PipeExpr`. Nested pipelines could be supported in the future, since the Brackit compiler already includes rewrite rules that perform unnesting. However, it introduces a generalized version of the `Join` operator which includes a fourth input stream, whose mapping to MapReduce is not yet supported.

4.3 Pipeline Splitting

The first step in mapping a FLWOR expression to a MapReduce job consists of converting the logical pipeline representation—the AST—to a logical description which fits the MapReduce computational model. In the following discussion, we first introduce this target logical description, later on moving to the algorithmic process which performs the conversion.

Splits

The goal of the Pipeline Splitting algorithm is to apply the wrapping and translation techniques mentioned in Section 4.1 to the operators in a FLWOR pipeline and build a tree of *splits*, which are pipeline sections of non-blocking operators executed by Map and Reduce functions. Hence, each split results from an application of the wrapping technique. The edges between splits correspond to applications of the Shuffle pattern, whose keys are determined by a `PhaseOut` operator at the end of the previous split. The key-value pairs produced by the Shuffle are then fed to the next split and processed by a `PhaseIn` operator. Therefore, the translation technique used to map blocking operators generates a connection between two splits, appending `PhaseOut` operator to the pipeline section of the lower split and prepending

a `PhaseIn` to the upper one. Figure 4.5 illustrates a split tree generated for a sample FLWOR pipeline.

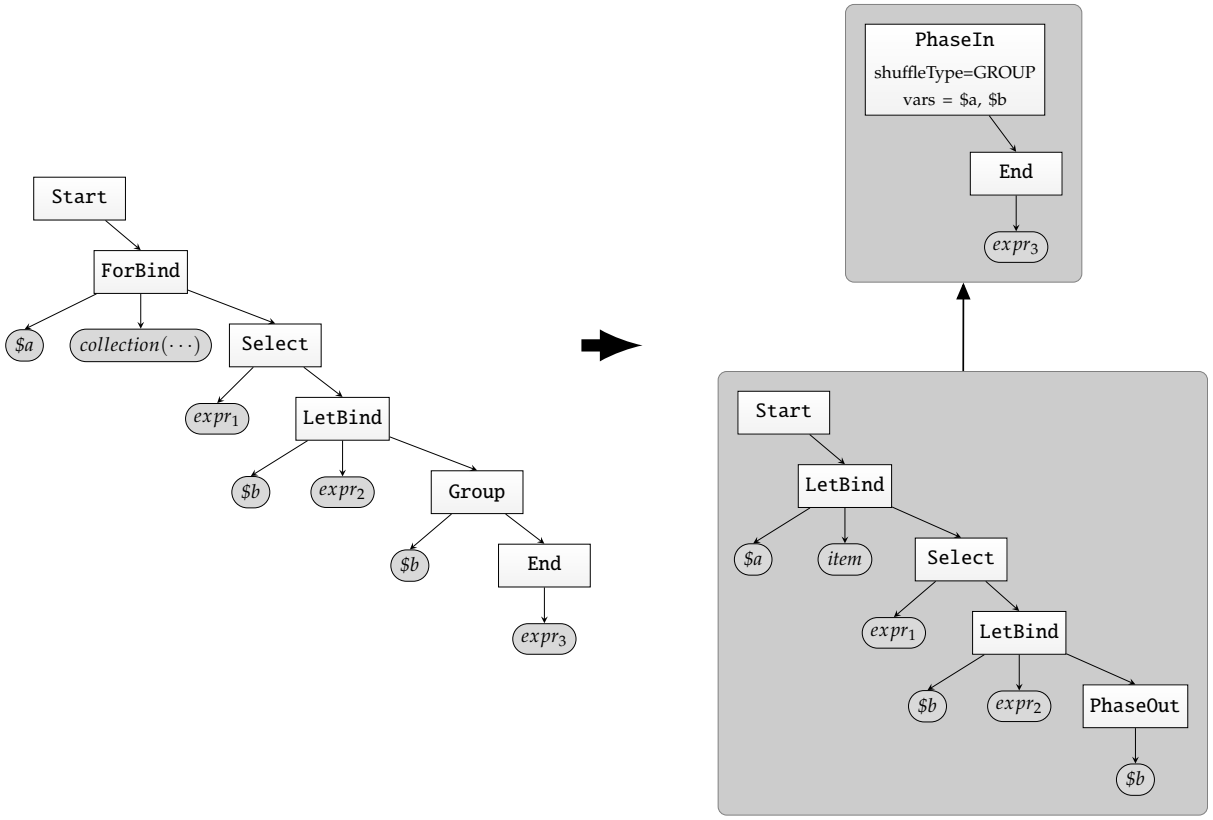


Figure 4.5: A sample split tree generated from a FLWOR pipeline

Splits are organized in a bottom-up manner, as opposed to the top-down representation of FLWOR pipelines. As explained in Section 3.4, the top-down representation is preferred only for applying rewrite rules during optimization, because it directly reflects the scope of variables. When it comes to analysis, execution, or, in our case, translation to a different computational model, the bottom-up representation is better suited. The pipeline section inside each split, however, is kept in the original top-down sequence, because it is later compiled using the standard top-down compiler of the Brackit engine.

A split is a simple data structure, which contains an AST and pointers to left and right input splits. Therefore, a split can be seen as a coarse-grained version of an AST. Because all the logic necessary to express MapReduce computations is encoded in the tree structure—where edges correspond to Shuffle patterns—and the special `PhaseIn` and `PhaseOut` operators, such AST fragments are all we need to generate MapReduce jobs.

The leaves in a split tree represent Map functions which read input files and apply the first operators, while the root corresponds to a final Reduce function which produces the query output. The inner splits can be either assigned to Map or Reduce functions, and this decision is—aside from technical implications which impact performance—irrelevant, since the Mapper and Reducer patterns both execute a transformation, in our case given by a composition of

non-blocking operators. Every inner split has a `PhaseIn` and a `PhaseOut` operator at the extremities of its pipeline, while the leaves only have a `PhaseOut` and the root only a `PhaseIn`.

The tree of splits represents the logical plan of a MapReduce-based query. Much like the AST representation used by Brackit, splits are a static representation which can be analyzed, visualized, optimized, serialized, and submitted to Job Generation at the end to produce an executable representation. It can be seen as an intermediate step between a FLWOR pipeline and a MapReduce job, providing a layer of abstraction between them. A split tree reflects the structure of computations in the MapReduce model, but it abstracts away from technical aspects of the MapReduce execution environment, such as input and output formats, file names and locations, key-value-pair encoding, partitioning strategies, etc. Therefore, splits can be seen as the equivalent of code in a programming language environment, where Job Generation represents the compiler or interpreter.

Splitting Algorithm

The generation of splits from a given pipeline AST is performed by the *splitPipeline* function, which recursively traverses the operator nodes in the AST. The process starts by building an initial split which contains the complete AST, passing it on to the first invocation of *splitPipeline*, whose algorithm is given in Figure 4.1.

At each recursion step, the function is applied to a particular AST *node* in the context of the *current split*. The first task is to validate the AST node for the restrictions of Section 4.2, which is done with the *validate* function. Then, the next operator in the pipeline is saved in the variable *nextOp*, on which the next recursive call will be invoked. The next operator is the last child of *node*, which is retrieved with the method *getLastChild*. Furthermore, an AST node provides the similar methods *getChild* and *getParent*.

The actions performed then depend on the type of operator. The lines 4 to 10 handle the non-blocking operators `ForBind`, `LetBind`, and `Select`. If the operator performs data access, the instantiation technique discussed in Section 4.1 is applied, replacing the XQuery *collection* function with a special `ItemExpr` expression which serves as a placeholder for individual data items. Its evaluation will be covered in Section 4.5. The *bindVariable* function checks whether the operator binds any variable, in which case the variable name is assigned to the current split in an auxiliary table, whose details we omit here. Finally, the function is invoked again with the next operator and the same current split. Thus, a non-blocking operator is simply wrapped in the currently available split.

Once a blocking operator is reached, new splits, which serve as input to the current split, are created. Appending the new splits as children to the current one is the technique which converts the top-down logic of the pipeline AST into a bottom-up split tree. In the case of a `Sort` or a `Group`, handled in line 11, a `PhaseOut` node, created by the *buildPhaseOut* function, must replace the blocking operator (i.e., the *node* parameter) in the AST. This process is carried out by the *replaceNextOp* and *addChild* functions, and it represents the cutting point between the current split and the one which will be created for the following operator. The new split will have a constructed `PhaseIn` node as AST root, with *nextOp* as child, and *currentSplit* as input. The *buildPhaseIn* function creates the node and sets the required properties according to

Algorithm 4.1 *splitPipeline* function

```
1:  function SPLITPIPELINE(node, currentSplit)

2:      validate(node)
3:      nextOp ← node.getLastChild()

4:      if isNonBlocking(node) then
5:          if isDataAccess(node) then
6:              instantiateForBind(node)
7:          end if
8:          bindVariable(currentSplit, node)
9:          return splitPipeline(nextOp, currentSplit)
10:     end if

11:     if isGroup(node) or isSort(node) then
12:         phaseOut ← buildPhaseOut(node)
13:         replaceNextOp(node.getParent(), phaseOut)
14:         phaseIn ← buildPhaseIn(node)
15:         addChild(phaseIn, nextOp)
16:         newSplit ← createSplit(phaseIn, currentSplit)
17:         return splitPipeline(nextOp, newSplit)
18:     end if

19:     if isJoin(node) then
20:         leftSplit ← splitPipeline(node.getChild(0))
21:         rightSplit ← splitPipeline(node.getChild(1))

22:         if isNotEmpty(currentSplit.left()) then
23:             replaceNextOp(currentSplit.getAst(), leftSplit.getAst())
24:             leftSplit ← currentSplit
25:         end if

26:         prepareJoinInput(node, leftSplit, 1)
27:         prepareJoinInput(node, rightSplit, 0)
28:         phaseIn ← buildPhaseIn(node)
29:         addChild(phaseIn, nextOp)
30:         newSplit ← createSplit(phaseIn, leftSplit, rightSplit)
31:         return splitPipeline(nextOp, newSplit)
32:     end if

33:     if isEnd(node) then
34:         return currentSplit
35:     end if

36: end function
```

the type of *node* and the variable table generated by *bindVariable*. The recursion then proceeds with *newSplit* as current split, continuing the splitting process from the *nextOp* node.

The translation of a `Join` is handled in lines 19-32. First, the algorithm invokes *splitPipeline* recursively on each join branch (located at child positions 0 and 1), resulting in two independent split trees, *leftSplit* and *rightSplit*. In case the function is being applied to a nested join, restriction §3 of Section 4.2 ensures that *leftSplit* will be empty, and hence the current split, which contains the incoming common input, should be considered as a left input to the newly constructed split. This is necessary because a top-down `Join` operator has three inputs to consider, but our runtime environment only supports binary joins. The standard Brackit compiler actually combines the common input with the left branch to form a binary physical join operator, but this process would not be trivial in our MapReduce environment.

The condition of line 22, namely the existence of a left input in the current branch, tells whether a nested join is being translated. Since *currentSplit* was created by a previous recursive invocation, the fact that it already has an input tells that *node* is not the root of the original AST, and because of restriction §2, we know that the parent operator must have been a join. Therefore, we can safely substitute *leftSplit* with *currentSplit*, but the `PhaseIn` operator which was generated previously must be moved to the new left split, which is done in the *replaceNextOp* function.

Once the left and right input splits of the join have been properly assigned, lines 26 and 27 invoke a preparation procedure on each of them. The procedure is responsible for creating artificial `LetBind`s which represent the join key and tag—discussed in Section 4.1—explicitly as variables. These two variables are then registered as grouping keys for the Shuffle procedure in a `PhaseOut` operator which is created and appended to the end of the join branch. The steps that follow are similar to the `Group` and `Sort` translation, except for the creation of the new split, which takes two inputs as parameters.

The recursive process finishes when the `End` operator is reached in line 33, in which case the current split is simply returned as root of the split tree.

