SCAPE

Scalable Preservation Environments

Optimization of Preservation Processes

Author

Alan Akbik (Technische Universität Berlin)

March 2014

This work was partially supported by the SCAPE Project. The SCAPE project is co-funded by the European Union under FP7 ICT-2009.4.1 (Grant Agreement number 270137).

Executive Summary

Preservation of digital media collections has become a scalability issue due to the large quantity of heterogeneous content. The amount of information has reached a point where distributed, shared nothing computing environments are necessary to enable processing in a timely manner. Hadoop is a system that abstracts from the hardware in a computing cluster and presents the user with a programming interface. The user can load data into the system and write programs that execute on the cluster, while not worrying about network transfers or data access. At the same time, governmental organizations like public libraries want to have an intuitive yet powerful graphical user interface to design preservation workflows that are supposed to run on Hadoop, or other scalable processing environments.

In this deliverable, we examine two practical scenarios in order to investigate how to overcome algorithmic limitations of the MapReduce paradigm to optimize execution speeds as well as the real-world applicability of preservation workflows defined in MapReduce. In particular, we give details on the optimization of an algorithmic operation that requires the use of iterations and give details on a large-scale preservation workflow in which we convert very large collections of images from TIFF to the JP2 file format in a distributed environment. We expand on the work reported in deliverable D6.2 in which we began formulating preservation workflows using the Apache Pig dataflow language as a higher order intermediary. Both the PPL translator and the large-scale preservation use case are now formulated in Apache Pig which in turn is then compiled down to MapReduce for execution in a distributed environment. Our findings indicate that preservation workflows can be formulated and executed efficiently within the boundaries of the MapReduce paradigm. Our PPL translator is available online at https://github.com/umaqsud/taverna-to-pig.

Table of Contents

1 Introduction

In this chapter, we first give an overview of the goals of the execution platform in the SCAPE project. We then give a short overview of the status of the work in WP6 and list the contributions of this deliverable.

1.1 Goals of the Execution Platform

The goal of the execution platform is to provide a general means of facilitating the specification and evaluation of complex preservation operations over very large volumes of data. Because preservation in the SCAPE context is generally performed by public organizations such as national libraries, a principal goal of WP6 is to make the process of developing complex preservation workflows that scale to distributed processing environments as easy as possible.

To this effect, we investigated different tools and higher order languages that enable the formulation of complex workflows with regards to the following main desiderata:

Ease-of-use. Firstly, the formulation of complex and scalable workflows should not require an extensive background in distributed processing technologies. Rather, it should be feasible for average persons with a background in computer science to create, modify and execute scalable preservation workflows. This requires abstraction from the commonly used MapReduce paradigm as this is not intuitive to persons without a background in distributed processing technologies.

Workflow elements. Secondly, we require a solution in which as large a range of workflow elements can be expressed as possible. These include sequential workflows, the possibility to define multiple inputs and outputs, join operations, dot and cross operations, the invocation of user defined functions, as well as possibly even more complex workflow elements like decisions or iterations. Generally, the larger the range of workflow elements that we can express, the more diverse preservation workflows we can support.

Efficient execution. Thirdly, even complex workflows should be executed in an optimized way to insure that the available hardware resources are efficiently used. This desideratum derives from use cases in digital preservation that require heavy computation. We see the need for optimization as of major importance for the efficient execution of preservation workflows, and at the same time as one of the major research challenges currently faced by the distributed processing community.

Maintainability. Finally, the underlying technologies should be well-established and well-maintained. A solution that rests on projects that are maintained, and constantly improved, by an active community will allow it to be maintainable after the conclusion of the SCAPE project.

The solutions we present in this deliverable are chosen according to these desiderata.

1.2 Deliverable D6.1

In Deliverable D6.1, we presented the PPL-translator that compiles Taverna workflows to MapReduce, and conducted a set of experiments to investigate the feasibility of executing preservation workflows on a Hadoop [Bor07] cluster. We compared setups in which a workflow was defined in Taverna [OAF+04] and compiled down to either a series of MapReduce jobs [DG08] or executed as a whole in one Map job. In addition, we compared these setups with a manually implemented MapReduce job. In our experiments, we found manually created MapReduce jobs to be significantly faster in execution than the automatic compilation of Taverna workflows into MapReduce. One factor responsible for the difference in runtime is that the automatically compiled version employs beanshells that require repeated Reduce calls to execute the workflow logic.

In general, our experiments revealed challenges with regards to the range of workflow elements that we can support and optimize with MapReduce. We discussed different solutions for executing the workflow logic, such as creating a single beanshell per Reduce class (not per call of Reduce) or to move the logic from the Reduce phase into the Map phase in order to enable the execution of a higher number of concurrent tasks. However, the downside of this approach is that creating dot and cross products would require an additional MapReduce phase before executing the actual logic of the activity.

1.3 Deliverable D6.2

In deliverable D6.2, we described the continued work on the PPL-translator and discussed a demonstration workflow. Building on the results from the work reported on in Deliverable D6.1, we investigated and discussed a number of difficulties concerning the automatic workflow compilation to MapReduce given the desiderata defined in Section 1.1. We evaluated different approaches of how to overcome the limitations of the MapReduce paradigm and enable as many workflow elements as possible while enabling optimization and maintainability.

Next to workflow elements, a major point of discussion was the deployment of Taverna workflow that makes use of user defined functions (UDFs) defined in other programming languages than Java. Building on our experiments, we proposed the use of Apache Pig as a higher order dataflow language to define more complex workflow than we were able to model with the PPL-translator. Our experiments indicated that we could extend the PPL-translator to convert workflow defined in Taverna into PIG scripts which are then compiled to optimized MapReduce programs. This allowed us to model workflow that incorporated joins as well as multiple inputs, which with the previous solution were unavailable.

In addition, by leveraging the considerable work done in the open source community, especially with regards to supported workflow elements and optimized workflow execution, we presented a solution that not only spanned a wider range of workflow elements, but would also be more maintainable due to the open source community. Nevertheless, we noted open issues, such as the use of user defined functions in other programming languages than Java, and workflow elements that are not available even in Apache Pig, such as iterations.

