Parallelizing XML Data-Streaming Workflows via Map-Reduce

Daniel Zinn[*,a], Shawn Bowers[b,c], Sven Köhler[b], Bertram Ludäscher[a,b]

[a] *Dept. of Computer Science, UC Davis*
[b] *UC Davis Genome Center, UC Davis*
[c] *Dept. of Computer Science, Gonzaga University*

Abstract

In prior work it has been shown that the design of scientific workflows can benefit from a collection-oriented modeling paradigm which views scientific workflows as pipelines of XML stream processors. In this paper, we present approaches for exploiting data parallelism in XML processing pipelines through novel compilation strategies to the Map-Reduce framework. Pipelines in our approach consist of sequences of processing steps that receive XML-structured data and produce, often through calls to "black-box" (scientific) functions, modified (i.e., updated) XML structures. Our main contributions are (i) the development of a set of strategies for compiling scientific workflows, modeled as XML process pipelines, into parallel MapReduce networks, and (ii) a discussion of their advantages and trade-offs, based on a thorough experimental evaluation of the various translation strategies. Our evaluation uses the Hadoop MapReduce system as an implementation platform. Our results show that execution times of XML workflow pipelines can be significantly reduced using our compilation strategies. These efficiency gains, together with the benefits of MapReduce (e.g., fault tolerance) make our approach ideal for executing large-scale, compute-intensive XML-based scientific workflows.

1. Introduction

Scientific data analysis often requires the integration of multiple domain-specific tools and specialized applications for data processing. The use of these tools within an analysis is typically automated through scripts and, more recently, through *scientific workflow* systems [1, 2, 3], resulting in data-driven and often compute-intensive processing pipelines that can be executed to generate new data products or to reproduce and validate prior results.

XML and XML-like tree structures are often used to organize and maintain collections of data, and so it is natural to devise data-driven processing pipelines (e.g., scientific workflows) that work over nested data (XML). A number of such approaches have recently been developed, e.g., within the web [4, 5, 6], scientific workflow [7, 8, 9], and database communities [10, 11, 12]. These approaches provide mechanisms to create complex pipelines from individual computation steps that access and transform XML data, where each step converts input data into new data products to be consumed by later steps. Additionally, pipelines often include steps that are applied only to portions of their input structures (leaving the remaining portions unchanged), in which case steps can be seen as performing specialized XML update operations [7, 13, 14]. The components implementing these computation steps often employ techniques from XML processing (e.g., XPath, XQuery, XSLT), and call built-in functions or external applications to perform "scientifically meaningful" computations (e.g., DNA sequence alignment, image processing, or similarly specialized algorithms).

Many of the above approaches employ *pipeline parallelism* to more efficiently execute pipelines by streaming XML data through components, thus allowing *different* steps within a pipeline to work concurrently over an XML stream. However, these approaches largely ignore *data parallelism*, which can significantly reduce pipeline execution time by allowing the *same* step to be executed in parallel over distinct subcollections of data.

[*] Corresponding author
Email addresses: dzinn@ucdavis.edu (Daniel Zinn), sbowers@ucdavis.edu (Shawn Bowers), svkoehler@ucdavis.edu (Sven Köhler), ludaesch@ucdavis.edu (Bertram Ludäscher)

In this paper, we present approaches that utilize MapReduce [15] to facilitate data-parallel computation over XML. For example, consider the simple XML processing pipeline shown in Fig. 1. This pipeline consists of five steps, each of which (i) receives XML structures from previous steps, and (ii) works over specific XML fragments (subtrees) within these structures. These fragments are determined through XPath expressions that specify the "*scope*" of a step. Steps are invoked over each scope match (i.e., matching XPath fragment), and steps can perform arbitrary modifications to matched fragments using general XML processing techniques (e.g., XQuery, XSLT). The modifications made by steps often involve calling built-in (scientific) functions whose outputs are added within the matched fragment, or used to replace existing parts of the fragment. The result is a modified (i.e., updated) XML structure that is passed to subsequent steps. As an example, the first step of Fig. 1 has the scope "//B", allowing it to perform arbitrary changes on "B"-rooted subtrees, i.e., new data items or subtrees can be inserted anywhere below the "B" node. However, for the middle step with scope "//D", changes may only be performed at the leaves of the given structure shown in the bottom-left of Fig. 1.

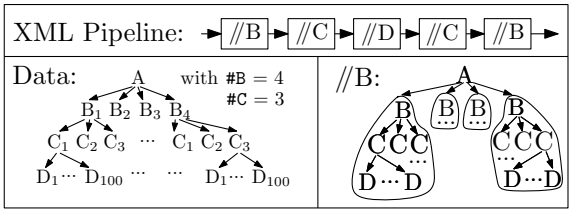


Figure 1: XML Pipeline (top), where each step is labeled with its *scope* of work (*e.g.*, //B, //C), shown with sample input data (bottom left) and data partitioning for Steps 1 and 5 (bottom right).

To exploit data parallelism, we map scope matches (fragments) to "work-pieces" that are then processed in parallel by MapReduce. The bottom-right of Fig. 1 shows how the data is partitioned for the scope "//B" as used in Steps 1 and 5. A naive adoption of this approach, however, can lead to bottlenecks in the splitting and regrouping phase of the parallel MapReduce execution. For example, from Step 1 to 2 the subtrees shown at the bottom right of Fig. 1 must be partitioned further. Grouping all work-pieces together again to then re-split for the second task is clearly inefficient. Furthermore, from Step 3 to 4, the "D"-rooted trees must be re-grouped to form trees rooted at "C". Performing this grouping in a single global task also adds an unnecessary bottleneck because all required regroupings could be done in parallel for each "C"-rooted subtree.

Contributions. We describe and evaluate three novel strategies—*Naive*, *XMLFS*, and *Parallel*—for executing XML processing pipelines via the MapReduce framework. The Naive strategy deploys a simple step-wise splitting as outlined above. The XMLFS strategy maps XML data into a distributed file system to eliminate the grouping bottleneck of the Naive strategy. The Parallel strategy further utilizes existing splits to re-split the data in parallel, thereby fully exploiting the grouping and sorting facilities of MapReduce. In general, each of these strategies offers distinct approaches for applying MapReduce to data-parallel processing of XML.

We also present a thorough experimental evaluation of our strategies. Our experiments show a twenty-fold speedup (with 30 hosts) in comparison to a serial execution, even when the basic Naive strategy is used. We also show that our Parallel approach significantly outperforms Naive and XMLFS for large data and when the cost for splitting and grouping becomes substantial. We consider a wide range of factors in our experiments—including the number of mapper tasks, the size of data and the XML nesting structure, and different computational load patterns—and we show how these factors influence overall processing time using our strategies.

Outline. The rest of the paper is organized as follows. In Section 2 we describe the MapReduce paradigm and an example that demonstrates the features utilized in our strategies. In Section 3 we describe a framework for XML processing pipelines, introduce important notions for their parallel execution, and give several pipeline examples. Section 4 presents our three parallelization strategies as well as their advantages and trade-offs. In Section 5 we present our experimental evaluation. We discuss related work and conclude in Section 6.

2. Preliminaries: MapReduce

MapReduce [15] is a software framework for writing parallel programs. Unlike with PVM or MPI, where the programmer is given the choice of how different processes communicate with each other to achieve a common task, MapReduce provides a fixed *programming scheme*. A programmer employing the MapReduce framework implements *map* and *reduce* functions, and the MapReduce library carries out the execution of these functions over corresponding input data. While restricting the freedom of how processes communicate with each other, the MapReduce framework is able to automate many of the details that must be considered when writing parallel programs, e.g., check-pointing,

execution monitoring, distributed deployment, and restarting individual tasks. Furthermore, MapReduce implementations usually supply their own distributed file systems that provide a scalable mechanism for storing large amounts of data.

Programming model. Writing an application using MapReduce mainly requires designing a *map* function and a *reduce* function together with the data types they operate on. Map and reduce implement the following signatures

$$\begin{aligned} \text{map} &:: (K_1, V_1) \rightarrow [(K_2, V_2)] \\ \text{reduce} &:: (K_2, [V_2]) \rightarrow [(K_3, V_3)] \end{aligned}$$

where all K_i and V_i are user-defined data types. The map function transforms a *key-value pair* (short kv-pair) into a list of kv-pairs (possibly) with different types. The overall input of a MapReduction is a (typically large) list of kv-pairs of type (K_1, V_1). Each of these pairs is supplied as a parameter to a map call. Here, the user-defined map function can generate a (possibly empty) list of new (K_2, V_2) pairs. All (K_2, V_2) pairs output by the mapper will be grouped according to their keys. Then, for each distinct key the user-defined reduce function is called over the values associated to the key. In each invocation of reduce the user can output a (possibly empty) list of kv-pairs of the user-defined type (K_3, V_3).

The MapReduce framework divides the overall input data into kv-pairs, and splits this potentially large list into smaller lists (so-called *input splits*). The details of generating kv-pairs (and input splits) can also be specified by the user via a custom *split* function. After kv-pairs are created and partitioned into input splits, the framework will use one separate map process for each input split. Map processes are typically spawned on different machines to leverage parallel resources. Similarly, multiple reduce processes can be configured to process in parallel different distinct keys output by the map processes.