4.4 Job Generation

The last step in mapping FLWOR pipelines to MapReduce computations is *Job Generation*, which takes a split tree as input and generates MapReduce job descriptions. These descriptions can then be submitted to a cluster for execution. Since a pipeline may require multiple MapReduce jobs for its execution, a list of job descriptions is generated. One may argue that a tree of jobs would be more representative, but considering the MapReduce execution model, it would not bring any advantage. This is because neither the parallel execution nor the pipelining of jobs brings significant performance benefit, since the processing power of the cluster is already exploited with the data-parallelism within a single job. Therefore, the list approach suffices as representation of MapReduce computations, and it makes the task of job execution much simpler.

Job Specification

Each job in the list is represented by a `Job` object, whose attributes we now discuss. The attributes we use here are very similar, but not identical, to the ones used by the Hadoop framework. For simplification and abstraction purposes, we provide a slightly modified description, which is illustrated in the class diagram of Figure 4.6. The class `XQJob` is an extended version which includes XQuery-specific information. It will be discussed later on.

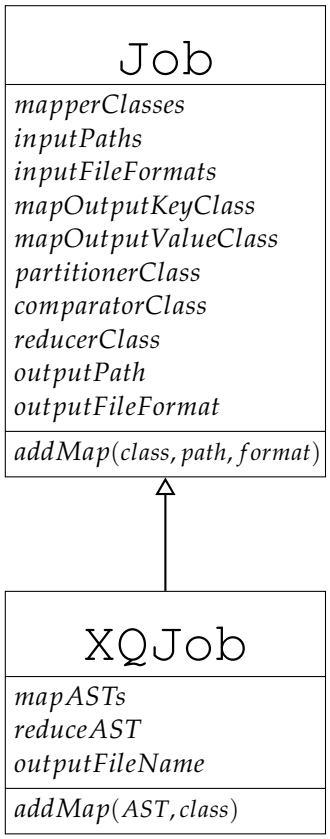


Figure 4.6: Class diagram for MapReduce job descriptions

The first attribute consists of a list of mapper classes. In the Hadoop framework, the user specifies the Map and Reduce functions by providing an implementation of the `Mapper` interface, in which the overridden *map* method implements the Map function. The same technique is used for the Reduce specification. The class approach allows more flexibility than simply providing functions, because the developer may keep internal state between Map invocations in the same task[2]. Because of the extension that allows multiple Mappers, classes are specified in the list *mapperClasses*. Accordingly, the input path and the format used for encoding are also specified for each mapper class, in the lists *inputPaths* and *inputFileFormats*, which have the same dimension as *mapperClasses*. For convenience, Map specifications are added using

[2]Recall, a task is a process which works on a partition of the data input, invoking the Map or Reduce function on each key-value pair.

the auxiliary *addMap* method of the `Job` class, which allows specifying the three parameters at once.

To complete the specification of the Mapper phase, the attributes *mapOutputKeyClass* and *mapOutputValueClass* specify the type of pairs generated by the Map function. These correspond to the K_2 and V_2 spaces introduced in the computational model of Section 2.3. The types K_1, V_1, K_3, and V_3 are specified in the mapper and reducer classes, using type parameters. Our framework makes use of three kinds of mapper and two kinds of reducer classes, each one with a different type parameter signature. We explain these classes in Section 4.5.

The Shuffle phase can be parameterized by partition and comparison functions. These are specified by the attributes *partitionerClass* and *comparatorClass*, using the same method-override technique as the mapper and reducer classes. The partition function is applied to generate the partitions of the key space K_2, given a desired number of partitions. In most cases, a built-in hash partition mechanism is used, but the user may specify custom value-based partition functions as well as mechanisms which take data locality into account. The comparison function determines the sort order of values in K_2. Our framework makes use of this customization in the implementation of joins, where the comparison must guarantee that tuples of the right input appear before those of the left one, and the partitioning must ignore the tag and send all tuples with the same join value to the same Reducer task.

To specify the Reducer phase, we make use of a similar triple as used for the Mapper: *reducerClass*, *outputPath*, and *outputFileFormat*. However, there is no need to use lists, since only one Reduce function is executed per job.

For each attribute, the `Job` class also provides *get* and *set* methods, which we omit here for brevity.

Up to now, all attributes introduced are part of the generic `Job` class, which is used for arbitrary MapReduce jobs. Our framework provides a derived class `XQJob`, which abstracts the definition of Map and Reduce functions using ASTs. The attribute *mapASTs* contains a list of AST pipelines, each of which comes from a split and is compiled and executed by one of three generic mapper classes. Therefore, jobs that execute XQuery programs need only the AST as parameter, and there is no need to implement specific mapper or reducer classes with overridden methods. Similarly, the *reduceAST* attribute is used to specify the Reducer phase. Lastly, the attribute *outputFileName* specifies the path in the distributed filesystem in which the final query results will be stored.

Job Generation Algorithm

The algorithm for Job Generation constructs a list of `XQJob` objects and sets their attributes accordingly. This task is relatively straight-forward, because the split tree already provides the job structure and parameters necessary to perform the MapReduce computation. Algorithm 4.2 basically traverses the split tree recursively, passing a list of `XQJob` objects which is gradually built as the recursion proceeds. At the end, this list is returned as the output of the Job Generation process, and the jobs are ready for execution.

Since each split corresponds to either a Map or a Reduce function, each invocation of *generateJobs* sets the parameters of either a Mapper or Reducer phase in a current job. How

Algorithm 4.2 Job Generation algorithm

```
1:  function GENERATEJOBS(split, jobList, parentJob)

2:      job ← null
3:      if isEmpty(split.left()) then
4:          mapperClass ← null
5:          if jobList.size() = 0 then
6:              job ← createXQJob()
7:              mapperClass ← XQFinalMapper
8:              job.setMapOutputKeyClass(OUTPUT_KEY)
9:              job.setMapOutputValueClass(OUTPUT_VALUE)
10:             job.setOutputFileFormat(OUTPUT_FORMAT)
11:             job.setOutputPath(OUTPUT_PATH)
12:             jobList.add(job)
13:         else
14:             job ← parentJob
15:             mapperClass ← XQMapper
16:             job.setMapOutputKeyClass(XQGroupingKey)
17:             job.setMapOutputValueClass(TupleWritable)
18:         end if

19:         job.addMap(split.getAST(), mapperClass)
20:     else
21:         job ← createXQJob()
22:         if jobList.size() = 0 then
23:             job.setReducerClass(XQFinalReducer)
24:             job.setOutputPath(OUTPUT_PATH)
25:         else
26:             job.setReducerClass(XQMidReducer)
27:             job.setOutputPath(getTempFile(jobsList.size()))
28:             parentJob.setMapOutputKeyClass(XQGroupingKey)
29:             parentJob.setMapOutputValueClass(TupleWritable)
30:             parentJob.addMap(null, XQIdMapper)
31:         end if

32:         job.setOutputFileFormat(OUTPUT_FORMAT)
33:         job.setReduceAST(split.getAST())
34:         jobList.add(job)

35:         generateJobs(split.left(), jobList, job)
36:         if isNotEmpty(split.right()) then
37:             generateJobs(split.right(), jobList, job)
38:         end if
39:     end if

40: end function
```

the parameters are set, and which phase is being configured depends on the position of the current split in the tree. The split will be mapped to a Map function if it is a tree leaf, which corresponds to the condition of line 3. Note that we do not need to check if the right split is also empty, since it only occurs in joins, in which case the left split will also be present.

The first and most trivial case corresponds to a tree with a single node, which is generated for a FLWOR pipeline without blocking operators. In this case, handled in lines 7-12, only one job is generated, and the complete pipeline is executed in a Map function, while the Reducer phase is empty. We refer to this type of Mapper phase as *final Mapper*, which is represented by the class `XQFinalMapper`. Here, the map output key and value classes are set according to the global variables OUTPUT_KEY and OUTPUT_VALUE, which actually correspond to the format of output files, since a final Mapper is being produced. Similarly the output path and file formats are set accordingly. We rely on global variables because, as mentioned earlier, the input and output formats are irrelevant for our query mapping purposes and thus kept abstract. Finally, the created job is added to the list and the AST and mapper class are set in line 19. The process then finishes and no recursive calls are necessary, since the only job was already generated and added to the list.

Because lines 14-17 are only executed when a job was already added to the list, we skip to line 20, where a Reducer phase is generated. Similar to the final Mapper, a root split which has input generates a *final Reducer*, whose class and output paths are set on lines 23 and 24. Then, starting on line 32, the proper output format and AST are set, and the job is added to the list. The final step is then to invoke the function recursively on the left and, if available, right inputs. Additionally, the current job is passed as the *parentJob* parameter of the invoked function. The parent job is used to generate non-final Mappers, as we explain below.

If the split is not a root (i.e., there is already a job in *jobList*), execution starts on line 26, where a *mid Reducer* is generated. In this case, the output path is some temporary file which will be consumed by the last job added to the list. Mid Reducers are simply Reducer patterns located between two Shuffles (i.e., the edge that connects parent and child splits). This means that an *identity Mapper*—which was introduced in Section 4.1—must be added to the job to be executed after the current one, namely *parentJob*. The identity Mapper is set by specifying the `XQIdMapper` class together with an empty AST. Furthermore, the key and value types produced by the Map function are adjusted accordingly. `TupleWritable` corresponds to a FLWOR tuple, while `XQGroupingKey` contains variables which are used as grouping keys, as explained in Section 4.1. From line 32 onwards, the process is the same as for a final Reducer.

Now, we go back to the generation of a Mapper phase, on lines 14-17. In this case, instead of generating a new job, we can simply add the current split AST as a Mapper to the parent job. Note that, except for the trivial final Mapper case, this is the general strategy for specifying a Mapper, namely by adding a Map AST to the parent job in a nested function invocation. We already covered the final Mapper case, which is always set together with a mid Reducer. In this case, however, we have reached a leaf split, which means a standard Mapper—performing data access on collections—must be generated. It is represented by the `XQMapper` class, and just like mid Reducers and identity Mappers, it generates `XQGroupingKey` and `TupleWritable` as key and value type, respectively.

Note that the algorithm does not set the *partitionerClass* and *comparatorClass* attributes,

which occur when mapping joins or, equivalently, when a split has both left and right inputs. These attributes are set internally in the *addMap* method when the list of Mapper classes already contains one element. Whenever a job has two Mappers, it is safe to assume that it performs a join, since it means that *generateJobs* was invoked on both lines 35 and 37.

The list of `XQJob` objects generated by the algorithm is then submitted to the MapReduce cluster for execution, a process which we describe in the following section.

4.5 Execution in Hadoop

We describe the MapReduce execution process by dividing it into three steps: constructing and shipping jobs to worker nodes, compiling the AST to an executable Brackit pipeline in each node, and finally executing the pipeline upon each invocation of the Map and Reduce functions.

Step One: Shipping Jobs to Workers

The front-end of our framework to users is the `XQueryDriver` class, which runs in the master node and is responsible for performing the whole mapping process from an XQuery string to job specifications, further invoking each job and controlling the execution. The driver has a *main* method which is invoked with three parameters: an XQuery expression, a metadata file, and a path in the distributed filesystem where the results will be stored. The XQuery expression is parsed and optimized using the Brackit engine, and the generated AST is then fed to our Pipeline Splitting and Job Generation processes, resulting in a list of `XQJob` objects. The metadata file contains the schema of the tables which are made available in XQuery as collections and physical information, such as their locations in the distributed filesystem and their file formats. The metadata information is used throughout the compilation and execution process, including the relational optimizations of the extended Brackit engine, discussed in Section 3.5.

The Hadoop `Job` class, which our `XQJob` extends, is not submitted to the worker nodes. Instead, the Hadoop runtime extracts all parameters of the job description and writes them in a *configuration* object, which contains key-value pairs of strings. In a typical MapReduce program, the user provides their own implementation of the Mapper and Reducer classes, and thus the name of these classes is all the worker nodes need to execute the job. Our framework, on the other hand, uses generic classes which must be parameterized with an AST. In this case—and in any other case where the mapper or reducer classes are parameterized—we must implement a serialization mechanism that provides the workers with the parameters they need, namely the AST to be compiled and executed. Our approach consists of serializing the AST to a string representation and setting it in the configuration object with a corresponding key.