1.4 Contributions of this Deliverable

In this deliverable, we report on the continuation of the work described in deliverables D6.1 and D6.2. We investigate the large-scale execution of a preservation workflows defined in Apache Pig, report on the automatic compilation of Taverna workflows to Apache Pig and investigate the optimization of an algorithmic operation that lies beyond the MapReduce paradigm, requiring the use of iterations. This deliverable therefore consists of four contributions:

Large-scale preservation workflow. Firstly, we introduce a preservation workflow in which a large collection of TIFF images is migrated to the JP2 file format. In order to examine the problem of non-Java UDFs as well as the expressive power of Apache Pig, we define the entire workflows in this higher order dataflow language. We execute the workflows on a collection of images from the Austrian National Library.

Optimization of an interactive workflow. Secondly, we examine the optimization of one particular type of algorithmic operation that typically requires the use of iterations. Iterations are one of the workflows elements which in the MapReduce paradigm cannot be effectively modelled. The goal is to investigate alternative methods of accomplishing this algorithmic operation, namely a low-rank matrix factorization algorithm, with MapReduce in a scalable manner.

Automatic compilation of Taverna workflows to Apache Pig. Thirdly, we present a prototypical system that compiles Taverna workflows into Apache Pig. We discuss advantages of this approach and give details on the capabilities of the system.

Discussion of approach and outline for future work. We conclude by discussing advantages and challenges of defining and executing preservation workflows in MapReduce.

The structure of this deliverable follows the four above mentioned points. In Chapter 2, we outline the motivation for the image migration workflow, introduce the Taverna workflow and the required tools. We illustrate how we create an Apache Pig script that executes this migration workflow in MapReduce and especially focus on the issue of user defined functions in programming languages other than Java. In Chapter 3, we investigate the optimization of an algorithmic operation that requires the use of iterations in MapReduce and present a method that enables us to execute it within the confines of this paradigm. We conduct a set of large-scale experiments to measure the computational cost of our optimized method. In Chapter 4, we present the *taverna-to-pig* compiler and give an outline of its capabilities. Finally, in Chapter 5, we discuss our approach with regards to our desiderata and conclude with an outlook on future developments in the scalable execution of preservation workflows.

2 Large-Scale Preservation Workflow

In this Section, we introduce a large scale preservation workflow implemented in Apache Pig and created for the execution on the data and the Hadoop cluster of the Austrian National Library. We first introduce the goal of the preservation workflow. We then discuss its implementation in Apache Pig as well as the challenges we encountered. Finally, we discuss how this work has influenced our work on automatic workflow compilation and outline further steps. All created code is available online at https://github.com/umaqsud/taverna-to-pig.

2.1 TIFF to JP2 Conversion

In recent years, the JP2 image (JP2000 part 1) file format has gained much attention from the digital preservation community and libraries in particular. It promises similar robustness and quality as the older and more established TIFF image format, but at greatly reduced file size. TIFF images compressed to JP2 using the so-called ``lossless mode'' are only half their original size. When choosing other types of compression, such as so-called ``virtually lossless'' compression, the possible savings in storage space rise considerably. As libraries are faced with preserving ever-growing quantities of high-quality images, compressing images to require less disk storage space while preserving lossless or near-lossless quality is a point of considerable interest.

However, the process of migrating large collections of images as well as the necessary quality control was found to be computationally too expensive for classic hardware environments. We therefore investigated a distributed solution, with a workflow of validation and file format migration defined in MapReduce and executed on a cluster of machines.

2.2 Conversion Workflow

The workflow takes as input a collection of images in TIFF format and outputs their JP2 compressions. The workflow not only converts images, but also checks whether images are valid and whether migrated JP2 images are valid surrogates of the original TIFF images. We make use of the following tools in this workflow, which we need to call as UDFs in the process:

- **Fits.** The File Information Tool Set[1] identifies, validates, and extracts technical metadata for various file formats. It wraps several third-party open source tools, normalizes and consolidates their output, and reports any errors. We use Fits in our workflow to validate whether the images that we aim to migrate are of file type TIFF.

- **JHOVE.** JHOVE[2] is one of the tools wrapped by Fits. It provides functions to perform format-specific identification, validation, and characterization of digital objects. In our workflow, this is the tool that performs the image file type validation.

- **OpenJPEG 2.0.** The OpenJPEG[3] library is an open-source JP2 library developed in order to promote the use of JP2. It is written in C language. We use OpenJPEG to convert TIFF images to JP2 and back.

[1] https://code.google.com/p/fits/
[2] http://jhove.sourceforge.net/
[3] http://www.openjpeg.org/

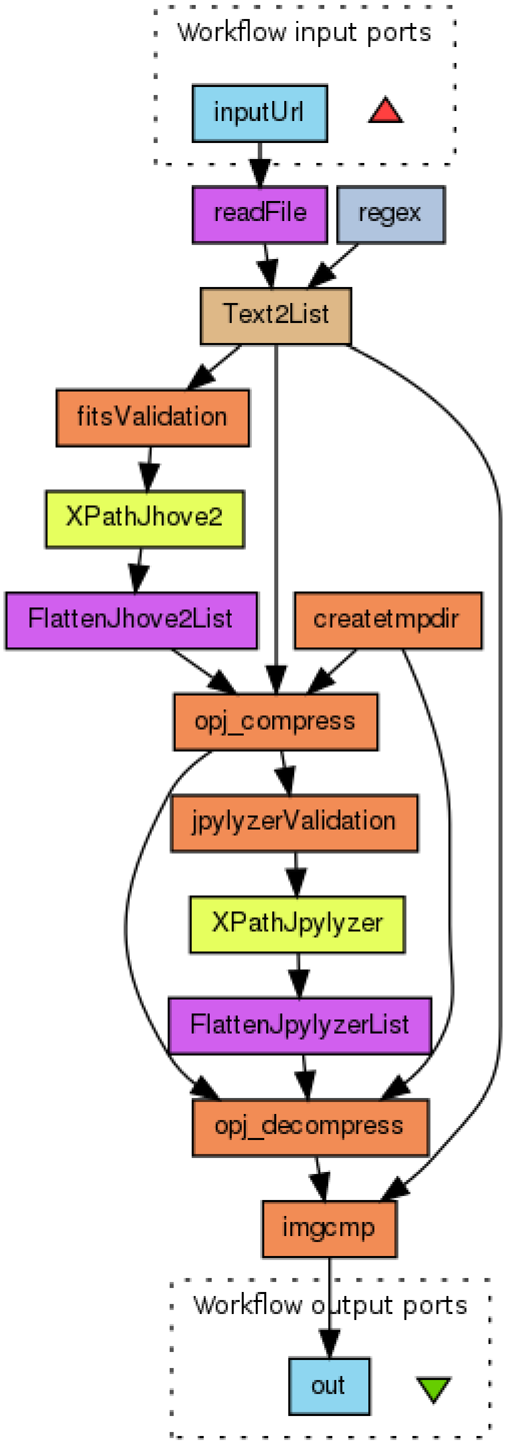


Figure 1: The Taverna workflow for validating and migrating TIFF images to JP2. Orange boxes indicate the invocation of tools: ``fitsValidation'' is the invocation of Fits to validate TIFF images. ``opj_compress'' and ``opj_decompress'' call OpenJPEG to compress or decompress TIFF images. ``jpylyzerValidation'' calls Jpylyzer to validate JP2 images and ``imgcmp'' calls ImageMagick to compare two TIFF images.