Example. Assume we want to generate a histogram and an inverted index of words for a large number of text files (e.g., the works of Shakespeare), where the inverted index is represented as a table with columns *word*, *count*, and *locations*. For each distinct word in the input data there should be exactly one row in the table containing the word, how often it appears in the data, and a list of locations that specify where the words were found. To solve this problem using MapReduce, we design the type K_1 to contain a filename as well as a line number (to specify a location), and the type V_1 to hold the corresponding line of text of the file. When given a (*location*, *text*) pair, map emits (*word*, *location*) pairs for each word inside the current line of text. The MapReduce framework will then group all output data by words, and call the reduce function over each word and corresponding list of locations to count the number of word occurrences. Reduce then emits the accumulated data (*count*, List of *locations*) for each processed word, i.e., the data structure V_3 will contain the required word count and the list of locations.

The MapReduce framework can additionally sort values prior to passing them to the reduce function. The implementation of secondary sorting depends on the particular MapReduce framework. For example, in hadoop [16] it is possible to define custom comparators for keys K_2 to determine the initial grouping as well as the order of values given to reduce calls. In our example above, we could design the key K_2 to not only contain the word but also the location. We would define the "grouping" comparator to only compare the word part of the key, while the "sorting" comparator would ensure that all locations passed will be sorted by filename and line number. The reduce function will then receive all values of type *location* in sorted order, allowing sorted lists of locations to be easily created.

In general, MapReduce provides a robust and scalable framework for executing parallel programs that can be expressed as combinations of map and reduce functions. To use MapReduce for parallel execution of XML processing pipelines, it is necessary to design data structures for keys and values as well as to implement the map and reduce functions. More complex computations can also make use of custom group and sort comparators as well as input splits.

3. Framework

The general idea behind transforming XML processing pipelines to MapReduce is to use map processes to parallelize the execution of each pipeline task according to the task's scope expression. For each scope match the necessary input data is provided to the map tasks, and after all map calls have executed, the results are further processed to form

either the appropriate input structures for the next task in the pipeline or the overall output data. For example, consider again the pipeline from Fig. 1. The partitioning and re-grouping of XML data throughout the pipeline execution is shown in Fig. 2: data in the first row is split into pieces such that at most one complete "B" subtree is in every fragment, which is then processed in parallel with the other fragments. Then, further splits occur for scope "C" and "D" respectively. Data is later re-grouped to ensure that all elements corresponding to a scope match are available as a single fragment.

In the following we define our data model and assumptions concerning XML processing pipelines. We also characterize the operations that may be performed on single fragments within map calls (i.e., by pipeline tasks) to guarantee safe parallel execution.

3.1. XML Processing Pipelines

We assume XML procesing pipelines that adopt the standard XML data model corresponding to labeled ordered trees represented as sequences of tokens; namely, opening tags "`T[`", closing tags "$\texttt{]}_{\texttt{T}}$", and data nodes "`#d`". Data nodes typically represent data products whose specific format is understood by the software components implementing pipeline tasks, but not by the XML framework itself, which treats data as opaque `CData` nodes. For instance, data nodes may represent binary data objects (such as images) or simple text-based data (e.g., DNA sequence alignments).

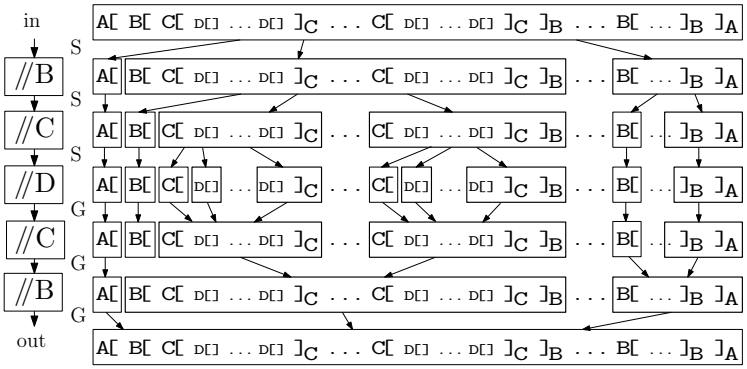


Figure 2: Splits and groups for Parallel execution. For each step in the pipeline the data is partitioned such that all data for one scope match is inside one fragment while each fragment holds at most one match.

Pipeline tasks typically call "scientific" functions that receive data nodes as input and produce data nodes as output. In addition, tasks are annotated with scopes that define where in the overall XML structure input data is taken from and output data is placed. Each scope specifies XML fragments within the input structure that represent the context of a task. Pipeline tasks may insert data (including XML tree data) anywhere within their corresponding context fragments or as siblings to the fragments, and remove data or complete subtrees anywhere within their fragments (including removing the fragments themselves). It is often the case that a given XML structure will contain multiple matching fragments for a task. In this case, the task is executed over each such match. We assume tasks do not maintain state between executions, thus allowing them to be safely executed in parallel over a given XML structure via the MapReduce framework.

More formally, a pipeline consists of a list of tasks $\mathcal{T}$ each of which *updates* an XML structure X to form a new structure X'. Further, $\mathcal{T} = (\sigma, \mathcal{A})$ where the scope σ is a (simple) qualifier-free XPath expression consisting of child (/) and descendent-or-self (//) axes, and $\mathcal{A}$ is a function over XML structures.

A subtree s_i in an input XML structure X is a scope match if $\sigma(X)$ selects the root node of s_i. For nested scope matches, only the highest-level match in X is considered—a common restriction (e.g., [17]) for avoiding nested executions. Formally, σ selects n non-overlapping subtrees s_i from X:

$$\sigma(X) = \{s_1, \ldots, s_n\}.$$

Then, the function $\mathcal{A}$ is called on each of these subtrees to produce a new XML fragment, i.e.:

$$\text{for each } s_i : \quad s'_i = \mathcal{A}(s_i).$$

The output document X' is then formed by replacing all s_i subtrees in X by the respective outputs s'_i:

$$X' = X[s_1 \to s'_1, s_2 \to s'_2, \ldots].$$

We require that $\mathcal{A}$ be a function in the mathematical sense, i.e., a result s'_i only depends on its corresponding input s_i. This implies that s'_i can be computed from s_i independently from data inside other fragments s_j or completely non-matched data in X.[1]

3.2. Operations on Token Lists

During pipeline execution we represent XML data as a sequence (i.e., list) of tokens of the form $\texttt{T[}_t$, $\texttt{]}_t$, and #d. By convention, we use capital letters to denote token lists and lowercase letters to denote trees and (ordered) forests[2]. Token lists are partitioned into fragments that are sent to map calls for concurrent processing. Below we characterize the changes the map calls may perform on fragments to avoid jeopardizing overall data integrity. Note that the proposed rules can be followed locally and thus eliminate the need for more involved locking mechanisms.

Definition (Balanced Token List). Given the following rules to modify token lists:

$$A\ \texttt{\#d}\ B \quad \Rightarrow \quad A\,B \qquad A, B \in \text{Token List} \tag{1}$$

$$A\ \texttt{X[}\ \texttt{]}_\texttt{X}\ B \quad \Rightarrow \quad A\,B \qquad A, B \in \text{Token List} \tag{2}$$

rule (1) deletes any data node whereas (2) deletes matching Open and Close nodes if they are next to each other within a sequence and have matching labels. As usual, we write $T \Rightarrow^* T'$ if there exists a sequence of token lists T_i such that $T = T_1 \Rightarrow T_2 \Rightarrow \cdots \Rightarrow T_n = T'$. A token list T is balanced if it can be reduced to the empty list, *i.e.*, $T \Rightarrow^* [\,]$.

Note that $\Rightarrow^*$ is normalizing, i.e., if $T \Rightarrow^* [\,]$ and $T \Rightarrow^* T'$ then $T' \Rightarrow^* [\,]$. This means that for a balanced list T, applying deletion rules (1) and (2) in any order will terminate in the empty list (by induction on list length). Also note that an XML forest naturally corresponds to a balanced token list and vice versa.

As described above, we want calls to map to compute new forests s'_i from existing trees s_i. In particular, s'_i can be computed by performing tree insertion and tree deletion operations in an arbitrary order on s_i. The following operations on token lists correspond to these allowed operations on trees.

Observation (Safe insertions). Inserting a balanced token list I at any position into a balanced token list T corresponds to inserting the forest i into the forest t (where forests i and t correspond to token lists I and T, respectively). In particular this operation results in a balanced token list T'. We call such an insertion a *safe insertion*.
Proof: The result T' is trivially a balanced token list, since the newly inserted balanced fragment I can be deleted from T' via the deletion rules given above, resulting in T, which is balanced. Furthermore, the balanced token list I corresponds to a forest i. Since any location between two tokens in the list T corresponds to a position in the forest t, a safe insertion will insert i at this position. □

Note that insertions which simply maintain the "balance" of a sequence, but are not safe, can change ancestors of already existing nodes. Consider the case of inserting the unbalanced fragment "$\texttt{]}_\texttt{A}\ \texttt{A[}$" into the middle of the balanced token list "$\texttt{A[\#d\#d]}_\texttt{A}$". This insertion will result in the balanced list "$\texttt{A[\#d]}_\texttt{A}\ \texttt{A[\#d]}_\texttt{A}$". However, the second #d token has changed parent nodes without explicitly being touched.

Observation (Safe deletions). Removing a consecutive and balanced token list D from a balanced token list T results in a balanced token list T'. This operation corresponds to deleting the forest d from t. We call such a deletion a *safe deletion*.
Proof: T' is trivially balanced since "⇒" is normalizing. □

Corollary 1 (Safe operations carry over to fragments of token lists). Viewing token-list fragments as parts of the total (balanced) token list, we can perform safe insertions and safe deletions to perform the desired operations inside the scope of a pipeline task.

Corollary (1) ensures that map calls can change their fragments by performing safe insertions and deletions without interfering with the data of other map calls. Moreover, since the complete scope is inside the fragment received by the map call, each map call is able to delete its scope match, or to perform any "localized" operations on it using all the data inside its scope.