Once the complete job information—including the ASTs—is written to the configuration object, is is submitted to available workers, which will create and execute instances of a Mapper or Reducer class. Before it starts processing input key-value pairs, an instance retrieves the corresponding AST from the configuration and moves on to step two of the execution process,

which we discuss in the following. Note that the two steps that follow are executed by every worker in the cluster.

Step Two: Compiling an AST into a Brackit pipeline

As mentioned earlier, we make use of the standard Brackit translator to generate an executable pipeline from an AST. The translator is the compiler component which translates the optimized logical plan into the physical version. Its result is a tree of `Operator` objects, which can be executed by providing an input tuple stream and invoking *open-next-close*. However, we must extend the translator to handle the additional `PhaseIn` and `PhaseOut` operators. The extended translator is called `MRTranslator`, whose class diagram is shown in figure 4.7. The base class which it derives is `TopDownTranslator`, which compiles the top-down FLWOR pipelines.

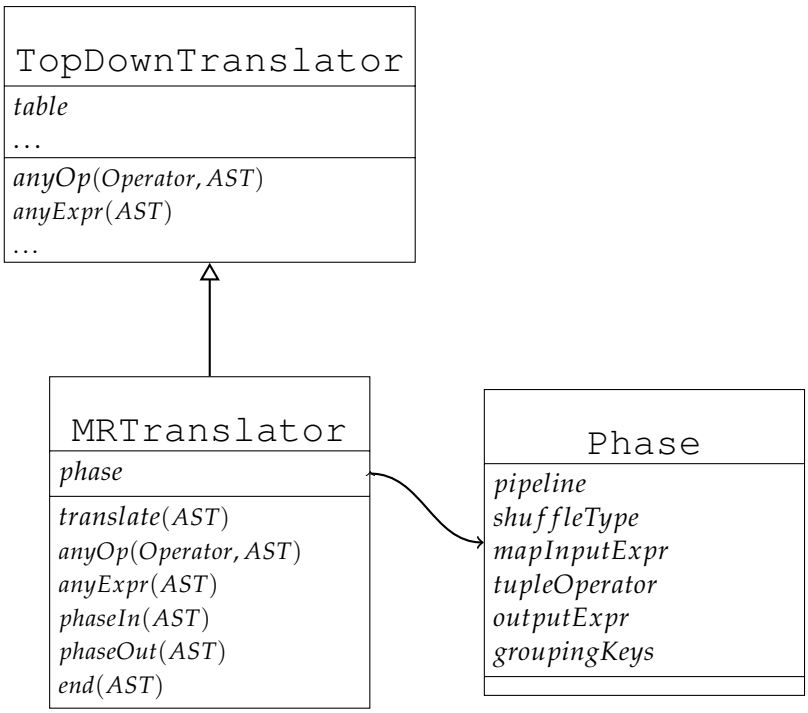


Figure 4.7: Class diagram for `MRTranslator`

The basic strategy of a translator is to traverse the AST, invoking a particular method for each operator or expression node. These methods produce the `Operator` and `Expression` objects accordingly, invoking the translation recursively on the node's children. Standard XQuery translation always results in an `Expression` object as root, which in case of FLWOR expressions corresponds to a `PipeExpr`. However, because our ASTs correspond to pipeline sections, the result of translation in `MRTranslator` is an `Operator` object.

An `Operator` tree is enough to execute pipelines using the Brackit engine, but our approach includes additional information related the MapReduce execution model. Such information is contained in the `PhaseIn` and `PhaseOut` operators, which essentially represent

a bridge between the two execution models. Instead of extending the physical operators of Brackit, we provide a `Phase` class, which wraps an `Operator` tree together with such additional MapReduce-related information. Therefore, the method *translate*, which performs the translation in `MRTranslator`, takes an AST as parameter and delivers an instance of `Phase`. Using this approach, there is no need to provide physical implementations of `PhaseIn` and `PhaseOut`—a basic tuple-wrapper operator suffices, as we show later.

The *translate* method initializes the *phase* attribute with an empty instance and invokes *anyOp* to translate the AST into an operator tree. During the translation process, the additional MapReduce-related information in this attribute is set accordingly when the `PhaseIn`, `PhaseOut`, and `End` operators are visited. The method *anyOp* calls an operator-specific method depending on the type of the node. In our translator, *anyOp* is extended to invoke the *phaseIn* and *phaseOut* methods when the homonym operators are visited. Furthermore, we override the *end* method, which translates the `End` operator.

Similar to *anyOp*, the *anyExpr* method is used to translate an expression AST into a corresponding `Expression` object. This method is also extended to support the special `ItemExpr`, which is introduced in the instantiation of a data-access `ForBind`, as mentioned in Section 4.3. It is a simple placeholder for an XDM item, which is populated when fetching input key-value pairs during a Mapper phase, as explained later in step three of execution. A reference to the `ItemExpr` object is kept in the *mapInputExpr* attribute.

Algorithm 4.3 describes the methods *translate*, *anyOp*, and *anyExpr*. Here, the special *super* object is used to invoke methods of the base class `TopDownTranslator`.

The *phaseIn* method is shown in Algorithm 4.4. The first task is to extract the *shuffleType* attribute from the `PhaseIn` AST node and set it accordingly in the *phase* object, which is done in line 2. The `PhaseIn` operator also contains a list of variables which were bound by previous pipeline sections. For each variable, the translator performs the binding explicitly in the variable table used by `TopDownTranslator`, which is contained in the attribute *table*. At the end, the *phaseIn* method invokes *anyOp* recursively on its child node, returning the delivered operator as result. As input to the next operator, it uses the *tupleOperator* attribute of the `Phase` object. This operator simply wraps a sequence of tuples which is iterated when *next* is called. It always occurs as the first operator executed in a Reduce function, delivering the tuples passed as value in the function's key-value parameter. We discuss this process later on in step three. Note that `PhaseIn` is always the first operator in a pipeline section, and as such it generates a `TupleOperator` which accordingly does not have any input and is the first operator to be executed. Analogously to *mapInputExpr*, the *tupleOperator* attribute serves as a placeholder, which during execution is populated with tuple streams coming from the Reduce function's parameters.

The *phaseOut* method, described in Algorithm 4.5, is responsible for setting the tuple field indexes which correspond to the grouping key of a Shuffle. The children of the `PhaseOut` node are variable references which form the grouping key. Therefore, the method iterates over each variable and obtains the corresponding tuple index from the variable table. The resulting indexes are then set in the *phase* object in line 8. At the end, the method simply returns the operator which it received as parameter. Note that there is no effect on the operator tree, and because a `PhaseOut` always finalizes a pipeline section, there is no need to invoke *anyOp*

Algorithm 4.3 *translate*, *anyOp*, and *anyExpr* methods of `MRTranslator`

```
1: function TRANSLATE(node)
2:     phase ← createPhase()
3:     op ← anyOp(null, node)
4:     phase.setPipeline(op)
5:     return phase
6: end function

7: function ANYOP(input, node)
8:     if isPhaseIn(node) then
9:         return phaseIn(input, node)
10:    end if
11:    if isPhaseOut(node) then
12:        return phaseOut(input, node)
13:    end if
14:    if isEnd(node) then
15:        return end(input, node)
16:    end if
17:    return super.anyOp(node)
18: end function

19: function ANYEXPR(node)
20:    if isItemExpr(node) then
21:        itemExpr ← createItemExpr()
22:        phase.setMapInputExpr(itemExpr)
23:        return itemExpr
24:    end if
25:    return super.anyExpr(node)
26: end function
```

Algorithm 4.4 *phaseIn* method

```
1: function PHASEIN(input, node)
2:     phase.setShuffleType(node.getProperty("shuffleType"))
3:     for var in node.getProperty("vars") do
4:         table.bind(var)
5:     end for
6:     return anyOp(phase.getTupleOperator(), node.getLastChild())
7: end function
```

recursively, since there is no next operator.

In the case of a final Mapper or Reducer, the pipeline section finishes with an `End` operator, whose translation is handled by the *end* method of Algorithm 4.6. The method simply compiles the expression of the FLWOR `return` clause and sets it as the *outputExpr* attribute of the `Phase` object. This expression will be used during execution of a Map or Reduce function to construct a `PipeExpr` and output the query results, as shown in step three later on.

The retrieval of an AST from the job configuration and its compilation into a `Phase` using `MRTranslator` are executed when initializing a Mapper or Reducer class, which stores the

Algorithm 4.5 *phaseOut* method

```
1: function PHASEOUT(input, node)
2:     groupingKeys ← emptyList()
3:     for child in node.getChildren() do
4:         var ← getVariableName(child)
5:         index ← table.getVarIndex(var)
6:         groupingKeys.add(index)
7:     end for
8:     phase.setGroupingKeys(groupingKeys)
9:     return input
10: end function
```

Algorithm 4.6 *end* method

```
1: function END(input, node)
2:     outputExpr ← anyExpr(node.getChild(0))
3:     phase.setOutputExpr(outputExpr)
4:     return input
5: end function
```

compiled phase in an attribute. Once initialized, the Hadoop framework starts fetching input key-value pairs, invoking the Map or Reduce function on each of them. The invocation of Map/Reduce corresponds to step three of the execution, which is what we discuss on the following.

Step Three: Executing Pipelines in Map/Reduce Invocations

The execution of a compiled `Phase` object depends on the type of Mapper or Reducer which is being executed. In general, this task consists of three steps: (i) preparing the pipeline by populating the placeholders in `Phase` with the key-value parameter of the Map/Reduce function; (ii) executing the pipeline or evaluating a pipe expression based on it; and (iii) producing one or more key-value pairs as output. In the following discussion, we go through these three sub-steps and explain how each of them works in each of the Mapper and Reducer classes.

The pipeline-preparation step depends on whether a Map or Reduce is being executed. In the case of a Map (i.e., classes `XQMapper` or `XQFinalMapper`), the method *prepareMapPipeline* of the `Phase` object is invoked. Because a Mapper phase only occurs when fetching items from an input collection, the task is to populate the `ItemExpr` expression referenced by the *mapInputExpr* attribute. As discussed in Section 4.3, this expression is introduced by the instantiation technique, which unrolls a data-access `ForBind` into a `LetBind` to an individual item. `ItemExpr` is a simple wrapper expression, which simply delivers a predefined item when evaluated. In our case, this item is the one that is fetched from the collection and passed to the Map function as the value in the key-value-pair argument.

In the case of a Reduce, the method *prepareReducePipeline* is invoked to perform the finalization step of a `Group`, `Join`, or `Sort` operator. Again, the goal is to populate the placeholder attribute *tupleOperator* with the resulting tuples. Similar to `ItemExpr`, a `TupleOperator` simply wraps a sequence of tuples, which is iterated and whose individual items are delivered

when invoking the *next* method of the open-next-close protocol. However, as mentioned in Section 4.1, we cannot simply use the list of tuples received from the previous Shuffle phase, because some blocking operators actually require a finalization step.

In the Hadoop environment, the list of tuples is actually represented by an instance of the `Iterable` interface, which allows iteration using the methods *hasNext* and *next*. This abstracts away from the actual list data structure, allowing the framework to freely decide between in-memory data structures or external cursors, depending on the input size. To implement the finalization steps of blocking operators, we provide customized `Iterable` classes, which wrap the iterator provided by the Shuffle and delivers the tuples as specified by the blocking operator semantics.

In the case of a `Group`, the finalization step consists of a merging process, which concatenates the values of each non-grouped variable into a single sequence, delivering a single tuple, as specified by the `group by` semantics in Section 3.2. This process is performed by the `PostGroupIterable`, whose effect on the tuple stream is shown in Figure 4.8, which shows the finalization step of tuples grouped by variables $a and $c. Note that the iterator always delivers a single tuple, and hence there is no actual iteration being carried out when *next* is invoked. This means that `PostGroupIterable` can be implemented using a "singleton iterator", which always contains a single element.