- **Jpylyzer.** Jpylyzer[4] is a JP2 validator and properties extractor. It is able to extract the technical characteristics of each image. We use this tool to check whether the JP2 images created with OpenJPEG are valid JP2 images.

- **ImageMagick.** ImageMagick[5] is a software suite to create, edit, compose, or convert bitmap images. It can read and write images in over 100 file formats. We use this tool to compare whether two images of file type TIFF are the same

Please refer to Figure 1, which shows a Taverna workflow that executes the migration. The validation and conversion workflow is as follows: First, we use Fits to check if the TIFF input images are valid instances of the TIFF file format. Fits returns an XML file that contains the validity information, which we extract using an XPath service. We then create a temporary directory where the migrated image files and some temporary tool outputs are stored. If the images are valid TIFF images, we use OpenJPEG to migrate them to the JP2 image file format. We refer to this step as the ``compression'' step, as the image file size is more compressed after the migration. We validate the compressed files using JP2 and again extract the validity information from the XML-based validation report. For this we use the XPath service marked as ``XPathJpylyzer'' in Figure 1.

The migrated JP2 images are converted back to TIFF in order to check if the original file can be restored, a step we refer to as ``decompression''. Finally, we verify that images that were compressed and decompressed are valid surrogates of the original TIFF images by comparing whether original and restored images are identical using ImageMagick.

2.3 Apache Pig Workflow

In this section we illustrate how the workflow is expressed as a script in Apache Pig. We address the challenge that we outlined in the previous deliverable D6.2 of how to effectively invoke local tools in a distributed environment. One option we discussed was to re-implement all required tools in pure Java. The advantage here is that MapReduce is Java-based and all important libraries can then be packaged into a JAR that is distributed amongst all nodes in a cluster upon every invocation of a workflow. However, in discussions with the community we found this approach not to be feasible, due to the number of required tools and their complexity to re-implement.

We therefore decided to use co-called ``local tool invocation''. This requires a cluster of nodes to be set up with all tools that a workflow requires. In the workflow we discuss in this section this means installing the five above mentioned tools on every node in the cluster. The MapReduce implementation then distributes the workload and calls these tools locally at each node and collects their results. The advantage of the chosen approach is that it does not require reimplementation of existing tools. Instead, we require administrators to set up a cluster of machines with identical local tools.

In our Apache Pig implementation, we call these local tools using the STREAM operator to send data through an external script, which in our case are Python scripts that invoke local tools. This way we enable the communication between the MapReduce workflow and the local tools. Apache Pig

[4] https://github.com/openplanets/jpylyzer
[5] http://www.imagemagick.org/

Streaming is similar in function to the Hadoop Streaming API, but more maintainable and enables the integration of such invocations into dataflows. Refer to Figure 2 for an illustration of the script.

```
/* STEP 1 in Workflow */
image_pathes = LOAD '$image_pathes' USING PigStorage()
                    AS (image_path: chararray);

/* STEP 2 in Workflow */
fits_validation = STREAM image_pathes THROUGH jhove_stream
                    AS (image_path:chararray, xml_text:chararray);

/* STEP 3 in Workflow */
jhove2 = FOREACH fits_validation GENERATE image_path, XPathService('$xpath_exp3', xml_text) AS node_list;

flatten_jhove2_list = FOREACH jhove2 GENERATE image_path, FLATTEN(node_list) as node;

/* STEP 4 in Workflow */
flatten_jhove_list_filtered = FILTER flatten_jhove2_list BY node == 'Well-Formed and valid';
flatten_jhove_list_filtered = FOREACH flatten_jhove_list_filtered
                                GENERATE image_path as image_path, '$tmp_path' AS tmp_path;

opj_compress = STREAM flatten_jhove_list_filtered THROUGH opj_stream
                    AS (image_path:chararray, image_jp2_path:chararray);

/* STEP 5 in Workflow */
jpylyzer_validation = STREAM opj_compress THROUGH jpylyzer_stream
                        AS (image_path:chararray, image_jp2_path:chararray, xml_text:chararray);

/* STEP 6 in Workflow */
jpylyzer = FOREACH jpylyzer_validation
                GENERATE image_path, image_jp2_path, XPathService('$xpath_exp2', xml_text) AS node_list;

flatten_jpylyzer_list = FOREACH jpylyzer
                GENERATE image_path, image_jp2_path, FLATTEN(node_list) AS is_valid_jp2;

/* STEP 7 in Workflow */
flatten_jpylyzer_list_filtered = FILTER flatten_jpylyzer_list BY is_valid_jp2 == 'True';
flatten_jpylyzer_list_filtered = FOREACH flatten_jpylyzer_list_filtered
                GENERATE image_path, image_jp2_path, '$tmp_path' as tmp_path;

opj_decompress = STREAM flatten_jpylyzer_list_filtered THROUGH opj_decompress_stream
                AS (image_path:chararray, image_jp2_path:chararray, image_tif_path:chararray);

/* Step 8 in Workflow */
opj_decompress_projected = FOREACH opj_decompress
                GENERATE image_path, image_tif_path, '$tmp_path' AS tmp_path;

compared = STREAM opj_decompress_projected THROUGH compare_stream AS (result:chararray, a:chararray);

STORE compared INTO '$output';
```

Figure 2: The Apache Pig script for the image migration and validation workflow.

2.4 Discussion

In order to examine the challenges with implementing a large scale preservation workflow in our system, we have taken a preservation workflow specified in Taverna and implemented it in Apache Pig. We identified a solution for the challenge identified in the previous deliverable, namely the invocation of tools that lie outside the MapReduce framework. We address this by invoking local tools through the Apache Pig STREAM operator through Python scripts that enable the communication between local tools and the distributed execution. Our tests have shown this to be a viable approach. We are currently preparing large-scale experiments at the Austrian National Library in order to migrate a very large collection of TIFF images to JP2. We make this code publicly available at https://github.com/umaqsud/taverna-to-pig.