[1] In essence, we perform a "map $\mathcal{A}$" on the list of scope matches with map being the standard map function of functional programming languages. We thus require that $\mathcal{A}$ be a function to parallelize $\mathcal{A}$ invocations.

[2] Although the term *hedge* seems more appropriate as it implies an order to the list of trees, we conform to most existing literature and use the term *forest* here to denote an ordered list of trees.

3.3. XML-Pipeline Examples

In addition to the simple pipeline introduced in Fig. 1, we also consider a common image processing pipeline shown in Fig. 3. This pipeline is similar to a number of (more complicated) scientific workflows that perform image processing, e.g., in functional Magnetic Resonance Imaging (fMRI) [18] and plasma-fusion data analysis [19]. The pipeline employs the *convert* and *montage* tools from the Image-Magick [20] suite to process multiple images organized according to nested XML collections. As shown in Fig. 3, a top-level "A" collection contains several "B" collections, each of which contains several "C" collections with an image d inside. The first step of the pipeline blurs the images (via the "`convert -blur`" command). Since this operation is performed on each image separately, we define the task's scope σ using the XPath expression //C and its corresponding function $\mathcal{A}$ such that it replaces the image within its scope with a modified image resulting from invoking the blur operation. The next step in the pipeline creates a series of four colorized images from each blurred image d' using the command "`convert -modulate 100,100,`i" with 4 different values for i. The last step of the pipeline combines all images under one "B" collection into a single large image using the montage tool. The scope σ of this last task is //B since all images inside a "B"-labeled tree are input to the montage operation. Here the framework groups previously split fragments to provide the complete "B" subtree to the montage task.

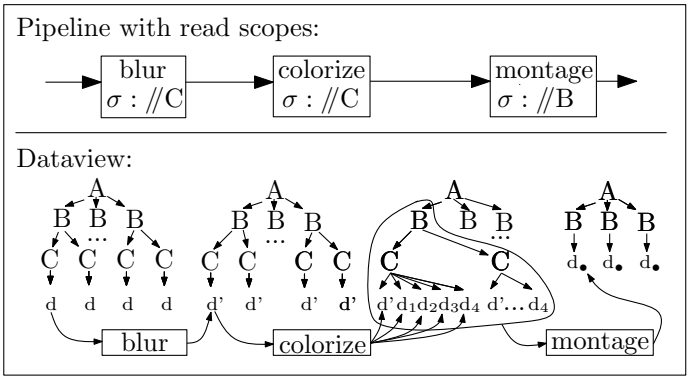


Figure 3: Image transformation pipeline. All images are *blurred*; then from each, four new images are created by *coloring*; and finally a big *montage* is created from all images below each "B".

4. Parallelization Strategies

We consider three strategies, *Naive*, *XMLFS* and *Parallel*, whose main differences are shown in Fig. 4. These strategies use variations of key-value data structures as well as split, map, and reduce operations, and build upon each other to address various shortcomings that arrise in large-scale and compute-intensive processing of nested data.

4.1. Naive Strategy

The Naive approach corresponds to a straightforward application of MapReduce over XML data. As shown in Fig. 4, we cut XML token sequences into pieces for the map calls, perform the task's operation $\mathcal{A}$ on its scope, and finally merge the result in the reduce step of the MapReduction to form the final output[3]. The Naive approach uses the following data structures for keys and values.

Key: Integer
Token := XOpen | XClose | Data
Value: TList := **List of** Token

For each task in an XML pipeline, we create a MapReduction with *split*, *map*, and *reduce* as shown in Fig. 5. From an XML structure, *SplitNaive* creates a kv-pair for each match of the task's scope: each pair comprises an *Integer* as key, and a *TList* as value.

To decide if a current token opens a new scope in line 4 of Fig. 5, we use a straightforward technique to convert the qualifier-free, simple XPath-expression σ into a deterministic finite-state automaton (DFA) reading strings of opening

[3]This parallelization is a form of a simple scatter and gather pattern.

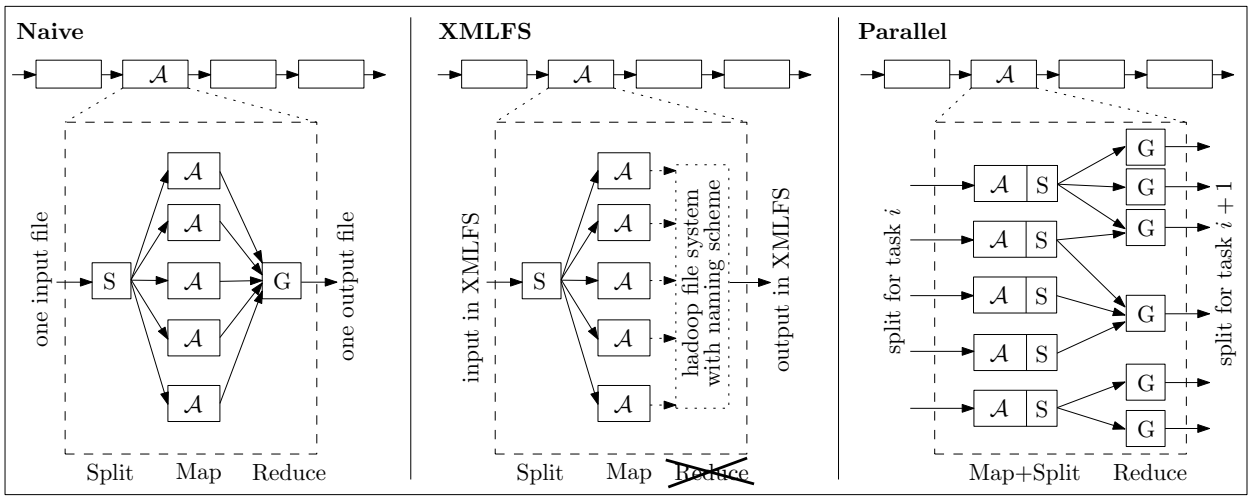


Figure 4: Processes and dataflow for the three parallelization strategies.

tokens. The DFA accepts when the read string conforms to σ. Using a stack of DFA states, we keep track of the current open tags. Here, we push the current state for an open token and reset the DFA to the state popped from the stack when a closing token is read. To prevent nested scope matches, we simply go into a non-acceppting state with self-loop after we encounter a match. Note that closing the match will "pop" the automaton back to the state before the match. We are able to use this simple and efficient approach for streaming token lists because of the simplicity of the XPath fragment in which scopes are expressed.[4] Considering more complex fragments of XPath together with available streaming algorithms for them, such as those in [21, 22], is beyond the scope of this paper.

The first pair constructed by *SplitNaive* contains all XML tokens before the first match, and each consecutive pair contains the matching data, possibly followed by non-matching data. Each pair is then processed in *MapNaive*. Then, *ReduceNaive* merges all data fragments back into the final XML structure. Since our grouping comparator always returns "equal", the one reduce task will receive all output from the mappers; also the fragments will be received in document order because the MapReduce framework will sort the values based on the key, which is increasing in document order. The output structure can now be used as input data for another MapReduce that executes the next step in the pipeline.

```
1  SplitNaive: TList input, XPath σ → [(Integer, TList)]
     int i := 0; TList splitOut := [ ]
3    FOREACH token IN input DO
       IF (token opens new scope match with σ) AND
5          (splitOut ≠ [ ]) THEN
         EMIT (i, splitOut) // one split for each scope match
7        i++; splitOut := [ ]
       splitOut.append(token)
9    EMIT (i, splitOut)

11 MapNaive: Integer s, TList val → [(Integer, TList)]
     val' := 𝒜(val) // execute pipeline task
13   EMIT (s, val')

15 ReduceNaive: Integer s, [TList] vals → [(Integer,TList)]
     TList output := [ ]
17   WHILE vals.notEmpty() DO
       output.append(vals.getValue()) // collapse to single value
19   EMIT (0, output)
```

Figure 5: Split, Map, Reduce for Naive strategy.

Shortcomings of the Naive Strategy

The major shortcoming of the Naive approach is that although data is processed in parallel by calls to map, both splitting and grouping token lists is performed by a single task. Split and reduce can thus easily become a bottleneck for the execution of the pipeline.

[4] In general, this fragment is sufficient for modeling many scientific applications and workflows.

4.2. XMLFS Strategy

The XMLFS strategy removes the bottleneck in the reduce phase of the Naive approach by mapping XML structures to a distributed file system (see Fig. 4). Many MapReduce implementations, including hadoop and Google's MapReduce, provide a distributed file system that allows efficient and fault-tolerant storage of data in the usual hierarchical manner of directories and files, and this distributed file system is employed in the XMLFS approach as follows.

Mapping XML structures to a file system. An XML document naturally corresponds to a file-system-based representation by mapping XML nodes to directories and data nodes to files. We encode the ordering of XML data by pre-pending the XML-labels with *identifiers* (IDs) to form directory and file names. The IDs will also naturally ensure that no two elements in the same directory will have the same name in the file system even though they have the same tag. Note that although we do not explicitly consider XML attributes here, we could, e.g., store them in a file with a designated name inside the directory of the associated XML element.

Using a file system based representation of XML data has many advantages: **(1)** XML structures can be browsed using standard file-system utilities. The hadoop software package, e.g., provides a web-based file-system browser for its hadoop file system (hdfs) [23]. **(2)** Large amounts of XML data can easily be stored in a fault-tolerant manner. Both hadoop-fs and the Google File System provide distributed, fault-tolerant storage. Specifically, they allow users to specify a replication factor to control how many copies of data are maintained. **(3)** The file system implementation provides a natural "access index" to the XML structure: In comparison to a naive token list representation, navigating into a subtree t can be performed using simple directory changes without having to read all tokens corresponding to subtrees before t. **(4)** Applications can access the "distributed" XML representation in parallel, assuming that changes to the tree and data are made at different locations. In particular, pipeline steps can write their output data s'_i in parallel.