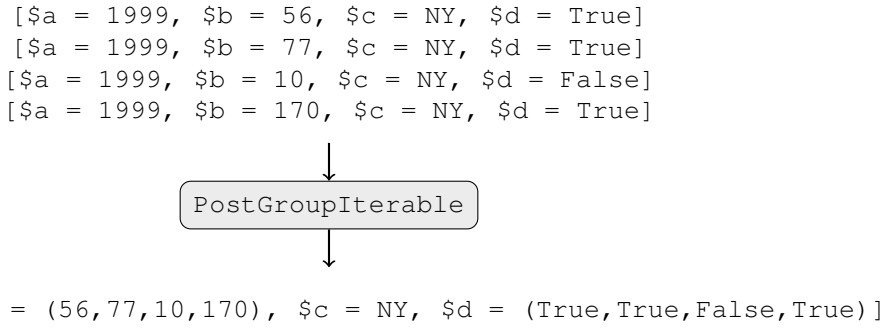


Figure 4.8: Effect of the `PostGroupIterable` iterator

In the finalization step of a `Join`, we must iterate over the tuples and separate them into two different tuple streams depending on the tag value. Because the tuples were already matched by the Shuffle, a simple nested-loops procedure produces the cartesian product of the streams, which gives the result of the join operation. This logic is implemented by the `PostJoinIterable`, whose behavior is illustrated in Figure 4.9. Here, we have 4 tuples, two from each input branch, as indicated by the *tag* variable which was bound artificially by the `PhaseOut` operator. One of the branches bound the variable `$a`, while the other one bound `$b`. For a better visualization, we omit the artificial join keys on the output, since they are not used anymore and may in fact be projected out.

A `Sort` does not require any finalization step, since the tuples are already delivered in sort order by the Shuffle. Hence, it simply delivers the tuples provided in the key-value pair.

Once the pipeline is prepared, we move on to step (ii), which is the pipeline execution. This step depends on whether the Mapper or Reducer class is of type "final". In the case of

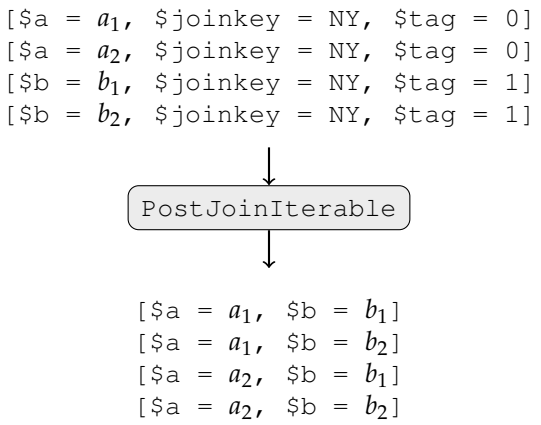


Figure 4.9: Effect of the `PostJoinIterable` iterator

`XQFinalMapper` and `XQFinalReducer`, we have to produce the final results of the FLWOR expression, and hence they must evaluate a pipe expression, represented by the `PipeExpr` object. A pipe expression is constructed with a tuple stream and an expression which is evaluated for each tuple in it. In our case, we use the prepared pipeline to deliver the tuple stream and the *outputExpr* attribute as expression. The constructed pipe expression is then evaluated, and the resulting sequence is written to the output in step (iii) as the value component. The key used is implementation-defined, and in our case we simply write an increasing counter value.

In the case of `XQMapper` and `XQMidReducer`, we must execute the pipeline section and deliver the resulting tuples to the next phase, without evaluating a result expression. Therefore, the process consists of opening a cursor on the prepared pipeline and iterating over it, emitting each delivered tuple as an output key-value pair on step (iii). In this case, the key is of type `XQGroupingKey`—since the output will be grouped by a following Shuffle—, and the values are tuples. To build the key, the grouping indexes in the *phase* attribute are used. Note that, in the case of final classes, the cursor is internally created by `PipeExpr`, which is why we can simply evaluate the expression without worrying about iterating over the tuples.

Note that we did not mention the `XQIdMapper` class, because it only performs the trivial task of emitting the received key-value pairs directly as output. It does not even contain a `Phase` object.

Once the execution of a job is finished, the `XQueryDriver` instance starts the execution of the next `XQJob` object on the list, repeating steps (i), (ii), and (iii) for the new job. If there are no further jobs in the list, the execution of the query is complete, and the results are directly available in the file specified by the user as output path.

Chapter 5

Performance Analysis

The framework we developed in this thesis is mainly concerned with the feasibility of expressing analytical queries using XQuery and execute them in the MapReduce middleware. As such, our focus is primarily on the proof of concept. Because we tried to abstract issues of data storage, network traffic, memory management, algorithms for physical operators, etc. Our prototype does not perform optimally yet in terms of execution time. This is confirmed by the experiments in Section 5.2, which compare the execution times of our prototype against Pig[1], a state-of-the-art framework for query processing in Hadoop.

Despite the performance not being the main focus of our current work, we discuss the major factors which impact performance of query processing in MapReduce in Section 5.1. Once such features are incorporated in our prototype, it is very likely to catch-up with the performance of existing frameworks, of which we provide an overview in Section 6.2.

5.1 Optimization Opportunities

The opportunities for optimization in a MapReduce-based query processor are diverse, and reach from standard query processing techniques and MapReduce-related optimizations to efficient storage modules and binary encoding/compression schemes. The following discussion provides techniques that would potentially provide a significant performance boost to our framework. Rather than falling specifically into one of the categories above, the techniques presented overlap concepts of the different areas, but we maintain our focus on the MapReduce scenario.

Query Optimization

Since our framework is based on the standard, single-machine query compiler of Brackit, there are of course the optimization techniques applied at the AST level. Brackit applies algebraic rules such as filter push-down, join recognition, unnesting, etc. to produce more efficient plans. Such rules are independent of execution environment and generally bring improvements in both database and MapReduce scenarios. These techniques are out of the scope of this thesis,

[1] http://pig.apache.org/

and thus we assume that the query plan provided to our framework is already algebraically optimized. A description is available in [4].

Despite the applicability of such generic rules, the awareness of a MapReduce execution environment may allow the optimizer to produce more efficient plans, given the characteristic optimization goals. A query optimizer is usually equipped with a variety of rules, whose impact on execution time is closely related to the runtime environment. For example, many optimization techniques are based on the fact that hard disks have a significant discrepancy between sequential and random read speeds, and thus an operation that fetches few data randomly may be significantly slower than one that scans through large datasets sequentially. In the case of flash memory, however, there is in principle no difference between random and sequential reads, so the optimizer should always prefer the more selective operation. The situation is analogous in MapReduce, whose cluster architecture implies different optimization criteria than in a single-node system, and hence its awareness is of great advantage for the query optimizer.

As an example, consider the case of an n-way join, which is normally implemented using a nesting of binary join operators. As shown in [3], a multi-join operator may actually be preferable in a MapReduce environment, since it incurs less transfer of data between workers. The same authors presented in [2] a cost model for the analysis of MapReduce algorithms, based on the assumption that the cost to ship data from one processor to another dominates the cost of actual processing. Such models are of key importance for algebraic query optimization, since it aims to reduce the time complexity of the resulting plan.

The ideas of cost-based query optimization also have great applicability in MapReduce. Besides the standard techniques used to predict selectivity of predicates (and navigation steps in the case of XML) based on statistics, the computational model of MapReduce provides new opportunities for cost-based rewriting, introducing variables such as the communication cost.

Key-Space Partitioning and Aggregation

As in any optimization model for parallel computations, the uniform distribution of data among workers is of crucial importance in MapReduce. Because workers operate on partitions of the input data in parallel, the execution time of a Mapper or Reducer phase is bounded by its slowest worker. This is usually not a problem for Mappers, since they are fed with equally-sized chunks of the input file on demand. In the case of a Reducer phase, however, the workload distribution is tightly connected to the partitioning of the key space. Using partitioning schemes based on hashing helps to cope with the situation of non-uniform distribution of keys, but it still suffers from skewed samples where a few values dominate the frequency histogram. Techniques have been proposed in the literature to provide better load balancing in the presence of skew [24]. These are very important for efficient query processing.

Other than skew, any partitioning mechanism based on attribute values (which includes hash- and range-based approaches) does not leverage data parallelism in situations where the values space is constituted of only a few points. Consider, for example, a group operation on a *gender* attribute. A direct implementation in MapReduce would use *gender* as Shuffle key, but that would result in only two worker nodes processing the complete dataset. Furthermore,

depending on the size of the dataset and on how the Shuffle is implemented, the job would not even be runnable, since the amount of data of a single key may not fit in a worker's local filesystem.

A crucial feature, which not only helps on those situations, but also improves grouping in general is to identify distributive aggregate functions [20]. They allow pairs of grouping-aggregation tasks to be executed gradually, in a hierarchical manner. Such functions could, for example, be applied at the Mapper side, feeding the Shuffle phase with partially aggregated results. To this end, the MapReduce framework allows the definition of a *combiner* function, which in many cases (including distributive aggregations) is the Reduce function itself. The scope of partial aggregation, however, is not limited to combiner functions. The query processor could take advantage of that and generate multiple Shuffle phases for a single group operator, resulting in a hierarchical execution. This would provide much better load balancing, even for the skewed partitions mentioned above.

If the aggregate function is not distributive, or if there is no aggregation being performed at all (e.g. a skewed join), it results in a more challenging problem. Because such cases are rather rare, even state-of-the-art distributed query processors fall short in providing high levels of parallelism. Nevertheless, it is definitely a problem to be exploited in future research.

Local Memory Management

Even when the key-space partitioning results in approximately equal partitions which can be processed by single nodes, the amount of values associated to a single key may not fit in main-memory. As mentioned in Section 4.5, the list of values passed as parameter to a Reduce function invocation is represented as an `Iterable` instance, whose contents may actually reside in external storage. Hence, like in database query-processing, the blocking operators must be able to implement external algorithms for datasets larger than main memory. In our framework, only non-blocking operators may occur in a pipeline section, but the problem must still be handled in the finalization steps of the `Join` and `Group` operators.

In the case of a join, we must read all tuples with a specific tag value and buffer them. This buffered content may then be used as the inner table in a nested-loops process that performs the cartesian product that finalizes the join. Hence, it is crucial that the `PostJoinIterable` class can write these buffer contents to disk in case a memory-usage threshold is reached. In extreme scenarios, the buffer may not fit in the machine's local disk, in which case the distributed filesystem could be used to store such intermediate results.

The same situation occurs in `PostGroupIterable`, which merges all values of a variable in a tuple stream into a single sequence. If a sequence is too large to fit in main memory, it must be buffered to disk.

This discussion also raises concern with respect to the pull style of evaluation with lazy sequences employed by Brackit. In contrast to the scenario mentioned above, most applications of the Map and Reduce functions operate on very few tuples—usually just one. In such cases, the overhead of creating lazy sequences and pipelining results with open-next-close may be significant when compared to the actual evaluation of the operators. It would thus be more effective to apply an eager, or push-based, evaluation strategy, in which each operator computes

its complete output before the next one is evaluated. Therefore, an execution engine which moderates between lazy and eager evaluation would be of great advantage—especially in the MapReduce scenario, where small numbers of input tuples per pipeline invocation occur more frequently.

Serialization

When tuples are shipped to worker nodes or written to a local buffer in disk, it is fundamental to apply efficient binary encoding schemes as well as compression techniques. The first reason why this is important is that it minimizes data transfer not only between main memory and disk (the traditional bottleneck in database systems), but also between worker nodes via the network. On the query level, it is also of great advantage to project tuples which will not be needed anymore, which drastically reduces the size of intermediate results. This is especially true in analytical computations, where datasets usually contain several attributes but queries only access a few of them.

The second reason is that an efficient encoding scheme allows certain operations—mainly comparisons—to perform on the byte level, without requiring the values to be de-serialized and constructed in main memory, whose cost completely dominates that of actually performing the operation.