We use the solution for calling local tools that we have identified through this use case in the *taverna-to-pig* compiler. This is discussed in more detail in Section 4.

3 MapReduce Optimization

In this chapter, we examine how to overcome limitations in the MapReduce paradigm, namely the unavailability of iterations. We investigate how a basic algorithmic operation that requires the use of iterations may be modelled in MapReduce and optimized using a series of so-called *broadcast-joins* [BPE+10], which are efficiently executable in MapReduce and avoid a lot of the common drawbacks of the Hadoop framework. For evaluation, we conduct experiments on two publicly available datasets of Web data. To test our approach on industrial-scale, we run experiments on a synthetic dataset with more than 5~billion items, mimicking real-world data sizes in Web use cases [Ama12].

3.1 Revisiting Limitations of MapReduce and Iterations

MapReduce [DG08] is a functional paradigm for data-intensive parallel processing on shared-nothing clusters running a distributed file system (DFS). Under this paradigm, the user has to express algorithms as first-order functions supplied to the second-order functions *map* and *reduce*. The framework then automatically parallelizes the program and takes care of details such as scheduling the program's execution on the cluster, managing the inter-machine communication as well as coping with machine failures. Hadoop[6] is a popular and widely deployed open source implementation of MapReduce. In Hadoop, jobs are executed as a pipeline *map-shuffle-reduce*, where the map function is invoked on the input data in the DFS in parallel, the output tuples are grouped by their key and then sent to the reducer machines in the shuffle phase.

The receiving machines merge the tuples, invoke the reduce function on all tuples sharing the same key and finally write the output to the DFS. The shuffle phase is typically the most costly operation as the mappers' output is spilled to disk at first and each reducer downloads its assigned data from every mapper afterwards.

[6] http://hadoop.apache.org/

In general, such a framework is a poor fit for iterative algorithms, as each iteration has to be scheduled as a single MapReduce job with a high start-up cost (potentially up to tens of seconds). Further, the system creates a lot of unnecessary I/O and network traffic as all static, iteration-invariant data has to be re-read from disk and re-processed during each iteration and the intermediary result of each iteration has to be materialized in the DFS.

3.2 Technical Approach

In this section, we give a technical definition of the algorithmic operation that typically requires iterations (Section 3.2.1), illustrate the problems of an implementation in MapReduce (Section 3.2.2) and then discuss in detail how we optimize its execution using a series of broadcast-joins in MapReduce (Section 3.3.3).

3.2.1 Operation

We focus on an algorithmic operation that typically requires the use of iterations, namely matrix factorization using either Stochastic Gradient Descent (SGD) [Zha04] or Alternating Least Squares (ALS) [HKV08]. Both operations are commonly used to analyse Web datasets for interactions, an example of which may be historical interactions between a user and an arbitrary kind of item. Based on patterns found in the historical data, a Collaborative Filtering (CF) algorithm recommends new, potentially high-preferred items to the users.

This is formalized as follows: Let A be a $|C| * |P|$ matrix holding all known interactions between a set of users C and a set of items P. If a user i interacted with an item j, then a_ij holds a numeric value representing the strength of the interaction. CF aims to predict the strength of interactions for unseen data given historical interactions stored in such a matrix.

So-called ``latent factor models'' approaches to CF that leverage a low-rank matrix factorization of the interaction data, have become very popular in recent work [KBV09]. These latent factors point to ``hidden'' dimensions which point to interactions between items. The idea is to approximately factor the sparse $|C| * |P|$ matrix A into the product of two rank r feature matrices U and M such that $A \approx UM$. The $|C| * r$ matrix U models the latent features of the users, (the rows of A), while the $r * |P|$ matrix M models the latent features of the items (the columns of A). A prediction for the strength of the relation between a user and an item (e.g., the preference of a user towards a movie) is given by the dot product $u\_i \cdot m\_j$ of the vectors for user i and item j in the low-dimensional feature space.

A popular technique to compute such a factorization is Stochastic Gradient Descent (SGD) [KBV09, GNHS11, TMG12], which randomly iterates through all observed interactions a_ij, computes the error of the prediction $u\_i \cdot m\_j$ for each interaction and modifies the model parameters in the opposite direction of the gradient. Another technique is Alternating Least Squares, which repeatedly keeps on of the unknown matrices (either U or M) fixed, so that the other one can be optimally re-computed. ALS then rotates between re-computing the rows of U in one step and the columns of M in the subsequent step. Note that such alternating convex optimization techniques require the use of iterations.

3.2.2 Problems with Implementation in MapReduce

For single machine implementations, SGD is the preferred technique to compute a low-rank matrix factorization [GRFST11]. SGD is easy to implement and computationally less expensive than ALS, where every iteration runs in $O((|C|+|P|) * r^3)$, as $|C| + |P|$ linear systems have to be solved. Unfortunately, SGD is inherently sequential, because it updates the model parameters after each processed interaction. Techniques for parallel SGD have been proposed, yet they are either hard to implement, exhibit slow convergence or require shared-memory [GNHS11, TMG12, Lin13].

Following this rationale, we chose ALS for our parallel implementation. Although it is computationally more expensive than SGD, it naturally amends itself to parallelization. When we re-compute the user feature matrix *U* for example, *u_i*, the *i*-th row of *U*, can be re-computed by solving a least squares problem only including *a_i*, the *i*-th row of *A*, which holds user *i*'s interactions, and all the columns *m_j* of *M* that correspond to non-zero entries in *a_i*. This re-computation of *u_i* is independent from the re-computation of all other rows of *U* and therefore, the re-computation of *U* is easy to parallelize if we manage to guarantee efficient data access to the rows of *A* and the corresponding columns from *M*. In the following we refer to the sequence of re-computing of *U* followed by re-computing *M* as a single iteration in ALS.