XMLFS-Write. XMLFS adapts the Naive approach to remove its bottleneck in the reduce phase. Instead of merging the data into a large XML structure, we let each task write its modified data s'_i directly into the distributed file system. Since we do not need to explicitly group token lists together to form bigger fragments, we can perform this operation directly in the map calls. This approach removes the overhead of shuffling data between map and reduce calls as well as the overhead of invoking reduce steps. In particular, the XMLFS strategy does not use the grouping and sorting feature of the MapReduce framework since each task is implemented directly within the map function.

In XMLFS, the file system layer performs an *implicit* grouping as opposed to the *explicit* grouping in the Naive reduce function. When map calls write the processed XML token list T to the file system, the current path p from the XML root to the first element in T needs to be available since the data in T will be stored under the path p in the file system. We encode this information as a *leading path* into the key. IDs for maintaining order among siblings must also be available. Since map calls may not communicate with each other, the decisions about the IDs must be purely based on the received keys and values, and the modifications performed by a task's operation $\mathcal{A}$. Unfortunately, the received token lists are not completely independent: An opening token in one fragment might be closed only in one of the following fragments. Data that is inside such a fragment must be stored under the same directory on the file system by each involved map call independently. It is therefore essential for data integrity that all map calls use the same IDs for encoding the path from the document root to the current XML data. We now make these concepts more clear, stating requirements for IDs in general, as well as requirements for ID handling in split and map functions.

Requirements for Token-Identifiers (IDs). The following requirements need to be fulfilled by IDs: *Compact String Representation:* We require a (relatively small) string representation of the ID to be included in the token's filename, since we must use the ID for storing the XML data in the distributed file system. *Local Order:* IDs can be used to order and disambiguate siblings with possibly equal labels. Note that we do not require a total order: IDs only need to be unique and ordered for nodes with the same parent. *Fast comparisons:* Comparing two IDs should be fast. *Stable insertions and deletions:* Updates to XML structures should not effect already existing IDs. In particular, it should be possible to insert arbitrary data between two existing tokens. It should also be possible to delete existing tokens without changing IDs of tokens that have not been deleted.

Choice of IDs. Many different labeling schemes for XML data have been proposed; see [24] for a recent overview. For our purposes, any scheme that fulfills the requirements stated above could be used. These include ORDPATHs described in [25] or the DeweyID-based labels presented in [24]. However, note that many proposed ID solutions (including the two schemes just mentioned) provide global IDs, facilitate navigation (e.g., along parent-child axes),

and allow testing of certain relationships between nodes (e.g., whether a node is a descendent of another node). Since we only require local IDs, i.e., IDs that are unique only among siblings, and we do not use IDs for navigation or testing, we adopt a conceptionally easier (though less powerful) labeling scheme in this paper. Of course, our IDs could easily be replaced by ORDPATHs, or other approaches, if needed.

Simple decimal IDs. A natural choice for IDs are objects that form a totally ordered and dense space such as the rational numbers. Here, we can find a new number m between any two existing numbers a and b, and thus do not need to change a or b to insert a new number between them. Using only these numbers that have a finite decimal representation (such as 0.1203 as opposed to 0.3 periodical 3) we would also gain a feasible string representation. However, there is no reason to keep the base 10. We instead use max-long as a base for our IDs. Concretely, an ID is a list of longs. The order relation is the standard lexicographical order over these lists. As a string representation we add "." between the single "digits". Since one digit already has a large number of values, long lists can easily be avoided: To achieve short lists we use a heuristic similar to the one proposed in [24] that works well in practice. When the initial IDs for a token stream are created, instead of numbering tokens successively, we introduce a gap between numbers (e.g., an increment of 1000). Note that since we only label nodes locally, we can accommodate Maxlong/1000[5] sibling nodes with a one-"digit" ID during the initial labeling pass. With a gap of 1000, e.g., we can also insert a large number of new tokens into existing token lists before we need to add a second "digit". In our tests, e.g., we never had to create an ID with more than one digit.

Splitting input data. Creating key-value pairs for the XMLFS strategy is similar to the Naive strategy with the exception that we create and maintain IDs of XOpen and Data tokens. The XMLFS strategy uses the following data structures for keys and values.

	ID	:=	**List of** Long
	IDXOpen	:=	**Record{** id: ID, t: XOpen**}**
	IDData	:=	**Record{** id: ID, t: Data**}**
	IDToken	:=	IDXOpen | IDData | XClose
Key:	XKey	:=	**Record{** start: ID, end: ID, lp: TIDList**}**
Value:	TIDList	:=	**List of** IDToken

In the key, we use *lp* to include the leading path from the XML root node to the first item in the TIDList stored in the value. As explained above, this information allows data to be written back to the file system based soley on the information encoded in a single key-value pair. Finally, we add the IDs *start* and *end* to the key, which denote fragment delimiters that are necessary for independently inserting data at the beginning or end of a fragment by map calls. For example, assume we want to insert data D before the very first token A in a fragment[6] f. For a newly inserted D, we would need to choose an ID that is smaller to the ID of A. However, the ID must be larger than the ID of the last token in the fragment that comes before f. Since the IDs form a dense space, it is not possible to know how close the new ID $D.id$ should be to the already existing ID of A. Instead, we use the *start* ID in the key, which has the property that the last ID in the previous fragment is smaller. Thus, the newly inserted data item can be given an ID that is in the middle of *start* and $A.id$. Similarly, we store a mid-point ID *end* for insertion at the end of a TIDList.[7]

Fig. 9 and Fig. 6 gives the algorithm for splitting input data into key-value pairs. We maintain a stack *openTags* of currently open collections to keep track of the IDs in the various levels of an XML structure as we iterate over the token list. Whenever we split the stream in fragments (line 11) we compute a mid-point of the previous Token-ID and the current one. The mid-point is then used as an *end* ID for the old fragment, and will later be the *start* ID for the fragment that follows. Note that we reset lastTokenID to "[0]" whenever we open a new collection since our IDs are only local. Moreover, if we split immediately after a newly opened collection, the mid-point ID would be [500] (the middle of [0] and the first token's ID [1000]). It is thus possible to insert a token both at the beginning of a fragment and at the end of the previous fragment.

[5] approximately 9×10^{15} on 32-bit systems

[6] The task might want to insert a modified version of its scope *before* the scope.

[7] When using ORDPATH IDs, we could exploit the so-called *careting* to generate an ID *very* close to another one. However, this technique would increase the number of digits for each such insertion, which is generally not desired.

Map step for XMLFS. Like in the Naive strategy, the map function in the XMLFS approach performs a task's operation $\mathcal{A}$ on its scope matches. Similarly, safe insertions and deletions are required to ensure data integrity in XMLFS. Whenever new data is inserted, a new ID is created that is between the IDs of neighboring sibling tokens. If tokens are inserted as first child into a collection, the assigned ID is between [0] and the ID of the next token. Similarly, if data is inserted as the last child of a node (i.e., the last element of a collection), then the assigned ID is larger than the previous token. Note that when performing safe insertions and deletions only, the opening tokens that are closed in a following fragment cannot be changed. This guarantees that the leading path, which is stored in the key of the next fragment, will still be valid after the updates on the values. Also, XClose tokens that close collections opened in a previous fragment cannot be altered with safe insertions and safe deletions, which ensures that following the leading paths of fragments will maintain their integrity.

After data is processed by a map call, the token list is written to the file system. For this write operation, the *leading path* in the key is used to determine the starting positions for writing tokens. Each data token is written into the current directory using its ID to form the corresponding file name. For each XOpen token, a new directory is created (using the token's ID and label as a directory name) and is set as the current working directory. When an XClose token is encountered, the current directory is changed to the parent directory.

Shortcomings of the XMLFS Strategy

Although the XMLFS approach addresses the bottleneck of having a single reduce step for grouping all data, splitting is still done in a single, serial task, which can become a bottleneck for pipeline execution. Further, even the distributed file system can become a bottleneck when all map calls write their data in parallel. Often only a few (or even only one) master-server administers the distributed file system's directory structure and meta data. As the logical grouping of the XML structure is performed "on the file system", these servers might not be able to keep up with the parallel access. Since both the Google file system and hdfs are optimized for handling a moderate number of large files instead of a large number of (small) files or directories, storing all data between tasks to the file system using the directory-to-XML mapping above can become inefficient for XML structures that have many nodes and small data tokens.

```
1  Split: TIDList input, XPath σ, ID startID, ID endID, TIDList lp
       → [(PKey, TIDList )]
3    TIDList openTags := lp // list of currently open tags
     TIDList oldOpenTags := lp // leading path
5    ID lastEnd := startID // ending ID of last fragment
     ID lastTokenID := startID // ID of last token
7    TIDList splitOut := [ ] // accu for fragment value
     FOREACH token IN input DO
9      IF (openTags / token matches scope σ) AND
         (splitOut ≠ [ ]) THEN
11       ID newend := midPoint(lastTokenID, token.id)
         key := NEW PKey(lastEnd, newend, oldOpenTags)
13       oldOpenTags := openTags
         EMIT key, splitOut // output current fragment
15       lastEnd := newend; splitOut := [ ]
       splitOut.append(token);
17     IF token is IDData THEN
         lastTokenID := token.id
19     IF token is IDXOpen THEN
         openTags.append(token)
21       lastTokenID := [0]
       IF token is Close THEN
23       lastOpenToken := openTags.removeLast()
         lastTokenID := lastOpenToken.id
25   ENDFOR
     key := new PKey(lastEnd, endID, oldOpenTags)
27   EMIT key, splitOut // don't forget the last piece
```

Figure 6: Split for XMLFS & Parallel

Additionally, after data has been stored in the file system, it will be split again for further parallel processing by the next pipeline task. Thus, the file system representation must be transformed back into TIDLists. This work seems to be unnecessary since the previous task used a TIDList representation, which was already split for parallel processing. For example, consider two consecutive tasks that both have the same scope: Instead of storing the TIDLists back into the file system, the first task's map function could directly pass the data to the map function of the second task. However, once consecutive tasks have different scopes, or substantially modify their data to introduce new scope matches, simply passing data from one task's map function to the next would not work. We address this problem in the Parallel strategy defined below.