Storage

The typical MapReduce job operates on data stored as plain text files in the distributed filesystem. This is also the only data format which we currently support in our framework. However, efficient query processing is only possible if an efficient storage module is provided underneath. The serialization aspects discussed above concern only intermediate results, but the same techniques apply to how data is stored and managed by the system.

Several proposals exist for storage modules which can be accessed by MapReduce, most of them arriving from the so-called NoSQL databases[2]. In the Hadoop community, the most prominent approach is BigTable [14], implemented by the HBase system[3]. BigTable provides a data model inspired by relational databases, but tailored to the needs of typical large-scale Web applications. Based on column-stores, its main focus is on sparse datasets with embedded schema evolution. Not only it is a natural fit to MapReduce, as the projects were developed together, but it also provides an environment similar to a database system, where datasets can be updated and point queries can be executed in BigTable, and large, scan-oriented workloads processed with MapReduce.

Another approach worth mentioning is that of HadoopDB [1], which combines single-node database management systems with MapReduce. The idea is to store and manage data in relational tables partitioned across the worker nodes in the cluster, employing Hadoop as a communication layer and query processor. Not only it leverages the storage efficiency of a DBMS but it also pushes operators such as projections and filters to the database layer, drastically minimizing the amount of data flowing between workers in the MapReduce execution. The

[2] An extensive list of such systems can be found at `http://en.wikipedia.org/wiki/NoSQL`

[3] `http://hbase.apache.org`

integration of database storage also opens opportunities to further optimizations such as index scans and buffer management.

5.2 Experiments

We performed experiments to demonstrate the feasibility of our approach using five different tasks, which represent basic patterns of analytical queries. For the first three tasks, we performed a measurement of the execution time for three dataset sizes: 100MB, 1GB, and 10GB. Due to the complexity of the remaining two tasks, the measurements could only be carried out on the 100MB and 1GB datasets. For larger datasets, it is crucial to apply the optimization techniques mentioned earlier, such as local memory management. Nevertheless, this constitutes a simple technical aspect of our prototype implementation, and it is not related to the concepts introduced in this thesis. The different dataset sizes demonstrate the scalability of our approach. Furthermore, for comparison purposes, we provide measurements of the equivalent queries in the Pig system.

We used data from the TPC-H benchmark [32], a standard benchmark for relational OLAP. For reference, the TPC-H schema is illustrated in Figure 5.1.

The experiments were performed on a single Intel Xeon X3350 quad-core CPU at 2.66GHz with 4GB RAM, running Ubuntu Linux 11.10 and Hadoop 1.0.1. The version of Pig used for the comparison was 0.10.0. Because performance in a cluster environment is not the main goal of our experiments, we only performed tests on a single machine. Note that this still allows a certain degree of data parallelism, since we may run up to 4 processes in parallel.

Filter Task

The filter task simply scans the `lineitem` table, which contains 6 million rows in the 1GB dataset, filtering the records by shipment date. Since there is no blocking operation involved, the generated MapReduce job consists of a single Mapper phase. Figure 5.2 shows the corresponding FLWOR expression as well as the measured execution times.

Aggregate Task

This task represents the standard kind of MapReduce job: grouping and aggregating. The query, shown in Figure 5.3 returns the sum of the final sale prices for each month and year. The task shows the major performance boost of partial aggregations with the *combiner* function, as implemented in Pig.

Join Task

This task performs a simple join between `lineitem` and `orders`, the largest tables in the dataset. It requires the execution of two Mappers—one for each table—and a single Reducer which performs the join computation. The query and its measured execution times are shown in Figure 5.4. Since the basic join mechanism is the same in both approaches, the execution times are not so discrepant as in grouping queries.

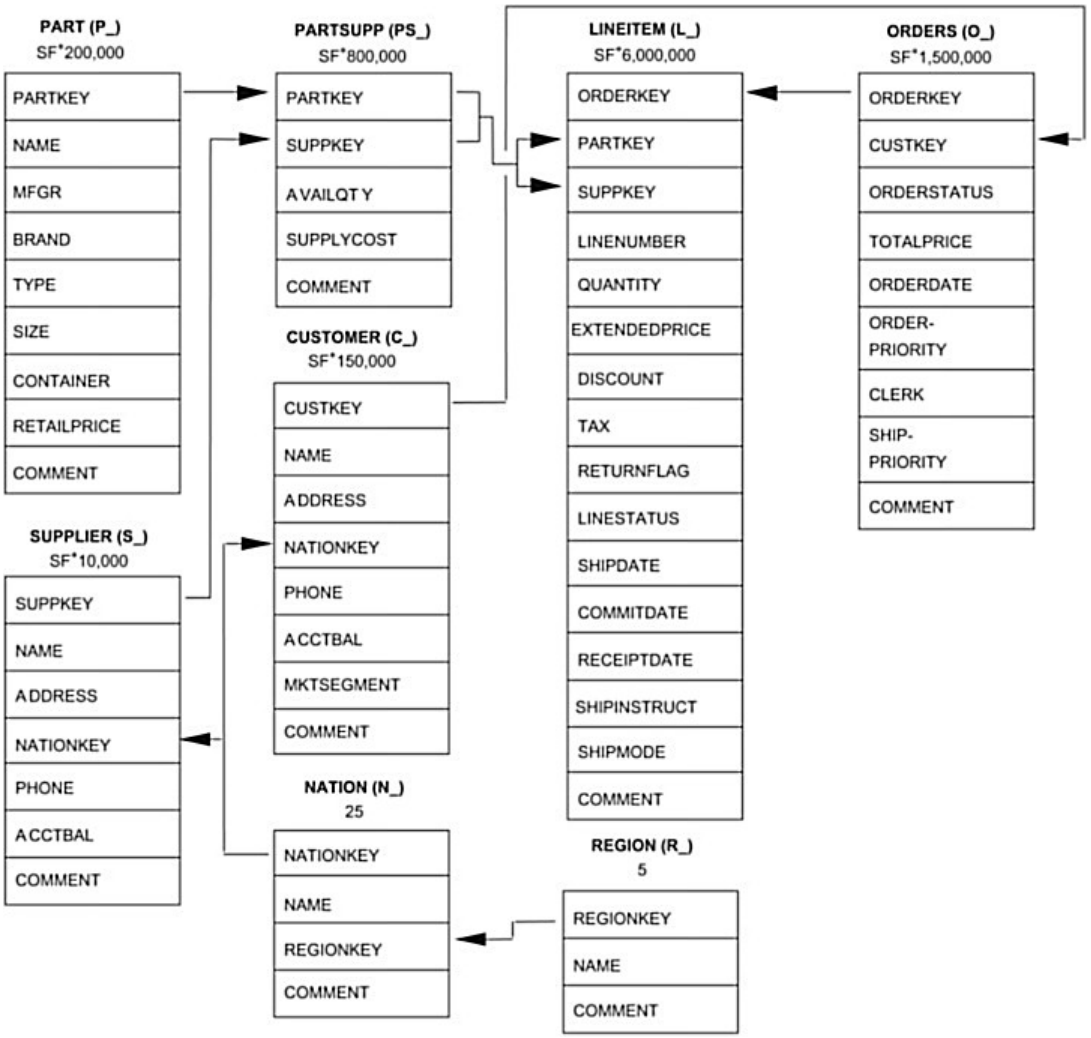


Figure 5.1: Schema of the TPC-H dataset

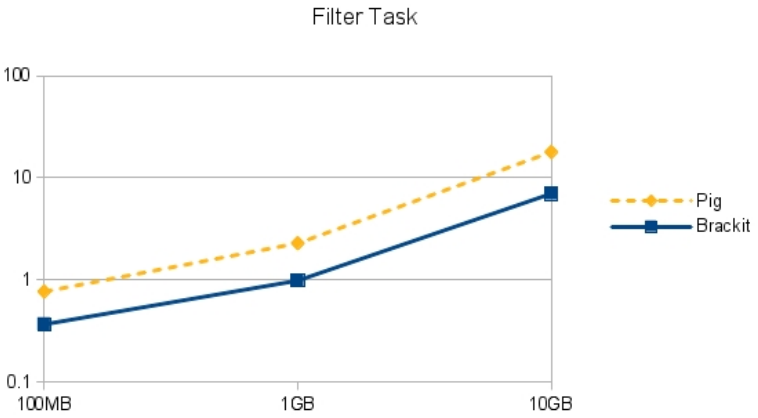
Join-Group-Sort Task

This task is an example of query with multiple blocking operators, namely a join like in the previous task, followed by the aggregation of sale prices in each month and year, finally ordered by the price. Accordingly, it generates three MapReduce jobs. The query and its measured execution times are shown in Figure 5.5.

Multi-Join Task

This task tests the multi-join capabilities of our framework, joining together all 8 tables of the TPC-H dataset. It results in a nesting of 7 join operations, which is accordingly executed using 7 MapReduce jobs. It is a very stressful task, even for the Pig framework. It showcases the optimization potential of a specialized MapReduce multi-join operator. The query and its measured execution times are shown in Figure 5.6.

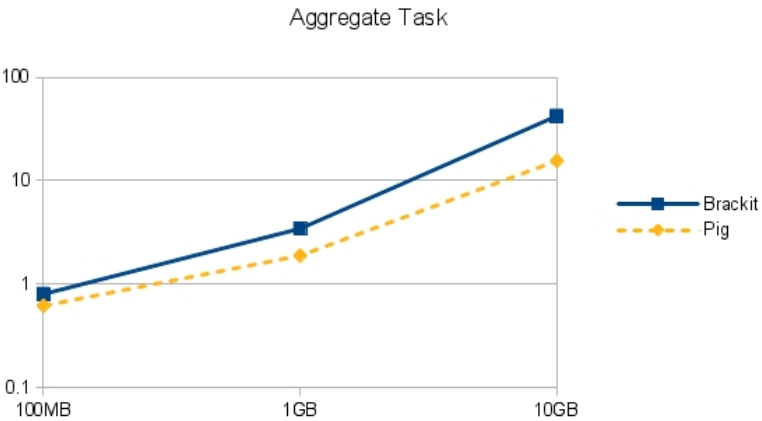
```
for $l in collection('lineitem.tbl')
where $l/shipdate > '1998-09-02'
return $l
```



	Brackit	Pig
100MB	22s	24s
1GB	59s	1m20s
10GB	7m	10m51s

Figure 5.2: Query and execution times of the filter task

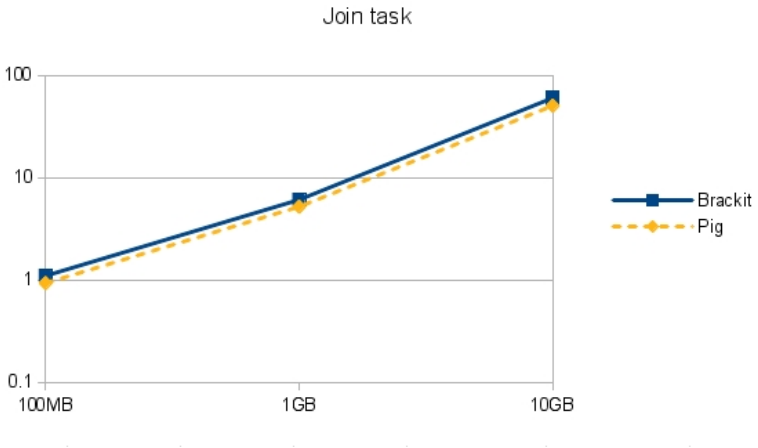
```
for $l in collection('lineitem.tbl')
let $month := substring($l/shipdate, 1, 7)
group by $month
return
  <result>
    <month>{$month}</month>
    <sum-price>{sum($l/extendedprice)}</sum-price>
  </result>
```