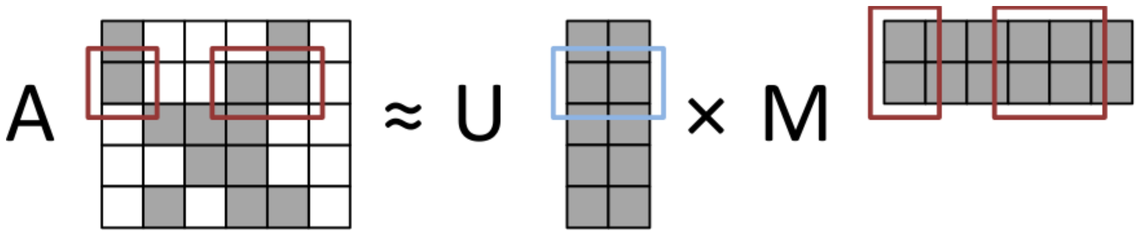


Figure 3: Dependencies for re-computing a row of *U* with Alternating Least Squares.

From a data processing perspective, this means that we have to conduct a parallel join between the interaction data *A* and *M* (the item features) in order to re-compute the rows of *U*. Analogously, we have to conduct a parallel join between *A* and *U* (the user features) to re-compute *M*. Finding an efficient execution strategy for these joins is crucial to optimizing the performance of our proposed parallel solution.

3.2.3 Optimizing MapReduce Execution

When executing joins in a shared-nothing environment, minimizing the required amount of inter-machine communication is decisive for the performance of the execution, as network bandwidth is the scarcest resource in a cluster. For matrix factorization with ALS, we have to alternatingly join *A* with *M* and *U*. In both cases, the interaction matrix *A* is usually much larger than any of the feature matrices. We limit our approach to use-cases where neither *U* nor *M* need to be partitioned (which means they individually fit into the memory of a single machine of the cluster). A rough estimate of the required memory for the re-computation steps in ALS is $max(|C|, |P|) * r * 8$ byte, as alternatingly, a single dense double-precision representation of the matrices *U* or *M* has to be stored in memory on each machine. Even for 10 million users or items and a rank of 100, the estimated required memory would be less than 8 gigabyte, which can easily be handled by today's commodity hardware. Our experiments in Section 3.3 show that despite this limitation, we can handle datasets with billions of data points.

In such a setting, an efficient way to implement the necessary joins for ALS in MapReduce is to use a parallel broadcast-join [BPE+10]. The smaller dataset (*U* or *M*) is replicated to every machine of the cluster. Each machine already holds a local partition of *A* which is stored in the DFS. Then the join between the local partition of *A* and the replicated copy of *M* (and analogously between the local partition of *A* and *U*) can be executed by a map operator. This operator can additionally implement the logic to re-compute the feature vectors from the join result, which means that we can execute a whole re-computation of *U* or *M* with a single map operator.

Figure 4 exemplarily illustrates the parallel join for re-computing U using three machines. We broadcast M to all participating machines, which create a hashtable for its contents, the item feature vectors. A is stored in the DFS partitioned by its rows and forms the input for the map operator (cf. Figure 4 where e.g., *A(1)* refers to partition 1 of *A*). The map operator reads a row *a_i* of *A* (the interaction history of user i) and selects all the item feature vectors *m_j* from the hashtable holding *M* that correspond to non-zero entries *j* in *a_i*. Next, the map operator solves a linear system created from the interactions and item feature vectors and writes back its result, the re-computed feature vector *u_i* for user *i*. The re-computation of *M* works analogously, with the only difference that we need to broadcast *U* and store *A* partitioned by its columns (the interactions per item) in the DFS.

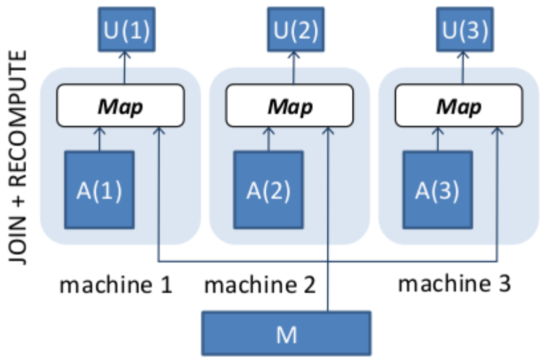


Figure 4: Parallel re-computation of user features by a broadcast-join.

The proposed approach is able to avoid some of the drawbacks of MapReduce and the Hadoop implementation described in Section 3.1. It uses only map jobs that are easier to schedule than jobs containing map and reduce operators. Additionally, the costly shuffle-phase is avoided, in which all data would be sorted and sent over the network. As we can execute the join and the re-computation using a single job, we also do not need to materialize the join result in the DFS, which increases efficiency.

Our implementation contains multithreaded mappers that leverage all cores of the worker machines for the re-computation of the feature matrices and uses JBlas[7] for solving the dense linear systems

[7] http://mikiobraun.github.io/jblas/

present in ALS. The broadcast of the feature matrix is conducted via Hadoop's distributed cache in the initialization phase of each re-computation. Furthermore, we configure Hadoop to reuse the VMs on the worker machines and cache the feature matrices in memory to avoid that later scheduled mappers have to reread the data.

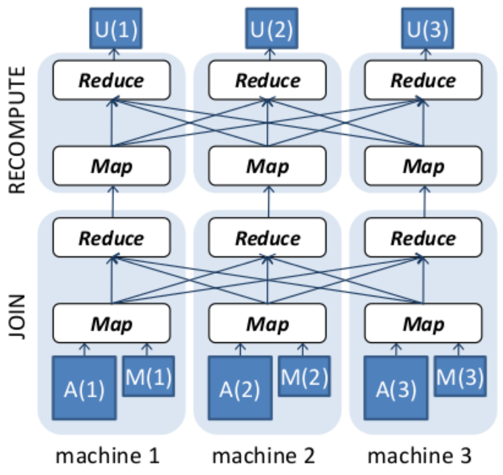


Figure 5: Parallel re-computation of user features by a repartition-join.

The main drawback of a broadcast approach is that every additional machine in the cluster requires another copy of the feature matrix to be sent over the network. An alternative would be to use a repartition join [BPE+10] with constant network traffic and linear scale-out (cf., Figure 5). However, this technique has an enormous constant overhead. Let *n_i* denote the number of interactions, *n_m* the number of machines in the cluster, *n_r* the replication factor of the DFS and *n_e* the number of users or items (we assume that rating values are stored with single precision). If we employ a repartition-join we need two map-reduce jobs per re-computation then. In the first job, we conduct the join which sends the interaction matrix *A* and either *U* or *M_i* over the network, accounting to *n_i * 4 + n_e * r * 8* bytes of network traffic. The result must be materialized in the DFS, which requires another *(n_i * 4 + n_e * r * 8) * n_r* bytes of traffic. The re-computation of the feature matrix must be conducted via a second map-reduce job that sends all the ratings and corresponding feature vectors per user or item over the network, accounting for an additional *n_i * r * 8* bytes. On the contrary, the traffic for our proposed broadcast-join technique depends on the number of machines and accounts to *n_e * r * n_m * 8* bytes.