4.3. Parallel Strategy

The main goal of the Parallel Strategy is to perform both splitting and grouping in parallel, providing a fully scalable solution. For this, we exploit existing partitioning of data from one task to the next while still having the data corresponding to one scope inside a key-value pair. Imagine two consecutive tasks A and B. In case both tasks have the same scope, the data can simply be passed from one mapper to the next if A does not introduce additional scope matches for B, in which case we would further need to split the fragments. In case the scope of task B is a refinement of A's scope, *i.e.*, A's σ_1 is a prefix of B's σ_2, A's mapper can split its TIDList further and output multiple key-value-pairs that correspond to B's invocations. However, it is also possible that a following task B has a scope that requires earlier splits to be undone, for example if task A's scope is //A//B whereas task B's scope is only //A, then the fine-grained split data for A needs to be *partially* merged before it is presented to B's mappers. Another example is an unrelated regrouping: here, spliting and grouping are necessary to re-partition the data for the next scope. Even in this situation, we want to efficiently perform the operation in parallel. We will use MapReduce's ability of grouping and sorting to achieve this goal. In contrast to the Naive Approach, we will not group all the data into one single TIDList. Instead, the data is grouped into lists as they are needed by the next task. As we will show, this can be done in parallel. We now present the necessary analysis of the scopes as well as detailed algorithms for splitting, mapping, and reducing.

Regrouping example. Consider an arbitrarily partitioned TIDList. Fig. 7 shows an example in the second row. Each rectangle corresponds to one key-value pair: The value (a TIDList) is written at the bottom of the box, whereas the key is symbolized at the top-left of the box. IDXOpen and IDXData tokens are depicted with their corresponding IDs as a subscript; XClose tokens do not have an ID. For ease of presentation we use decimal numbers to represent IDs with the initial tokens having consecutively numbered IDs. The smaller text line in the top of the boxes show the leading path *lp* together with the ID *start*. The key's ID *end* is not shown—it always equals the start-ID of the next fragment, and is a very high number for the last fragment. The first box in the second row, for example, depicts a key-value pair with the value consisting of two XOpen tokens, each of which having the ID of 1. The leading path in the key is empty, and the start-ID of this fragment is 0.5. Similarly, the second box represents a fragment that has as value only a token `D[` with ID 1. Its leading path is $\texttt{A}_1\texttt{[}\,\texttt{B}_1\texttt{[}$, and the start-ID of this fragment is 0.5.

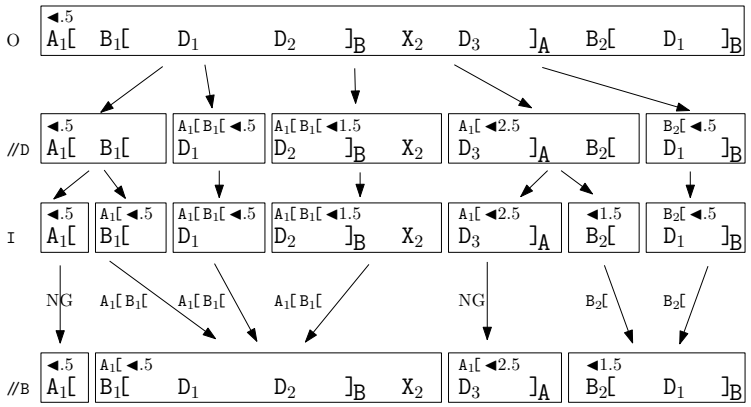


Figure 7: Example of how to change fragmentation from //D to //B in parallel. Since splitting from row two to row three is performed independently in each fragment this step can be performed in the Mapper. Grouping from row three to row four is performed in parallel by the shuffling and sorting phase of MapReduce such that the merge can be done in the Reducers, also in parallel.

Now, consider that the split as shown in the second row of Fig. 7 is the result after the task's action $\mathcal{A}$ is performed in the Mappers. Assume the next task has a scope of //B. In order to re-fragment an arbitrary split into another split, two steps are performed: A split and a merge operation.

Split-Operation. Inside the mapper, each fragment (or key-value pair) is investigated whether additional splittings are necessary to comply with the required final fragmentation. Since each fragment has the leading path, a start and an end-ID encoded in the key, we can use algorithm *Split* as given in Fig. 6 to further split fragments. In Fig. 7, for instance, each fragment in the second row is investigated if it needs further splits: The first and the fourth fragment will be split since they each contain a token `B[`. If there were one fragment with many B subtrees, then it would be split in many different key-value pairs, just like in the previous approach. Note that this split operation is performed

on each fragment independently from each other. We will therefore excecute *Split* in parallel inside the Mappers as shown in the dataflow graph in Fig. 4 and the pseudo-code for the Mapper task in Fig. 8, line 6.

Merge-Operation. The fragments that are output by the split-Operation contain enough split-points such that at most one scope match is in each fragment. However, it is possible that the data within one scope is spread over multiple, neighboring fragments. In Fig. 7, for example, the first B-subtree is spread over three fragments (fragment 2, 3, and 4). We use MapReduce's ability to group key-value pairs to merge the correct fragments in a Reduce step. For this, we put additional *GroupBy* information into the key. In particular, the key and value data structures for the parallel Strategy are as follows:

	GroupBy	:=	**Record{** group: Bool, gpath: TIDList **}**
Key:	PKey	:=	**Record of** XKey **and** GroupBy
Value:	TIDList	:=	**List of** IDToken

Fragments, that do not contain tokens that are within scope simply set the *group*-flag to *false* and will thus not be grouped with other fragments by the MapReduce framework. In contrast, fragments that contain relevant matching tokens will have the group flag set. For these, we use *gpath* to store the path to the node matching the scope. Since there is at most one scope-match within one fragment (ensured by the previous split-operation) there will be exacltly one of these paths. In Fig. 7, we depicted this part of the key in the row between the intermediary fragments I and the final fragments split according to //B: The first fragment, not containing any token below the scope //B, is not going to be grouped with any other fragment. The following three fragments all contain $A_1[\,B_1[$ as *gpath*, and will thus be presented to a single Reducer task, which will in turn assemble the fragments back together (pseudo-code is given in Fig. 8. The output will be a single key-value pair containing all tokens below the node B as required.

```
1  MapParallel: SKey key, TIDList val → [(SKey, TIDList)]
     IF (key.lp / val[0]) matches scope σ
3      val' := 𝒜(val)
     List of (SKey, TIDList) outlist;
5    // split according to the scope σ' of the following step
     outlist := Split(val', σ', key.start, key.end, key.lp)
7    FOREACH (key,fragment) ∈ outlist DO
       EMIT(key, fragment);
9
   ReduceParallel: SKey key, [TIDList] vs → [(SKey, TIDList)]
11   TIDList out := [ ]
     WHILE (val := vs.next())
13     out.append(val);
       key.end := val.end // set end in key to end of last fragment
15   EMIT(key, out)
```

Figure 8: Map and Reduce for Parallel

Order of fragments. The IDs inside the TokenList of the leading path *lp* together with the ID *start* in a fragment's key can be used to order all fragments in document order. Since IDs are unique and increasing within one level of the XML data, the list of IDs on the path leading from the root node to any token in the document forms a global numbering scheme for each token whose lexicographical order corresponds to standard document order. Further, since each fragment contains the leading path to its first token and the ID *start*, a local ID, smaller than the ID of the first token, the leading path's ID-list extended by *start* can be used to globally order the fragments. See, for example Fig. 7: In the third row (labeled with I) the ID lists $0.5 < 1, 0.5 < 1, 1, 0.5 < 1, 2.5 < 1.5 < 2, 0.5$ are ordering the fragments from left to right. We use this ordering for sorting the fragments such that they are presented in the correct order to the reduce functions. Figure 11 shows the definitions for the grouping and sorting comparator used in the Parallel strategy. Two keys that both have the *group* flag set, are compared based on the lexicographical order of their *gpath* entries. Keys that have *group* not set are simply compared. This ensures that one of them is strictly before the other that the returned order is consistent. The sorting comparator simply compares the IDs of the leading paths extended by *start* lexicographically.

```
1  SplitXMLFS: TIDList input, XPath σ → [(PKey,TIDList)]
     CALL Split(input, σ, [0], [maxlong], [ ])
3
   MapXMLFS: PKey key, TIDList val → [(PKey, TIDList)]
5    IF (key.lp / val[0]) matches scope σ
       val' := 𝒜(val)
7    Store val' in distributed file system

9  // No Reduce necessary, Map stores data
```