	Brackit	Pig
100MB	48s	37s
1GB	3m26s	1m53s
10GB	42m12s	15m31s

Figure 5.3: Query and execution times of the aggregate task

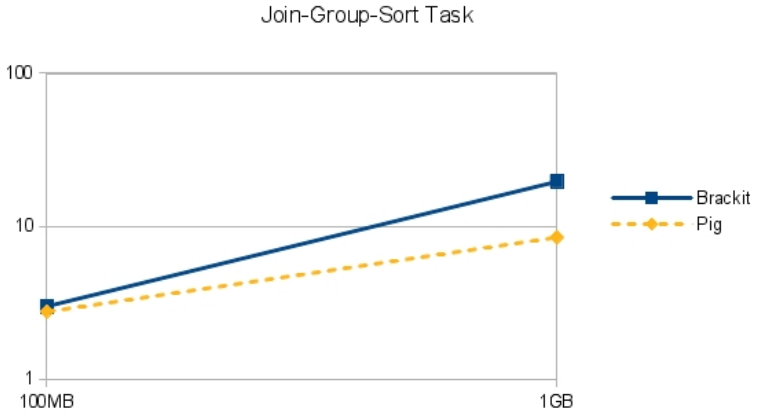
```
for $l in collection('lineitem.tbl')
for $o in collection('orders.tbl')
where $l/orderkey eq $o/orderkey
return
  <result>{($l, $o)}</result>
```



	Brackit	Pig
100MB	1m6s	56s
1GB	6m05s	5m12s
10GB	59m56s	50m15s

Figure 5.4: Query and execution times of the join task

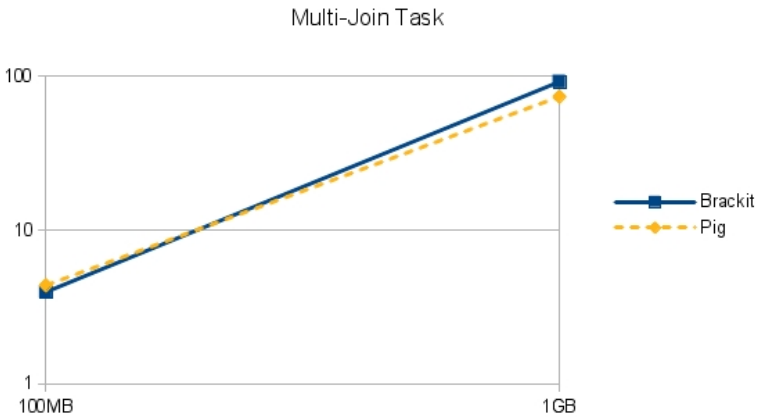
```
for $l in collection('lineitem.tbl')
for $o in collection('orders.tbl')
where $l/orderkey eq $o/orderkey
let $month := substring($l/shipdate, 1, 7)
group by $month
let $price := sum($o/totalprice)
order by $price
return
  <result>
    { $o/orderkey }
    <sum-price>{$price}</sum-price>
  </result>
```



	Brackit	Pig
100MB	3m	2m46s
1GB	19m37s	8m28s

Figure 5.5: Query and execution times of the join-group-sort task

```
for $l in collection('lineitem.tbl')
for $o in collection('orders.tbl')
for $c in collection('customer.tbl')
for $ps in collection('partsupp.tbl')
for $p in collection('part.tbl')
for $s in collection('supplier.tbl')
for $n in collection('nation.tbl')
for $r in collection('region.tbl')
where
  $l/orderkey eq $o/orderkey and
  $o/custkey eq $c/custkey and
  $l/partkey eq $ps/partkey and
  $l/suppkey eq $ps/suppkey and
  $p/partkey eq $ps/partkey and
  $s/suppkey eq $ps/suppkey and
  $c/nationkey eq $n/nationkey and
  $n/regionkey eq $r/regionkey
return <result>{($l,$o,$c,$ps,$p,$s,$n,$r)}</result>
```



	Brackit	Pig
100MB	3m57s	4m2s
1GB	1h32m	1h13m

Figure 5.6: Query and execution times of the multi-join task

Chapter 6

Conclusion

We presented a method for implementing XQuery processing in the MapReduce framework, concerned mainly with the mapping of operations from one computational model to the other. Therefore, this work represents a fusion of the two technologies, not only in the theoretical sense of the mapping rules, but also in the system layer, where we basically extended a full-fledged XQuery engine to support distributed processing.

This chapter concludes this thesis with an evaluation of the benefits and drawbacks of both XQuery and MapReduce in Section 6.1, which also analyzes related ideas that can be adapted to improve the presented solution. Section 6.2 will briefly discuss existing approaches for performing data processing in MapReduce using a higher-level language. Finally Section 6.3 provides a brief overview of the contributions made in this thesis and the open challenges in the field.

6.1 Criticism

Benefits and Drawbacks of XQuery

The rise of XML and XQuery in the field of data management was motivated primarily by the flexibility that they provide in comparison to the previously omnipresent SQL. The rigidness of relational schemas incurs excessive overhead for dynamic applications, which undergo constant change through the incorporation of new features and enduring improvement. This is especially true for large-scale Web applications, which are modified drastically as new features are added. Changes also occur in the traditional business scenario of relational OLTP and OLAP, because companies are constantly merged or incorporated by larger organizations. In such systems, rigid relational schemas represent a challenge for the task of data integration. The flexibility of XML and XQuery is of great advantage in these scenarios, but it obviously comes at a cost in both language and system layers.

At the language level, XQuery makes compromises on the clarity of operation semantics in favor of conciseness. Main examples of such compromises are atomization and general comparisons, which require the developer to understand the semantic of expressions depending on the kind of values on which they are applied. Other "hidden" requirements of XQuery, which not only may be confusing for a developer coming from SQL, but also represent a big challenge

for efficient query evaluation, are node identity and document ordering. These requirements often invalidate potential query results which would be acceptable by the application, but are incorrect according to the XQuery standard. It is reasonable for a data-centric XQuery engine to alleviate these requirements, or to turn them off by default, as they are important only for document-centric scenarios.

Further drawbacks of XQuery are related to its type system and data model. Because the only way to define structured (i.e., non-primitive) data types provided is by using nodes and sequences, it is quite cumbersome to define data structures such as records, maps, or lists. These must all be somehow encoded in an XDM sequence, which results in an unintuitive, non-standard representation. Furthermore, a node structure is inefficient when compared to lists, tuples, or maps which could be natively implemented in the compiler. Nevertheless, the definition of such types is concisely done using XML Schema [35], which provides a comprehensive set of rules for composition, inheritance, and constraint definition. Despite providing type safety, the standard is too strict and complex for many practical scenarios where data is primarily record-oriented.

For the particular end of bulk or set-oriented data-processing, XQuery provides FLWOR expressions as a single coarse-grained, all-in-one language construct. The processing a FLWOR expression, which in our case is done through pipelines, requires its own internal computational model, which is separated and hidden from that of other XQuery expressions. A tuple is a data item which exists only internally and implicitly during the evaluation of FLWOR expressions. This implies that we cannot consider a `for` or `where` clause as an isolated expression or function, which consumes and produces objects data model. This has significant implications for data-centric applications, such as the MapReduce mapping mechanism of this thesis.

The coarse granularity of FLWOR expressions requires a query engine to define its own internal constructs for processing them. In the Brackit engine, for instance, we make use of operator pipelines, which consume and produce tuple streams. A more elegant approach would be to specify tuples as part of the language's data model and then define each FLWOR clause as a self-contained expression type. This approach gives more expressiveness to the language and eliminates the need for compiler-specific logical plan representations. This is particularly interesting for the scenario of MapReduce query-processing, because it would allow us to specify standard XQuery expressions to be evaluated in Map or Reduce functions. Our current approach makes use of pipeline sections, which alone do not constitute valid XQuery programs.

A further drawback of the implicit FLWOR semantics is that it does not permit source-to-source compilation—a technique in which compiled programs are themselves valid XQuery expressions which use lower-level core functions. This technique is of great advantage for tuning and customizing the physical plan of a query using nothing but XQuery. With source-to-source compilation, extending the query engine is a simple matter of defining new functions and adjusting parameters.

Data generation in FLWOR expressions is also inflexible. Because tuples are not valid XDM objects, a FLWOR is not allowed to simply return its tuple stream as a result, or store it directly in an output file. This means that a `return` clause often makes use of superfluous node constructors to produce valid result. In may examples throughout this thesis, for example, a

FLWOR expression generates a `<result>` element which serves the unique purpose of encapsulating the output of a single tuple. Furthermore, the FLWOR syntax does not allow for a bifurcation of a tuple stream into independent pipelines. This scenario is especially useful in data-intensive analytical workloads, where several tasks share the same initial steps. Consider, for instance, a sequence of joins and filters which produces a standard dataset—similar to a relational *view*—that is used to produce different analyses. Such functionality, which in some frameworks is supported by a *tee* operator[1], cannot be easily reproduced in XQuery.

Benefits and Drawbacks of MapReduce

MapReduce has achieved great popularity in both industry and research communities. It is a pioneer approach to large-scale data-processing with distribution transparency, which differentiates from the long-existing parallel database systems for providing more generality and flexibility. As discussed in Section 2.1, MapReduce was not initially designed for query-processing workloads, as its focus lies on simple transformation and aggregation tasks. There are thus some drawbacks in applying MapReduce for query processing, and these are discussed below.

Computations in MapReduce are expressed as a triple of Mapper-Shuffle-Reducer patterns, which are strictly executed in this sequence. Therefore, data-processing patterns that are mapped to multiple Shuffle phases actually require multiple jobs, and thus multiple instances of such triples. The problem is that this composition of triples is overkill, since all we need is the capability to execute a sequence like Mapper-Shuffle-Reducer-Shuffle-Reducer-etc. As discussed in Section 4.4, we overcome this restriction by introducing identity mappers, but it still incurs a major performance bottleneck, since intermediate results are written to the (usually slower and replicated) distributed filesystem. Furthermore, a partitioning of the key space cannot be applied to the Reducer output as it is done after a Mapper, which is another reason why we need identity mappers. If the framework would allow for arbitrary composition of the processing patterns, instead of the fixed triples, a query plan would be not only more naturally mapped to the MapReduce model, but also executed more efficiently.

A second major drawback of the MapReduce framework is that it does not gracefully support operators with multiple inputs, such as joins. As mentioned in Section 4.3, implementing a join requires defining separate Mapper phases and attaching a tag to identify the inputs. These are then processed by the standard Shuffle pattern and delivered to Reducer tasks which must separate the tags and finalize the join execution. This situation is clearly a kind of "hack", in which we use the available Shuffle pattern to partially execute the operator, hand-coding the rest of the execution inside the Reduce function. This results in a sequence of operations which is in fact obscuring the semantics of a single join. The solution is not only inelegant, but also introduces a performance bottleneck, because it does not exploit the data parallelism between joined tuples, as all tuples with the same join key end up in a single Reducer task. For these reasons, a parallel computational model would be more suitable for query processing if it would provide multi-input patterns tailored to operations such as joins, unions, cartesian

[1] *tee* is a UNIX command which pipes the output of a command to another process and writes a copy of the streamed data in a separate file

products, etc.

Towards a General Approach

Several proposals for query-processing languages that can be translated to MapReduce have been published and will be discussed in Section 6.2 below. Some of them overcome the drawbacks of XQuery which we introduced earlier, but this usually comes at a cost in flexibility or expressiveness. Unfortunately, there is no language which provides all the advantages of XQuery and does not suffer from its disadvantages. In this scenario, it seems that an extended version of XQuery, with an expanded data model, finer granularity of data-processing patterns, and higher expressive power—equivalent to a functional or imperative programming language—would be a strong candidate for the "holy grail" of data-processing languages. Whether such a language would provide substantial advantages to existing ones, while still being reasonably simple, is a matter for extensive future research, but XQuery has already impacted many recent query languages, which proves that its benefits are valuable for expressive and flexible query processing.