Applying these estimations to the datasets used for our experiments, the cluster size would have to exceed several hundred machines to have our broadcast-join technique cause more network traffic than a computation via repartition-join. Further, this argumentation only looks at the required network traffic and does not account for the fact that the computation via repartition-join would also need to sort and materialize the intermediate data in each step. We conclude that our approach with a series of broadcast-joins is to be preferred for common production scenarios.

3.3 Experiments

In this section, we experimentally show that our approach is suitable for real-world production settings where an approach has to be compatible to the Hadoop ecosystem and has to run in a few hours to be integratable into analytical workflows. As our purpose is to measure the optimization potential, we solely focus on measuring the runtime of computing a factorization, as the prediction quality of the ALS approach is well studied [HKV08, GNHS11]. For all experiments, we assume that *A* is already stored in the DFS, our measurements do not include the time to copy it there. We use a formulation of ALS that is aimed at rating prediction [ZWSP08].

3.3.1 Experimental Setup

Setup. The cluster for our experiments consists of 26 machines running Java 7 and Hadoop 1.0.4. Each machine has two 8-core Opteron CPUs, 32 GB memory and four 1 TB disk drives.

Datasets. We use two publicly available datasets for our experiments[8]: A set of 100,480,507 ratings given to 17,770 movies by 480,189 users from the Netflix prize [BL07] and another set comprised of 717,872,016 ratings given to 136,736 songs by 1,823,179 users of the Yahoo! Music community[9]. For tests at industrial-scale, we generate a synthetic dataset termed Bigflix using the Myriad data generation toolkit [ATM12]. From the training set of the Netflix prize, we extract the probability of rating each item and increase the item space by a factor of six to gain more than 100,000 movies. Next, we extract the distribution of ratings per user. We configure Myriad to create data for 25 million users by the following process: For each user, Myriad samples a corresponding number of ratings from the extracted distribution of ratings per user. Then, Myriad samples an item from the item probability distribution for each rating. The corresponding rating value is simply chosen from a uniform distribution, as we do not interpret the result anyways, but only want to look at the performance of computing the factorization. Bigflix contains 5,231,536,647 interactions and mimicks the 25 million subscribers and 5 billion ratings, which Netflix recently reported as its current production workload [Ama12].

3.3.2 Experimental Results

Runtime results. We start by measuring the average runtime per individual re-computation of *U* and *M* for different feature space sizes on the Netflix dataset and on the Yahoo Songs dataset (cf., Figure 6) using 26 machines. For Netflix, the average runtime per re-computation always lies between 35 and 60 seconds regardless of the feature space size. This is a clear indication that for this small dataset the runtime is dominated by the scheduling overhead of Hadoop. For the larger Yahoo Songs dataset we see the same behaviour for small feature space sizes and observe that the computation becomes dominated by the time to broadcast one feature matrix and re-compute the other one for larger feature space sizes of 50 and 100. We notice that re-computing *M* is much more costly then re-computing *U*. This happens because in this dataset, the number of users is nearly twenty times as large as the number of items, which means that it takes much longer to broadcast *U*. The experiments show that we can run 37 to 47 iterations per hour on Netflix and 15 to 38 iterations per

[8] Details about our experiments can be found at http://goo.gl/YXj9B

[9] http://webscope.sandbox.yahoo.com/catalog.php?datatype=r

hour on the Yahoo Songs dataset (e.g., ALS typically needs about 15 iterations to converge on Netflix [ZWSP08]). This shows that our approach is easily able to compute factorizations of datasets with hundreds of millions of interactions repeatedly per day.

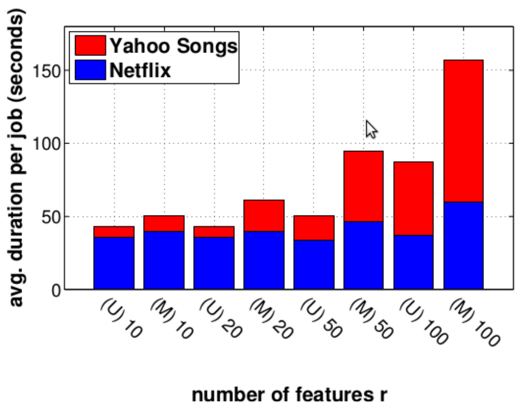


Figure 6: Runtimes on Netflix and Yahoo Songs with different feature space sizes.

Scale-Out Tests. Finally we run scale-out tests on Bigflix: we measure the average runtime per re-computation of U and M during five iterations of ALS on this dataset on clusters of 5, 10, 15, 20 and 25 machines (cf., Figure 7) conducting a factorization of rank ten. With 5 machines, an iteration takes about 19 minutes and with 25 machines, we are able to bring this down to 6 minutes. We observe that the computation speedup does not linearly scale with the number of machines. This behaviour is expected because of the broadcast-join we employ, where every additional machine causes another copy of the feature matrix to be sent over the network. As discussed in Section 3.2.3, on clusters with less than several hundred machines, our chosen approach is much more performant than a repartition-join, although the latter is linearly scaling. With 25 machines, we can run about 9 to 10 iterations per hour, which allows us to obtain a factorization in a few hours, a timeframe completely suitable for a production setting.

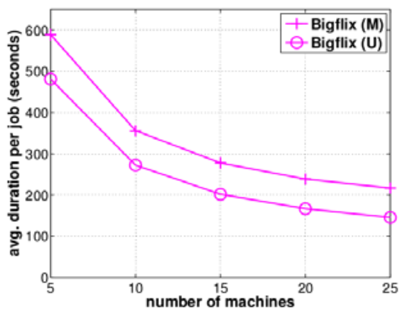


Figure 7: Runtimes on Bigflix with different cluster sizes.

3.4 Discussion

We examined a principal limitation of the MapReduce paradigm regarding iterations and presented a method to enable data-parallel low-rank matrix factorization using Alternating Least Squares using a series of broadcast-joins instead of iterations. We demonstrated how this approach is efficiently scalable approach using MapReduce on a cluster of commodity machines. We experimentally validated our approach on two large datasets commonly used in research and on a synthetic dataset with more than 5 billion data points, which mimics an industry use case. Furthermore, our code has partially been contributed to the open source machine learning library Apache Mahout[10].