Figure 9: Split and Map for XMLFS

	Naive	XMLFS	Parallel
Data	XML File	File system representation	Key-value pairs
Split	Centralized	Centralized	Parallel
Group	Centralized by one reducer	Via file system + naming No shuffle, no reduce	Parallel by reducers
Key-Structure	One integer	Leading path with Ids	Leading path with Ids and grouping information
Value-Structure	SAX-elements	SAX-elements with XMLIds	SAX-elements with XMLIds

Figure 10: Main differences for compilation strategies

4.4. Summary of Strategies

Figure 10 presents the main differences of the presented strategies, Naive, XMLFS, and Parallel. Note, that while Naive has the simplest data structures it splits and groups the data in a centralized manner. XMLFS parallelizes grouping via the file system but still has a centralized split phase. The Parallel strategy is fully parallel for both splitting and grouping at the expense of more complex data structures and multiple reduce tasks.

```
1  GroupCompare: SKey keyA, SKey keyB → { <, =, > }
     IF (keyA.group AND keyB.group) THEN
3      // group based on grouping−path
       RETURN LexicCompare(keyA.gpath, keyB.gpath)
5    ELSE
       // don't group (returns < or > for two different fragments)
7      RETURN SortCompare(keyA, keyB)

9  SortCompare: SKey keyA, SKey keyB → { <, =, > }
     // always lexicographically compare "leading path ⊕ start"
11   RETURN LexicCompare( keyA.lp ⊕ keyA.start,
                          keyB.lp ⊕ keyB.start )
```

Figure 11: Group and sort for Parallel strategy

5. Experimental Evaluation

Our experimental evaluation of the different strategies presented above is focused on addressing the following questions: (1) Can we achieve significant speedups over a serial execution? (2) How do our strategies scale with an increasing data load? And, (3) are there significant differences between the strategies?

Execution Environment. We performed our experiments on a Linux cluster with 40 3GHz Dual-Core AMD Opteron nodes with 4GB of RAM and connected via a 100MBit/s LAN. We installed Hadoop [16] on the local disks[8], which also serve as the space for hdfs. Having approximately 60G of locally free disk storage provides us with 2.4TB of raw storage inside the hadoop file system (hdfs). In our experiments, we use an hdfs-replication factor of 3 as it is typically used to tolerate node failures. The cluster runs the ROCKS [26] software and is managed by SunGrid-Engine (SGE) [27]; we created a common SGE parallel environment that reserves computers for being used as nodes in the Hadoop environment while performing our tests. We used 30 nodes running as "slaves", *i.e.*, they run the MapReduce tasks as well as the hdfs name nodes for the Hadoop file system. We use an additional node, plus a backup-node, running the master processes for hdfs and the MR task-tracker, to which we submit jobs. We used Hadoop version 0.18.1 as available on the web-page. We configured Hadoop to launch Mapper and Reducer tasks with 1024MB of heap-space (`-Xmx1024`) and restricted the framework to 2 Map and 2 Reduce tasks per slave node. Our measurements are done using the UNIX `time` command to measure wall-clock times for the main Java program that submits the job to Hadoop and waits until it is finished. While our experiments were running, no other jobs were submitted to the cluster to not to interfere with our runtime measurements.

Handling of Data Tokens. We first implemented our strategies while reading the XML data including the images into the Java JVM. Not surprisingly, the JVM ran out of memory in the split function of the Naive implementation as it tried to hold all data in memory. This happened for as few as `#B` = 50 and `#C` = 10. As each picture was around 2.3MB in size, the raw data alone already exceeds the 1024MB of heap space in the JVM. Although all our algorithms could be implemented in a streaming fashion (required memory is of the order of the depth of the XML

[8] Running hadoop from the NFS-home directory results in extremely large start-up times for Mappers and Reducers.

tree; output is successively returned as indicated by the EMIT keyword), we chose an in-practice-often used "trick" to place references in form of file-names into the XML data structure, while keeping the large binary data at a common storage location (inside hdfs). Whenever we place an image reference into the XML data, we obtain a free filename from hdfs and store the image there. When an image is removed from the XML structure we also remove it from hdfs. The strategy of storing the image data not physically inside the data tokens also has the advantage that only the data that is actually requested by a pipeline step is lazily shipped to it. Another consequence is that the data that is actually shipped from the Mapper to the Reducer tasks is small and thus making even our naive strategy a viable option.

Number of Mappers and Reducers. As described in sec. 2, a *split* method is used to group the input key-valuable pairs into so-called *input splits*. Then, for each input split one Mapper is created, which processes all key-value pairs of this split. Execution times of MapReductions are influenced by the number of Mapper and Reducer tasks. While many Mappers are beneficial to load balancing they certainly increase the overhead of the parallel computation especially if the number of Mappers significantly outnumbers the available slots on the cluster. A good choice is to use one Mapper for each key-value pair if the work per pair is significantly higher than task creation time. In contrast, if the work $\mathcal{A}$ is fast per scope match then the number of slots, or a small multiple of them is a good choice.

All output key-value pairs of the Mapper are distributed to the available Reducers according to a hashing function on the key. Of course, keys that are to be reduced by the same reducer (as in naive) should be mapped to the same hash value. Only our smart approach has more than one Reducer. Since the work for each group is rather small, we use 60 Reducers in our experiments. The hash-function we used is based on the *GroupBy*-part of the *PKey*. In particular for all fragments that have the *group* flag set, we compute a hash value h based on the IDs inside *gpath*: Let l be the flattened list of all the digits (longs) inside the IDs of *gpath*. Divide each element in l by 25 and then interpreted l as a number N to the base 100. While doing so, compute $h = (N \bmod 263) \bmod$ the number of available reduce tasks. For fragments with the group flag not set, we simply return a random number to distribute these fragments uniformly over reducers[9]. Our hash-function resulted in an almost even distribution of all k-v-pairs over the available Reducers.

5.1. Comparison with Serial Execution

We used the image transformation pipeline (Fig. 3), which represents pipelines that perform intensive computations by invoking external applications over CData organized in a hierarchical manner. We varied the number #C of "C" collections inside each "B", *i.e.*, the total number of with "C" labeled collections in a particular input data is #B · #C. Execution times scaled linear for increasing #B (from 1 to 200) for all three strategies. We also ran the pipeline in serial on one host of the cluster. Fig. 12 shows the execution times for #B = 200 and #C ranging over 1, 5 and 10. All three strategies significantly outperform the serial execution. With #C = 10, the speedup is more than twenty-fold. Thus, although the parallel execution with MapReduce has overhead in storing images in hdfs and copying the data from host to host during execution, speedups are substantial if the individual steps are relatively compute intensive in comparison to the data size that is being shipped. In our example, each image is about 2.3MB in size; and blur executed on the input image in around 1.8 seconds, coloring the image once takes around 1 second, the runtime of montage varies from around 1 second for one image to 13 seconds for combining 50 images[10].

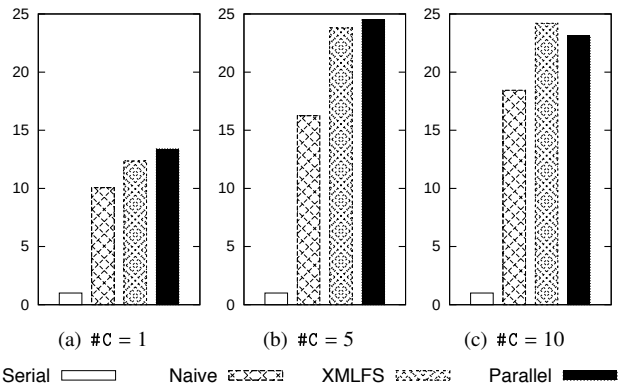
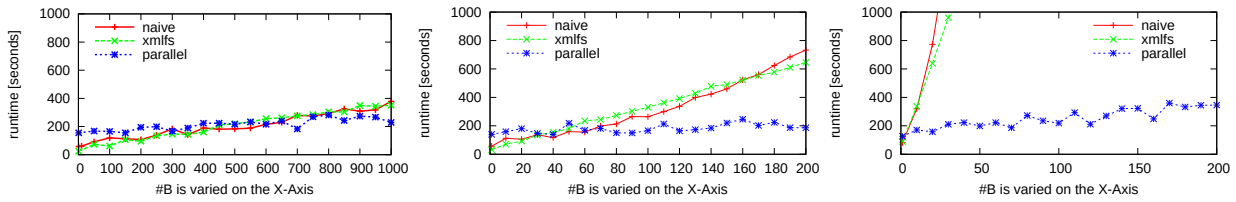


Figure 12: Serial versus MapReduce-based execution. Relative speed-ups to serial execution of image processing pipeline (Fig. 3). All three strategies outperform a serial execution. The achieved speed-ups for #C =1 is only around 13x, wheras in the experiments with more data, more than 20x speed-ups were achieved. #B was set to 200.

[9] Hadoop does not support special handling for keys that will not be grouped with any other key. Instead of shuffling the fragment to a random Reducer, the the framework could just reduce the pair at the closest Reducer available.

[10] There are 5 differently colored images under each "C", with #C = 10, thus 50 images have to be "montaged".



(a) `#C` = 1 (b) `#C` = 10 (c) `#C` = 100

Figure 13: Runtime comparison of the three strategies executing the pipeline given in Fig. 1. On the X-Axis `#B` is varied, Y-axis shows wall-clock runtime of the pipeline. For small XML structures, Naive and XMLFS outperform Parallel since fewer tasks has to be executed. The larger data the more superior is Parallel.

We also experimented with the number of Mappers. When creating one Mappers for each fragment, we could achieve the fastest, and most consistent runtimes (shown in the graphs). When fixing the number of Mappers to 60, runtimes started to have high fluctuations due to so-called "stragglers", *i.e.*, single Mappers that run slow and cause all other to wait for the stragglers' termination.