In terms of frameworks for distributed query processing, there have been strong and elegant proposals, such as Nephele/PACT [7] and DryadLINQ [37], which overcome all the drawbacks of MapReduce. Similar to our FLWOR pipelines and splits, such approaches model queries as trees (or directed acyclic graphs) of higher-order processing patterns. The key advantage is that this pattern-based programming model is decoupled from the execution engine, which is based on parallel dataflow graphs. The vertices in these graphs are *processes*, which correspond to the execution of a certain function by a single node in the cluster. Hence, the more vertices occur in the graph, the more parallelism is being exploited. The edges represent data streams between such processors. Figure 6.1 illustrates the characteristic dataflow graph of a single MapReduce job, where we have Mapper processes $m_1 \dots m_M$ and Reducer processes $r_1 \dots r_R$.

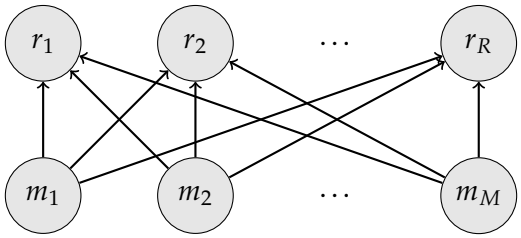


Figure 6.1: Parallel dataflow graph of a MapReduce job

Parallel dataflow graphs are a flexible execution model for distributed computations, in which a vertex is scheduled to execution in a machine once all of its incoming data streams are available. The model of dataflow graphs is very general, allowing the representation of arbitrary parallel computations beyond the scope of query processing. Data-parallel computations are manifested by nodes with multiple outgoing edges, while task parallelism can be achieved by identifying independent sets, or co-cliques. Note that the number of vertices is independent from the number of machines in the cluster, but for every task there is a certain

graph which produces the highest level of parallelism in a certain cluster configuration. The work in [2] specifies a cost model for the analysis of algorithms represented as such graphs.

The approach of a distributed query processor based on dataflow graphs is to compile the query plan into such graphs. Hence, there is a clear distinction between the programming model, i.e., the query plan, and the execution model, i.e., the dataflow graph, and the task of the query compiler is to translate from the former representation to the latter. The computational model of MapReduce, on the other hand, produces a fixed pattern of dataflow graphs, illustrated in Figure 6.1. Such inflexibility restricts the ability of the compiler to generate optimal graphs, which is why experiments in the literature have shown that the dataflow-graph model outperforms MapReduce in query-processing tasks [7]. For these reasons, it seems that the adoption of this general execution model, which generalizes MapReduce, is of great advantage for distributed query processing.

6.2 Related Work

The following discussion provides a brief overview and analysis of related approaches for query processing on MapReduce. Each of them provides a different language, but the translation mechanism to MapReduce jobs is essentially the same, similar to our approach of Pipeline Splitting and Job Generation. Furthermore, all the approaches provide a convenient abstract layer on top of serialized data formats, similar to our XDM mapping mechanism. Therefore, we focus solely on the analysis of the query languages and their data models.

Hive

The main goal of Hive [31] is to provide a data-warehousing solution on top of the Hadoop framework, focused on traditional relational OLAP scenarios. Its data model is based on tables whose columns contain standard atomic types, such as integer and strings, as well as the composite types *map*, for associative lists; *list* for lists; and *struct* for a sequence of attributes (i.e., key-value pairs).

The query language used by Hive—HiveQL—closely resembles a subset of SQL, with a few changes and enhancements to support the typical workloads of a MapReduce scenario. Queries usually begin with an `INSERT OVERWRITE` statement, which stores the output of a query in a specified table, dropping all previously existing records. The statement requires the columns returned by the query to conform to the table schema. Therefore, Hive does not support the flexibility of untyped data in a "data first, schema later" scenario.

Point-update statements such as `INSERT` and `UPDATE` are not supported, given the MapReduce environment. However, Hive supports the addressing of table partitions, which can be overwritten individually without affecting the other partitions. Partition keys are defined inside `CREATE TABLE` statements, and they are logically presented as columns of the table, despite not being physically stored as such. In a typical data warehousing scenario, partitions are created based on a time period, such as day or month, and data is produced incrementally. Then, at the end of a time period, analytical data is collected using `SELECT` statements and stored in a new partition corresponding to that period. Figure 6.2 shows two examples

of HiveQL statements, the first one creating a table containing an aggregated log of monthly website visits and the second one filling this table with records from the raw log in table *visits*. Note how *month* is accessed as a normal column, but it is in fact a partitioning key.

```
CREATE TABLE monthly_visits (host string, visits integer)
PARTITIONED BY (year string, month string);

INSERT OVERWRITE TABLE monthly_visits
PARTITION (year='2012', month='06')
  SELECT host, count(*)
  FROM visits
  WHERE year='2012' AND month='06'
  GROUP BY host;
```

Figure 6.2: Example HiveQL queries

Furthermore, HiveQL provides the `MAP` and `REDUCE` statements, which allow specifying MapReduce jobs explicitly. This is particularly useful to use external programs to execute the Map or Reduce functions. Both statements result in tables which can be accessed in a `FROM` clause or delivered to an `INSERT OVERWRITE` to be stored.

The clear advantage of Hive is that it is particularly convenient for SQL users and relational OLAP scenarios. However, as already mentioned, the requirement for schemas and the relational data model make it a quite inflexible approach, which does not exploit the generality of MapReduce in handling different data models. Furthermore, in adopting the bulky style of SQL, it does not provide a semi-declarative approach to apply operations step-by-step, in the order which they are naturally conveyed by the developer.

Pig

The Pig system [30] uses the PigLatin language, which provides a procedural style for representing computations, more familiar to the usual programmer. Each statement corresponds to a single operator, and its resulting data stream can be assigned to a variable. These variables are then made available to following statements, which consume their data streams and produce a new data stream. Hence, the dataflow is constructed explicitly by the programmer through variable assignments and references, which is probably why the authors refer to is as a *scripting language*.

The data model of PigLatin is similar to HiveQL: essentially relational with support for collection types. The difference is that the tables themselves are represented as instances of the *bag* data type, which is essentially a list of *tuple* values which supports duplicates. Hence, every variable holds a reference to a bag, and data access operators return bags as well. A tuple is simply a list of values without attribute names, equivalent to the mathematical definition. Furthermore, PigLatin supports a *map* data type, which represent associative lists where the keys must be atomic types. As in Hive, the collection types can be arbitrarily nested.

Figure 6.3 shows a sample query which joins and aggregates the *lineitem* and *orders* tables of the TPC-H benchmark.

```
lineitem = LOAD 'lineitem.txt'
  AS (orderkey:int, partkey:int, ...);
filtered = FILTER lineitem BY shipdate > '1998-09-02'
orders = LOAD 'orders.txt'
  AS (orderkey:int, custkey:int, ...);
joined = JOIN filtered BY orderkey, orders BY orderkey;
grouped = GROUP joined BY returnflag;
result = FOREACH grouped GENERATE group, SUM(joined.totalprice);
STORE result INTO 'result.txt';
```

Figure 6.3: Example PigLatin query

The `LOAD` operator fetches tabular data from an input source such as a file, creating a bag with the contents. The `AS` clause specifies the schema, which allows columns to be addressed by name. Note that bags simply contain tuples, and hence the column names are kept in an expression context rather than embedded in the data. If no schema is specified, columns can be referenced through their index, using the syntax `$0`, `$1`, etc. The bag resulting from scanning the *lineitem* table is stored in the variable `lineitem`, but the data is not fetched eagerly. Input files are only retrieved and processed when a write is requested with `STORE`, or results displayed in the user interface with `DUMP`. Therefore, the execution model corresponds to that of query plans, where the directed acyclic graph of operators is built from following variable references. This means that the procedural style is only a matter of syntax, since internally the traditional model of higher-order data-processing patterns is applied.

The `FILTER` operator performs filtering according to the expression in the `BY` clause. Note that the column `shipdate` can be accessed by its name, according to the schema definition in `LOAD`. The `JOIN` operator supports equi-joins on values returned by the evaluation of an arbitrary expression on each input bag. `GROUP` is similar to XQuery's `group by`, forming groups of tuples which have the same key. Non-grouped columns are combined in a bag. The `FOREACH` operator is similar to the `return` clause of a FLWOR expression. It evaluates a list of expressions for each tuple in a bag, therefore performing a transformation pattern. At the end, the `STORE` operator writes out the resulting bag into a file.

The translation of queries into MapReduce is based on the general `COGROUP` operator. A `COGROUP` is a generalized version of `GROUP`, which operates on multiple inputs. Given n input bags, it produces a bag with tuples of length $n + 1$, where the first item is the value of the grouping key, and the remaining n items are bags containing the grouped tuples of the respective inputs. Therefore, it is equivalent to a Shuffle with multiple Mappers, where each input is identified with a tag value. Here, the input can be identified by the index on the resulting tuple, and hence no explicit tags are necessary.

The translation mechanism converts all blocking operators into an equivalent form using `COGROUP`. In order to implement `JOIN`, for instance, a `COGROUP` is performed on the inputs based on their join keys, and the resulting tuples are processed with the `FLATTEN` operator, which unnests bags of tuples that contain other tuples themselves, as in the output of `COGROUP`. Note how this translation of a `JOIN` into `COGROUP` and `FLATTEN` has the exact same

semantics as our join mechanism, based on a Shuffle and a `PostJoinIterable` iterator.

Once all blocking operators have been converted into `COGROUP`s, the Pig system splits the query plan into Mapper and Reducer phases, translating each `COGROUP` into a Shuffle.

The main advantage of Pig towards Hive is the less declarative style of the language, which allows for greater flexibility in composing extensive data-processing pipelines. The decomposition of operator composition into explicitly defined variable references also provides a more natural mechanism to express the *tee* functionality, allowing different operators to consume the output of a single variable. Whether the procedural style is an advantage is only a matter of taste, because, internally, query processing is still done in the functional style of pattern composition. Authors of the system claim that developers prefer this style, as it is more familiar for programmers of imperative languages like Java, Python, or C.

The authors of PigLatin also claim in [30] that the language supports a fully nested data model, through the use of the composable collection types bag, tuple, and map. It does in fact allow modelling a hierarchy of values, but we object their claim because the computational model itself is not natively hierarchical. Bags which are nested inside the tuples of a containing bag are not treated transparently in the same manner as the top-level bag, which means sub-queries are only supported to a limited extent. The only operations allowed in a nested bag are `FILTER`, `ORDER`, and `FOREACH`. Because of these restrictions, which are not present in XQuery, we do not consider PigLatin's data model as fully nested. Furthermore, the use of collection types does not naturally model hierarchical relationships as in semistructured models like XML, where we have the concept of a child, parent, sibling, etc. This also means that the access to nested data is expressed only with data-access functions, instead of the much more expressive path expressions of XML.

JAQL

The last related approach which we discuss is JAQL [9]. It was introduced some years after the other approaches have been published and released, and thus it was designed to overcome many of their drawbacks. The JAQL language was heavily influenced by XQuery, but it is based on the JSON data model [15], which is basically a simplified version of XML. JSON has become very popular as a format for data exchange in the Web, and currently many public Web APIs are designed to deliver JSON instead of XML. The fact that the JAQL language is based on JSON represents a major advantage towards the alternative approaches, because not only it provides a truly nested data model, but, like XML, it is also an industry standard.

The actual data model used by JAQL is called JDM, but the only difference to JSON is that is provides more atomic types. Therefore, we simply consider JSON in our analysis.

JSON values can be atomic values, arrays, or records. Arrays are simple ordered lists, and records are a sequence of key-value pairs, where keys are identifiers (e.g., names used at the language syntax level) and values can be arbitrary JSON values. An advantage for query processing in JSON is that, unlike XML, it does not have node identity or references, which simplifies the implementation. Nevertheless, it may be a restriction for modelling data which requires identity or referential constraints.

The type system of JAQL is very simple and flexible. Unlike XQuery and XML Schema, it

decouples the notion of type and schema. The only supported types are the aforementioned JSON values, which do not support type parameters. An array of records, for instance, has the same type as an array of integers. However, the type of values can be restricted by specifying schemas, which are explicit definitions of the types contained in arrays or records. For example, the schema `[{uri:string, *} ...]` represents an array of records whose first attribute is `uri`. Note that the `*` symbol allows for partial schema definition, and hence for a "data first, schema later" approach, which is a great advantage when compared to Hive or Pig. Later on, the schema my be refined to `[{uri:string, accessed:date,*} ...]`, but its *type* is still compatible with the previous definition, since they both represent arrays.