Our results show that it is possible to sidestep the use of iterations in some cases and still achieve optimized execution times. Nevertheless, as we illustrate, the formulation of such algorithms in the MapReduce requires considerable effort and will vary from case to case, limiting the generalizability to arbitrary workflows. We believe this to be a promising avenue for future research.

4 Workflow Compilation

In this chapter, we discuss our approach for automatically compiling Taverna workflows to MapReduce. Based on our observations from the previous two chapters, we focus on workflow elements that are available in Apache Pig and integrate tools created in programming languages other than Java as local tool invocations which we call through Python scripts. Our system, the *taverna-to-pig* compiler, is the continuation of the *taverna-to-hadoop* work presented in previous deliverables and is publicly available online at https://github.com/umaqsud/taverna-to-pig.

4.1 The PPL-Translator

In Deliverable D6.1, we presented an overview of the "Program for parallel Preservation Load" (PPL). Its purpose is to enable the parallel execution of preservation workflows. Essentially, the program takes a workflow as input and automatically generates a JAR that can be uploaded to and executed by Hadoop. The resulting Hadoop execution is a linear list of MapReduce jobs produced from the input workflow. The required input for each individual job is either read locally (as provided by the Hadoop framework), or is fetched from other machines in the Hadoop distributed file system (HDFS), if required. The JAR is executed using an Apache Pig script that is generated by the PPL-translator as well.

4.2 Workflow Translation

The workflow translation process generates lines in a Pig Script for each dataflow element in a Taverna script (Section 4.2.1) and handles local tool invocations using Python scripts and Apache Pig streaming functions (Section 4.2.2). It outputs a Pig script and a JAR, plus optionally Python and Shell scripts as shown in Section 4.2.3.

[10] https://mahout.apache.org/

4.2.1 Basic Translation Process

The basic translation process is similar to the *taverna-to-hadoop* compiler introduced in previous deliverables; we use the SCUFL2 API to interpret workflow files created with Taverna. The translator starts at the workflow's output ports and follows the data links that connect the workflow elements backwards and recursively, with the goal of building a linear list of workflow elements. This process is repeated until the workflow's input ports are reached and all workflow elements have been added to the list. This list is then reversed to reflect the actual sequence of workflow elements in the order of dependence, meaning that if an element A depends upon element B, element B appears first in the list.

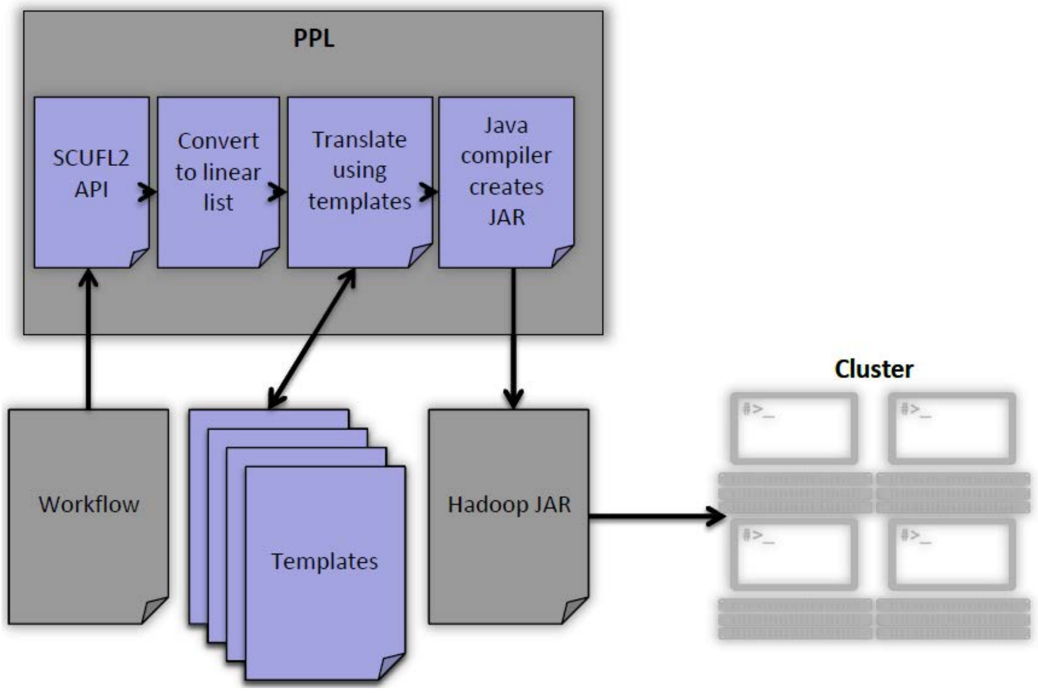


Figure 8: A schematic overview of the workflow compilation process in PPL.

Next, each element in the list is translated individually from a template either into a line of Apache Pig script, or a more complex sequence of Apache Pig script elements and Python scripts in the case of local tool invocations (see Section 4.2.2). An activity in the Taverna workflow that is to be translated can have any number of input ports and any number of output ports. The data location of outputs from output ports is chosen based on the input ports it feeds in to. In turn, the reader that reads the data for those input ports looks in those locations and reads the data. The location consists of a prefix defined by the user, the activity's name, and the input port's name. Finally, Java creates a JAR that can be executed on Hadoop.

This JAR includes all Java UDFs, so that they are available to Hadoop, an example of which is the XPathFunction we introduced in Section 2.3 that extracts values from a given XML file and an Xpath expression. Non-Java UDFs are executed as described in the next section.

4.2.2 Local Tool Invocations

Based on our large-scale preservation use case we discussed in Section 2, we created a solution for calling non-Java tools as part of preservation workflows. Our solution requires the installation of the tools on every node of a cluster. The tools are then called from a Pig script by calling the STREAM operator, which executes a Python script that calls the local tool, collects its results and feeds them back to the distributed workflow. We hereby make use of Apache Pig's streaming functions, which are similar to Hadoop Streaming, but allow us to chain such operations into dataflows.

As the SCUFL2 API does not currently support local tools, we implemented an XML script parser for the purpose of identifying local tool invocations in Taverna scripts. The parser extracts all relevant information from the Taverna scripts so that we can generate Python scripts for parallel execution. These Python scripts are then called through the STREAM operator in the Apache Pig workflow. Per local tool, we generate one such Python script that handles the communication.