For this pipeline, all our approaches showed almost the same run-time behavior with Naive performing slightly worse in all three cases. The reason for the similar runtimes is that the XML structure that is used to organize the data is rather small. Therefore, not much overhead is caused by splitting and grouping the XML structure, especially compared to the workload that is performed by each processing step.

5.2. Comparison of Strategies

To analyze the overhead introduced by splitting and grouping, we use the pipeline given in the introduction (Fig. 1). Since it does not invoke any expensive computations in each step, the run times directly correspond to the overhead introduced by MapReduce in general and our strategies in particular. In the input data, we always use 100 empty "D" collections as leaves, and vary `#B` and `#C` as in the previous example.

The results are shown in Fig. 13. For small data sizes (`#C` = 1 and small `#B`) Naive and XMLFS are both faster than Parallel, and XMLFS outperforms Naive. This confirms our expectations: Naive uses fewer Reducers than the Parallel approach (1 vs. 60) even though the 60 reducers are executed in Parallel, there is some overhead involved to launch the tasks and wait for their termination. Furthermore, the XMLFS approach has no reducers at all and is thus as Mapper-only pipeline very fast. We ran the pipeline with `#C` = 1 until `#B` = 1000 to investigate behavior with more data. From approximately `#B` = 300 to around 700, all three approaches had similar execution times. Starting from `#B` = 800, Naive and XMLFS perform worse than Parallel (380s and 350s versus 230s, respectively).

Runtimes for `#C` = 10 are shown in Fig. 13 (b), Here, Parallel outperforms Naive and XMLFS at around `#B` = 60 (with a total number of 60,000 "D" collections). This is very close to the number of 80,000 "D" collections at the "break-even" point for `#C` = 1. In Fig. 13 (c) this trend continues. Our fine-grained measurements for `#B` = 1 to 10 show that the "break-even" point is, again, around 70,000 "D" collections. The consistency in the break-even-point numbers suggests that our parallel strategy outperforms XMLFS and Naive once the number of fragments to be handled and regrouped from one task to the next is in the order of 100,000.

In this experiment, we set the number of Mappers to 60 for all steps as the work for each fragment is small in comparison to task startup times. As above, we used 60 Reducers for the Parallel strategy.

Experimentation result. We confirmed that our strategies can increase execution time for (relatively) compute-intense pipelines. Our image-processing pipeline executed with a speedup of 20x. For XML data that is moderately sized, all three strategies work well, often with XMLFS outperforming the other two. However, if data size increases Parallel clearly outperforms the other two strategies due to its fully parallel split and group.

6. Related Work

Although the approaches presented here are focused on efficient parallelization techniques for executing XML-based processing pipelines, our work shares a number of similarities to other systems (e.g., [28, 29, 30, 31]) for optimizing workflow execution. For example, the Askalon project [31] has a similar goal of automating aspects

of parallel workflow execution so that users are not required to program low-level grid-based functions. To this end Askalon provides a distributed execution engine, in which workflows can be described using an XML-based "Abstract Grid Workflow Language" (AGWL). Our approach, however, differs from Askalon (and similar efforts) in a number of ways. We adopt a more generic model of computation that supports the fine-grain modeling and processing of (input and intermediate) workflow data organized into XML structures. Our model of computation also supports and exploits processes that employ "update semantics" through the use of explicit XPath *scope* expressions. This computation model has been shown to have advantages over traditional workflow modeling approaches [13], and a number of real-world workflows have been developed within the Kepler system using this approach (e.g., for phylogenetics and metagenomics applications). Also unlike Askalon, we employ an existing and broadly used open-source distribution framework for MapReduce (i.e., hadoop) [15] that supports task scheduling, data distribution, and checkpointing with restarts. This approach further inherits the scalability of the MapReduce framework.[11] Our work also significantly differs from Askalon by providing novel approaches for exploiting data parallelism in workflows modeled as XML processing pipelines.

Alternatively, Qin and Fahringer [32] introduce simple data collections (compared with nested XML structures) and collection shipping constructs that can reduce unnecessary data communication (similar approaches are also described in [33, 34, 7]). Using special annotations for different loop constructs and activities, they compute matching iteration data sets for executing a function, and forward only necessary data to this iteration instance. Within a data collection each individual element can be addressed and separately shipped. This technique requires users to specify additional constraints during workflow creation, which can make workflow design significantly more complex. In [14] we address similar problems for XML processing pipelines, however, the necessary annotations in our approach can be automatically inferred based on the the workflow scope descriptions. We complement these approaches here by focusing on strategies for efficient and robust workflow execution through data parallelization strategies, while leveraging data and process distribution and replication provided by hadoop. Thus, through our compilation strategies, we directly take advantage of the operations and sorting capability of the MapReduce framework for data packaging and distribution. MapReduce is also employed in [33] for executing scientific workflows. This approach extends map and reduce operations for modeling workflows, requiring users to design workflows explicitly using these constructs. In contract, we provide a high-level workflow modeling language and automatically compile worfklows to standard MapReduce operations.

Our work also has a number of similarities to the area of query processing over XML streams (e.g., see [35, 36, 37, 38, 39, 40, 41]). Most of these approaches consider optimizations for specific XML query languages or language fragments, sometimes taking into account additional aspects of streaming data. FluXQuery [35] focuses on minimizing the memory consumption of XML stream processors. Our approach, however, is focused on optimizing the execution of compute and data intensive "scientific" functions and developing strategies for parallel and distributed execution of corresponding pipelines of such components. DXQ [42] is an extension of XQuery to support distributed applications, and similarly, in Distributed XQuery [43], remote-execution constructs are embedded within standard XQuery expressions. Both approaches are orthogonal to our approach in that they focus on expressing the overall workflow in a distributed XQuery variant, whereas we focus on a dataflow paradigm with actor abstractions, along the lines of Kahn process networks [44]. A different approach is taken in Active XML [45], where XML documents contain special nodes that represent calls to web services. This approach constitutes a different type of computation model applied more directly to P2P settings, whereas our approach is targeted at XML processing applied to the area of scientific applications deployed within in cluster environments. To the best of our knowledge, our approach is the first to consider applications of the MapReduce framework for efficiently executing XML processing pipelines.

7. Conclusion

This paper has presented novel approaches for exploiting data parallelism for efficient execution of XML-based processing pipelines. We consider a general model of computation for scientific workflows that extends existing approaches by supporting fine-grain processing of data organized via XML structures. Unlike other approaches, our

[11] Which was demonstrated, e.g., by solving the tera-sort challenge, where hadoop successfully scaled to close to 1000 nodes and Google's MapReduce to 4000 nodes on the Peta-sort benchmark.

computation model also supports processes that employ "update semantics" [13]. In particular, each step in a workflow can specify (using XPath expressions) the fragments of the overall XML structure they take as input. During workflow execution, the framework supplies these fragments to processes, receives the updated fragments, combines these updates with the overall structure, and forwards the result to downstream processes. To efficiently execute these workflows, we introduce and analyze new strategies for exploiting data parallelism in processing pipelines based on workflow compilation to the MapReduce framework [15]. While MapReduce has been shown to support efficient and robust parallel processing of large (relational) data sets [46], similar approaches have not been developed that leverage MapReduce for efficient XML-based data processing. The work presented here addresses these open issues by describing parallel approaches to efficiently split and partition XML-structured data sets input to and produced by workflow steps. Similarly, we describe mechanisms for dynamically merging partitions at any level of granularity while maximizing parallelism. Our parallel strategy allows for maximal decentralized splitting and grouping at any level of granularity: If there are more fragments than slots for parallel execution, *i.e.*, hosts or cores, than any re-grouping is performed in parallel. This has been achieved via specific key-structures and MapReduce's sorting support. Furthermore, our framework *also* allows the data to be merged into a very small number of very large partitions. This is in contrast to existing approaches, in which the partitions are either computed centrally (which can lead to bottlenecks) or a fixed partition scheme is assumed. Supporting a dynamic level of data partitioning is beneficial to the workflow tasks as they are provided the data in exactly the granularity they requested via declarative scope expressions. Our experimental results verify the efficiency benefits of our parallel regrouping in comparison to more central approaches (Naive and XMLFS).

By employing MapReduce we also obtain a number of benefits "for free" over more traditional workflow optimization strategies, including fault tolerance, monitoring, logging, and recovery support. As future work, we intend to extend the Kepler Scientific Workflow System with support for our compilation strategies as well as to combine the data parallel approaches presented here with the pipeline parallel and data shipping optimizations presented in [14].

Acknowledgments. This work was supported in part by NSF awards IIS-612326, OCI-722079, ATM-619139, DBI-619060, and IIS-630033. The authors also thank Timothy McPhillips who first suggested and subsequently developed and implemented an approach called COMAD (Collection-Oriented Modeling And Design) [47, 48] based on an assembly line metaphor. Our XML Processing Pipelines are an abstract version of the COMAD idea. They also thank Jianwu Wang and the anonymous reviewers for valuable comments on an earlier draft.

References

[1] G. C. Fox, D. Gannon (Eds.), Concurrency and Computation: Practice & Experience, Special Issue: Workflow in Grid Systems, Vol. 18(10), Wiley, 2006.

[2] I. J. Taylor, E. Deelman, D. B. Gannon, M. Shields (Eds.), Workflows for e-Science: Scientific Workflows for Grids, Springer, 2007.

[3] E. Deelman, D. Gannon, M. Shields, I. Taylor, Workflows and e-Science: An Overview of Workflow System Features and Capabilities, Future Generation Computer Systems 25 (2009) 528–540.