It is questionable whether the schema approach of JAQL is preferable to XML Schema, since it relies on a type system which provides weak type-safety guarantees. In XML Schema, we can define a record schema similar to the above (including partial schema definitions like the `xs:any` type), but the difference is that we actually define a named type for all records with the given structure. This means that we can define functions whose arguments are of type `Person`, or `LogRecord`, which results in better abstraction and more readable code. The authors of JAQL claim that its schema strategy improves reusability because schema refinement does not require rewriting all the queries which were written based on the previous schema, as the types are still compatible. However, in order to exploit the refinement and produce safer and more efficient queries, these still have to be rewritten. In fact, in the case of function definitions, the type of the arguments must be modified to reflect the refinements, whereas in XML Schema no changes are necessary, since types are referred to by name. For these reasons, we believe that JAQL has severe drawbacks for scenarios where schemas are well-defined and type safety is desired. The type system of XQuery, in contrast, provides comprehensive support to the whole range of completely, partially, and totally defined schemas.

Similar to PigLatin, a JAQL script is a sequence of statements, separated by the `;` symbol. However, instead of requiring variables for composing data-processing patterns, JAQL allows use of standard function composition. Operators are simply higher-order functions on arrays of records, which can be composed to produce a query plan with the convenient composition operator `->`, which avoids the cumbersome function composition syntax. The expression `e->f(a)->g(b)` is equivalent to `g(f(e,a),b)`. Nevertheless, JAQL still provides a variable assignment statement, which is useful to break complex computations into parts. Furthermore, because data-processing patterns are plain functions, pipeline sections can be encapsulated into user-defined functions and be used transparently in query expressions. Figure 6.4 shows an example query, which computes the same results as the PigLatin example in Figure 6.3.

In the example, data is fetched as an array of records with the `read` function, which is composed with `hdfs` to read from a file in the distributed filesystem. Note that the variable-binding style of PigLatin can also be used, as we did in the example with the fetched tables that serve as join inputs. Since the join is a binary operator, it does not support the `->` syntax, so it is more convenient to assign the input arrays to variables. The expression which is evaluated in each operator has access to the special variable `$`, which holds a reference to a each record in the array. Thus, it introduces an iteration semantics, similar to a `for` clause in XQuery. The behavior of the `group by` operator is similar to SQL, in which aggregation is performed

```
lineitem = read(hdfs('lineitem.txt'));
orders = read(hdfs('orders.txt'));
result = join orderkey in lineitem, orderkey in orders
  -> filter $.shipdate > '1998-09-02'
  -> group by rt = $.returnflag
     into { rt , sum_price: sum($.totalprice) }
  -> write(hdfs('result.txt'));
```

Figure 6.4: Example JAQL query

together with grouping, in a single step. The aggregation can be specified in the `into` clause, which in our case constructs a record with the grouping key and a `sum_price` field containing the sum of total prices. As the last step, the aggregated array is passed to the `write` function to store the results in a file.

The great advantage of JAQL towards XQuery is the representation of operators as higher-order functions, which allows for greater flexibility, abstraction, and composability. This not only allows pipeline sections to be encapsulated in functions or assigned to variables, but also gives opportunity for the user to implement their own operators. Furthermore, the use of higher-order functions allow for a feature which the authors of JAQL refer to as *physical transparency*, which basically means source-to-source compilation. The compilation of a JAQL script results in another JAQL script where only a subset of core, low-level functions is used. This allows users to customize and tweak the compiled query plans, in an approach referred to as *bottom-up extensibility*.

The translation to MapReduce jobs is also based on the physical transparency feature, using the built-in higher-order functions `mapReduce` and `mrAggregate`. The former receives a JSON record as parameter, which describes the inputs and Map functions, as well as the single Reduce function and its output path. Note that given the higher-order nature of JAQL, the elements in a record may also be functions, which is how the Map and Reduce functions are specified. The latter function is a variant for Reduce functions that apply a distributive aggregate function, which end up generating a *combiner* in the MapReduce job.

6.3 Future Directions

The goal of this thesis was to introduce a framework for distributed query processing based on the XQuery language and the MapReduce computational model. Despite the wide success of MapReduce in industry and research, query-processing solutions based on it are still in their primary stages of development, and there are still many open challenges to be tackled.

In the field of higher-level data-processing languages for MapReduce, what we have is a wide range of proposals, which despite being similar goals, have varying characteristics. Since SQL is usually discarded in the large-scale Web-application scenario for its lack of flexibility, different companies and research projects have developed their own languages, and there is currently no standardization effort. Our proposal is to use XQuery, a language which has been standardized for nearly a decade, is already established in several real-world applications,

and fulfills most of the requirements for query processing in MapReduce. In order to fulfill all the requirements, we envision extensions of XQuery to incorporate the virtues of existing approaches.

Concerning the runtime layer of a MapReduce-based query processor, we have identified several techniques which can be applied to improve efficiency and achieve performance closer to parallel databases. Existing frameworks for MapReduce-based query processing undergo constant improvement, adapting techniques and algorithms from decades of database research. The incorporation of NoSQL systems or single-node DBMSs as storage layer of such frameworks is also a very promising research direction, as it provides a significant performance boost for MapReduce queries.

When analyzing the problem of distributed query processing from scratch, we conclude that an execution based on parallel dataflow graphs, which generalize MapReduce computations, may be better suited. Such ideas have already been exploited in research, but there are still many open questions.

All these opportunities for future research are part of a much larger context in the world of data management. Since the arrival of new technologies like cloud computing, Big Data, and MapReduce, virtually all areas of database systems research have been revisited and new concepts were introduced to meet the requirements of these new environments. This thesis tackled a small niche of the open challenges, but given the many paradigm shifts that are currently occurring, the road ahead looks promising and exciting.

Bibliography

[1] Azza Abouzeid, Kamil Bajda-Pawlikowski, Daniel J. Abadi, Alexander Rasin, and Avi Silberschatz. HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads. *PVLDB*, 2(1):922–933, 2009.

[2] Foto N. Afrati and Jeffrey D. Ullman. A new computation model for cluster computing. Technical report, National Technical Univ. of Athens/Stanford University, December 2009.

[3] Foto N. Afrati and Jeffrey D. Ullman. Optimizing Multiway Joins in a Map-Reduce Environment. *IEEE Trans. Knowl. Data Eng.*, 23(9):1282–1298, 2011.

[4] Sebastian Bächle and Caetano Sauer. Unleashing XQuery for Data-independent Programming. *Submitted*.

[5] Roger Bamford, Vinayak R. Borkar, Matthias Brantner, Peter M. Fischer, Daniela Florescu, David A. Graf, Donald Kossmann, Tim Kraska, Dan Muresan, Sorin Nasoi, and Markos Zacharioudaki. XQuery Reloaded. *PVLDB*, 2(2):1342–1353, 2009.

[6] Luiz André Barroso, Jeffrey Dean, and Urs Hölzle. Web Search for a Planet: The Google Cluster Architecture. *IEEE Micro*, 23(2):22–28, 2003.

[7] Dominic Battré, Stephan Ewen, Fabian Hueske, Odej Kao, Volker Markl, and Daniel Warneke. Nephele/PACTs: a programming model and execution framework for web-scale analytical processing. In *SoCC*, pages 119–130, 2010.

[8] Kevin S. Beyer, Donald D. Chamberlin, Latha S. Colby, Fatma Özcan, Hamid Pirahesh, and Yu Xu. Extending XQuery for Analytics. In *SIGMOD Conference*, pages 503–514, 2005.

[9] Kevin S. Beyer, Vuk Ercegovac, Rainer Gemulla, Andrey Balmin, Mohamed Y. Eltabakh, Carl-Christian Kanne, Fatma Özcan, and Eugene J. Shekita. Jaql: A Scripting Language for Large Scale Semistructured Data Analysis. *PVLDB*, 4(12):1272–1283, 2011.

[10] Richard Bird. *Introduction to Functional Programming using Haskell*. Prentice Hall PTR, 2 edition, May 1998.

[11] Vinayak R. Borkar, Michael J. Carey, Dmitry Lychagin, and Till Westmann. The BEA AquaLogic data services platform (Demo). In *SIGMOD Conference*, pages 742–744, 2006.

[12] Irina Botan, Peter M. Fischer, Daniela Florescu, Donald Kossmann, Tim Kraska, and Rokas Tamosevicius. Extending XQuery with Window Functions. In *VLDB*, pages 75–86, 2007.

[13] Sergey Brin and Lawrence Page. The Anatomy of a Large-Scale Hypertextual Web Search Engine. *Computer Networks*, 30(1-7):107–117, 1998.

[14] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM Trans. Comput. Syst.*, 26(2), 2008.

[15] D. Crockford. The application/json Media Type for JavaScript Object Notation (JSON). RFC 4627 (Informational), July 2006.

[16] Jeffrey Dean and Sanjay Ghemawat. MapReduce: a flexible data processing tool. *Commun. ACM*, 53(1):72–77, 2010.

[17] David J. DeWitt and Jim Gray. Parallel Database Systems: The Future of High Performance Database Systems. *Commun. ACM*, 35(6):85–98, 1992.

[18] Ghislain Fourny, Markus Pilman, Daniela Florescu, Donald Kossmann, Tim Kraska, and Darin McBeath. XQuery in the browser. In *WWW*, pages 1011–1020, 2009.

[19] Goetz Graefe. Query Evaluation Techniques for Large Databases. *ACM Comput. Surv.*, 25(2):73–170, 1993.

[20] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Mining and Knowledge Discovery*, 1:29–53, 1997. 10.1023/A:1009726021843.

[21] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[22] Theo Härder. DBMS Architecture - New Challenges Ahead. *Datenbank-Spektrum*, 14:38–48, 2005.

[23] Tony Hey, Stewart Tansley, and Kristin M. Tolle, editors. *The Fourth Paradigm: Data-Intensive Scientific Discovery*. Microsoft Research, 2009.

[24] Shadi Ibrahim, Hai Jin, Lu Lu, Song Wu, Bingsheng He, and Li Qi. LEEN: Locality/Fairness-Aware Key Partitioning for MapReduce in the Cloud. In *CloudCom*, pages 17–24, 2010.

[25] Adam Jacobs. The pathologies of big data. *Commun. ACM*, 52(8):36–44, August 2009.

[26] Martin Kaufmann and Donald Kossmann. Developing an Enterprise Web Application in XQuery. In *ICWE*, pages 465–468, 2009.

[27] Michael Kay. Ten Reasons Why Saxon XQuery is Fast. *IEEE Data Eng. Bull.*, 31(4):65–74, 2008.

[28] Ralf Lämmel. Google's MapReduce programming model - Revisited. *Sci. Comput. Program.*, 70(1):1–30, 2008.

[29] Mike Loukides. What is data science? `http://radar.oreilly.com/2010/06/what-is-data-science.html`.

[30] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD Conference*, pages 1099–1110, 2008.

[31] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang, Suresh Anthony, Hao Liu, and Raghotham Murthy. Hive - a petabyte scale data warehouse using Hadoop. In *ICDE*, pages 996–1005, 2010.

[32] Transaction Processing Council. The TPC-H Benchmark. http://www.tpc.org/tpch/, October 2011.

[33] W3C. XQuery 3.0: An XML Query Language. `http://www.w3.org/TR/xquery-30/`, 2011.

[34] W3C. XQuery and XPath Data Model 3.0. `http://www.w3.org/TR/xpath-datamodel-30/`, 2011.

[35] W3C. XQuery and XPath Data Model 3.0. `http://www.w3.org/TR/xmlschema-11-1/`, 2011.

[36] Tom White. *Hadoop - The Definitive Guide: Storage and Analysis at Internet Scale (2. ed.)*. O'Reilly, 2011.

[37] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. DryadLINQ: a system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI*, pages 1–14, Berkeley, CA, USA, 2008. USENIX Association.