[4] P. Amnuaykanjanasin, N. Nupairoj, The BPEL orchestrating framework for secured grid services, Information Technology: Coding and Computing (ITCC) 1 (2005) 348–353.

[5] J. Fagan, Mashing up Multiple Web Feeds Using Yahoo! Pipes, Computers in Libraries 27 (10) (2007) 10–18.

[6] Intl. Workshop on Web Service Choreography and Orchestration for BPM, Nancy, France (September 2005). URL http://events.deri.at/bpm2005/

[7] T. Oinn, M. Greenwood, M. Addis, M. N. Alpdemir, J. Ferris, K. Glover, C. Goble, A. Goderis, D. Hull, D. Marvin, P. Li, P. Lord, M. R. Pocock, M. Senger, R. Stevens, A. Wipat, C. Wroe, Taverna: Lessons in Creating a Workflow Environment for the Life Sciences, in: Concurrency and Computation: Practice & Experience [1], pp. 1067–1100 (2006) 1067–1100.

[8] B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E. A. Lee, J. Tao, Y. Zhao, Scientific Workflow Management and the Kepler System, in: Concurrency and Computation: Practice & Experience [1], pp. 1039–1065 (2006) 1039–1065.

[9] J. Hidders, N. Kwasnikowska, J. Sroka, J. Tyszkiewicz, J. V. den Bussche, A Formal Model of Dataflow Repositories, in: Data Integration in the Life Sciences (DILS), Vol. 4544, 2007, pp. 105–121.

[10] F. Tian, B. Reinwald, H. Pirahesh, T. Mayr, J. Myllymaki, Implementing a scalable XML publish/subscribe system using relational database systems, SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data (2004) 479–490.

[11] N. Walsh, A. Milowski, XProc: An XML Pipeline Language, W3C Working Draft, April 2007.

[12] I. Avila-Campillo, T. J. Green, A. Gupta, M. Onizuka, D. Raven, D. Suciu, XMLTK: An XML toolkit for scalable XML stream processing, in: PLAN-X, Pittsburgh, PA, 2002.

[13] T. McPhillips, S. Bowers, D. Zinn, B. Ludäscher, Scientific Workflow Automation for Mere Mortals, Future Gener. Comput. Syst. 25 (5) (2009) 541–551.

[14] D. Zinn, S. Bowers, T. M. McPhillips, B. Ludäscher, X-CSR: Dataflow Optimization for Distributed XML Process Pipelines, in: ICDE, 2009, pp. 577–580, also see Technical Report CSE-2008-15, UC Davis.

[15] J. Dean, S. Ghemawat, MapReduce: simplified data processing on large clusters, Communications of the ACM 51 (1) (2008) 107–113.
[16] Hadoop, http://hadoop.apache.org/.
[17] V. Benzaken, G. Castagna, A. Frisch, CDuce: An XML-Centric General-Purpose Language, in: ICFP '03: Proceedings of the eighth ACM SIGPLAN international conference on Functional programming, ACM, New York, NY, USA, 2003, pp. 51–63.
[18] Y. Zhao, J. Dobson, I. Foster, L. Moreau, M. Wilde, A notation and system for expressing and executing cleanly typed workflows on messy scientific data, SIGMOD Rec. 34 (3) (2005) 37–43.
[19] N. Podhorszki, B. Ludäscher, S. Klasky, Workflow Automation for Processing Plasma Fusion Simulation Data, in: WORKS '07: Proceedings of the 2nd workshop on Workflows in support of large-scale science, ACM, New York, NY, USA, 2007, pp. 35–44.
[20] Image-magick, http://www.imagemagick.org.
[21] C. Barton, P. Charles, D. Goyal, M. Raghavachari, M. Fontoura, V. Josifovski, Streaming XPath Processing with Forward and Backward Axes, in: Proceedings of the International Conference on Data Engineering, IEEE Computer Society Press; 1998, 2003, pp. 455–466.
[22] A. Gupta, D. Suciu, Stream Processing of XPath Queries with Predicates, in: Proceedings of the 2003 ACM SIGMOD international conference on Management of data, ACM New York, NY, USA, 2003, pp. 419–430.
[23] D. Borthakur, The Hadoop Distributed File System: Architecture and Design, Apache Software Foundation (2007).
[24] T. Härder, M. Haustein, C. Mathis, M. Wagner, Node labeling schemes for dynamic XML documents reconsidered, Data & Knowledge Engineering 60 (1) (2007) 126–149.
[25] P. O'Neil, E. O'Neil, S. Pal, I. Cseri, G. Schaller, N. Westbury, ORDPATHs: Insert-friendly XML node labels, in: Proceedings of the 2004 ACM SIGMOD international conference on Management of data, ACM New York, NY, USA, 2004, pp. 903–908.
[26] Rocks clusters, http://www.rocksclusters.org/.
[27] W. Gentzsch, Sun Grid Engine: Towards Creating a Compute Power Grid, in: First IEEE/ACM International Symposium on Cluster Computing and the Grid, 2001. Proceedings, 2001, pp. 35–36.
[28] I. Taylor, M. Shields, I. Wang, O. Rana, Triana Applications within Grid Computing and Peer to Peer Environments, Journal of Grid Computing 1 (2) (2003) 199–217.
[29] E. Deelman, Pegasus: A framework for mapping complex scientific workflows onto distributed systems, Scientific Programming 13 (3) (2005) 219–237.
[30] Y. Zhao, M. Hategan, B. Clifford, I. Foster, G. von Laszewski, V. Nefedova, I. Raicu, T. Stef-Praun, M. Wilde, Swift: Fast, Reliable, Loosely Coupled Parallel Computation, in: IEEE Congress on Services, 2007, pp. 199–206.
[31] T. Fahringer, R. Prodan, R. Duan, F. Nerieri, S. Podlipnig, J. Qin, M. Siddiqui, H. Truong, A. Villazon, M. Wieczorek, ASKALON: A Grid Application Development and Computing Environment, International Workshop on Grid Computing (2005) 122–131.
[32] J. Qin, T. Fahringer, Advanced data flow support for scientific grid workflow applications, in: Proceedings of the ACM/IEEE conference on Supercomputing (SC), ACM, 2007, pp. 1–12.
[33] X. Fei, S. Lu, C. Lin, A MapReduce-Enabled Scientific Workflow Composition Framework, in: IEEE International Conference on Web Services (ICWS), 2009, pp. 663–670.
[34] D. J. Goodman, Introduction and evaluation of Martlet: a scientific workflow language for abstracted parallelisation, in: International World Wide Web Conference (WWW), 2007, pp. 983–992.
[35] C. Koch, S. Scherzinger, N. Schweikardt, B. Stegmaier, FluXQuery: An Optimizing XQuery Processor for Streaming XML Data, VLDB (2004) 1309–1312.
[36] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, F. Reiss, M. A. Shah, TelegraphCQ: Continuous Dataflow Processing for an Uncertain World, in: SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data, ACM, New York, NY, USA, 2003, pp. 668–668.
[37] J. Chen, D. J. DeWitt, F. Tian, Y. Wang, NiagaraCQ: A Scalable Continuous Query System for Internet Databases, in: SIGMOD 2000: Proceedings of the 2000 ACM SIGMOD international conference on Management of data, 2000, pp. 379–390.
[38] M. Balazinska, H. Balakrishnan, S. R. Madden, M. Stonebraker, Fault-Tolerance in the Borealis Distributed Stream Processing System, ACM Trans. Database Syst. 33 (1) (2008) 1–44.
[39] C. Koch, S. Scherzinger, N. Schweikardt, B. Stegmaier, Schema-based Scheduling of Event Processors and Buffer Minimization for Queries on Structured Data Streams, in: VLDB '04: Proceedings of the Thirtieth international conference on Very large data bases, VLDB Endowment, 2004, pp. 228–239.
[40] T. J. Green, A. Gupta, G. Miklau, M. Onizuka, D. Suciu, Processing XML Streams with Deterministic Automata and Stream Indexes, TODS 29 (4) (2004) 752–788.
[41] Y. Chen, S. B. Davidson, Y. Zheng, An Efficient XPath Query Processor for XML Streams, in: ICDE '06: Proceedings of the 22nd International Conference on Data Engineering, IEEE Computer Society, Washington, DC, USA, 2006, p. 79.
[42] M. F. Fernández, T. Jim, K. Morton, N. Onose, J. Siméon, Highly distributed XQuery with DXQ, in: SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data, 2007, pp. 1159–1161.
[43] C. Re, J. Brinkley, K. Hinshaw, D. Suciu, Distributed XQuery, Workshop on Information Integration on the Web (2004) 116–121.
[44] G. Kahn, The Semantics of a Simple Language for Parallel Programming, in: J. L. Rosenfeld (Ed.), Proc. of the IFIP Congress 74, North-Holland, 1974, pp. 471–475.
[45] S. Abiteboul, A. Bonifati, G. Cobéna, I. Manolescu, T. Milo, Dynamic XML documents with distribution and replication, in: SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data, 2003, pp. 527–538.
[46] H.-c. Yang, A. Dasdan, R.-L. Hsiao, D. S. Parker, Map-reduce-merge: simplified relational data processing on large clusters, in: SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data, ACM, New York, NY, USA, 2007, pp. 1029–1040.
[47] T. M. McPhillips, S. Bowers, An Approach for Pipelining Nested Collections in Scientific Workflows, SIGMOD Record 34 (3) (2005) 12–17.
[48] T. McPhillips, S. Bowers, B. Ludäscher, Collection-Oriented Scientific Workflows for Integrating and Analyzing Biological Data, Data Integrating for Life Sciences (DILS) (2006) 248–263.