4.2.3 Generated Files

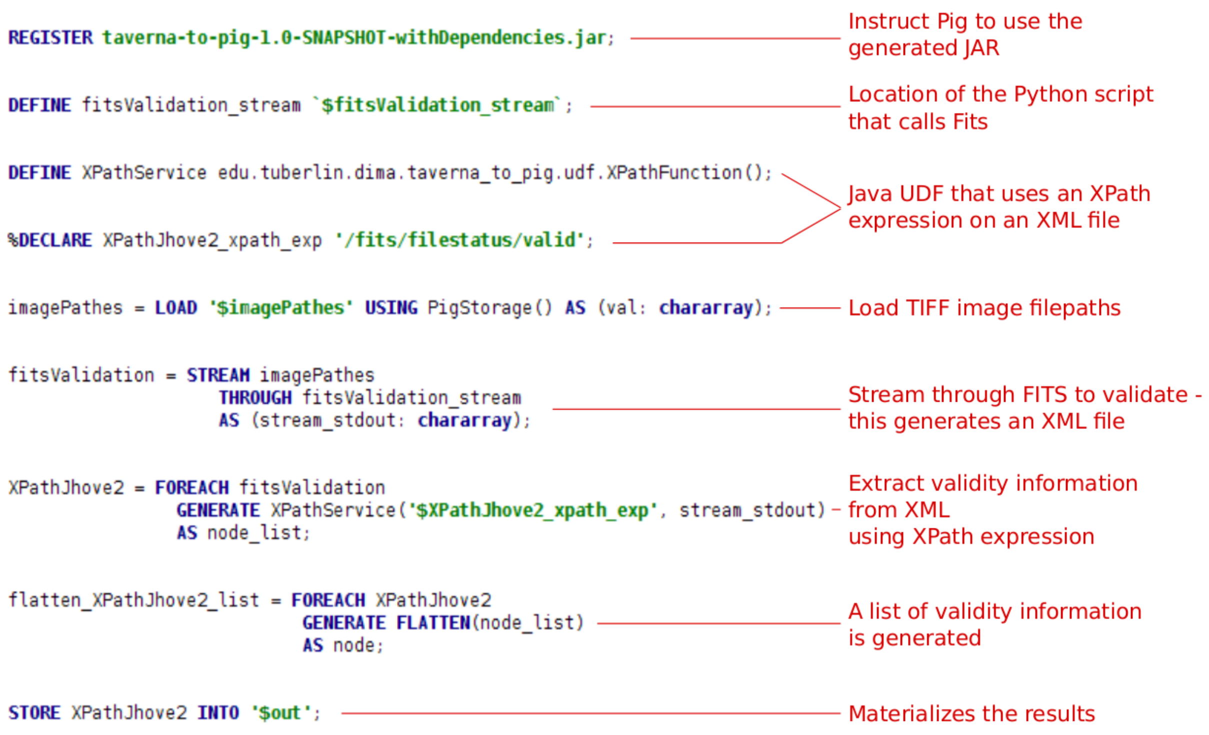
The *taverna-to-pig* compiler generates a total of 4 distinct file types for a given Taverna workflow: Firstly, it generates a JAR that includes all Java UDFs and workflow logic. This JAR is uploaded to a cluster for parallel execution. Secondly, it generates an Apache Pig script that includes all dataflow elements and connects UDFs, input and output ports. Thirdly, it generates a *parameter file*, which includes all parameters that may need to be executed for workflow execution. Such parameters are all the variables that need to be provided to start the original Taverna workflows, such as the location of the input data and the location where the workflow output should be written to. Finally, it also generates a series of Python scripts, one script for each local tool.

For the execution of the workflow, the Pig script needs to be executed by passing the parameter file. For simplicity purposes, the *taverna-to-pig* compiler generates an execution script that handles all parameterization.

4.3 Example Workflow Translation

In this section, we illustrate the workflow conversion process with a simple example script in Taverna that makes use of each of the dataflow elements described in this section: A Java UDF that uses an XPath expression to read an XML file, a non-Java UDF that is called through a Python script and Pig Streaming, and parameters that are passed to define input and output locations. Refer to Figure 9 for an illustration and explanation of the generated script.

We use the *taverna-to-pig* compiler to generate this script from a simple Taverna workflow that reads image files, executes JHOVE through Fits to determine the validity of the TIFF files, and extracts the validity information using a simple Java UDF. It therefore comprises the first two steps of the workflow described in Section2.



```
REGISTER taverna-to-pig-1.0-SNAPSHOT-withDependencies.jar;

DEFINE fitsValidation_stream `$fitsValidation_stream`;

DEFINE XPathService edu.tuberlin.dima.taverna_to_pig.udf.XPathFunction();

%DECLARE XPathJhove2_xpath_exp '/fits/filestatus/valid';

imagePathes = LOAD '$imagePathes' USING PigStorage() AS (val: chararray);

fitsValidation = STREAM imagePathes
                 THROUGH fitsValidation_stream
                 AS (stream_stdout: chararray);

XPathJhove2 = FOREACH fitsValidation
              GENERATE XPathService('$XPathJhove2_xpath_exp', stream_stdout)
              AS node_list;

flatten_XPathJhove2_list = FOREACH XPathJhove2
                           GENERATE FLATTEN(node_list)
                           AS node;

STORE XPathJhove2 INTO '$out';
```

Instruct Pig to use the generated JAR

Location of the Python script that calls Fits

Java UDF that uses an XPath expression on an XML file

Load TIFF image filepaths

Stream through FITS to validate - this generates an XML file

Extract validity information from XML using XPath expression

A list of validity information is generated

Materializes the results

Figure 9: Example of a Pig script generated using the *taverna-to-pig* compiler for a given Taverna workflow. The left hand side are the Pig scripting elements, while the right hand side provides explanations to what logic each step in this script executes.

As we can see in Figure 9, the first line points the Pig script to the JAR that is generated by the *taverna-to-pig* compiler, which includes all Java UDFs. The second line defined the location of the Python script that is used to call Fits. This line is a parameter - when deploying the script to a cluster all such Python scripts need to be placed in the distributed file system so that they can be executed at each node. The parameter must point to this location accordingly.

The third and fourth lines define the Java UDF as well as the XPath expression that the UDF uses to extract validity information from a given XML file. This XPath expression is generated as a parameter that is already set, i.e. from the Taverna workflow we already know the XPath expression. Nevertheless, we expose the path as a parameter so that users can modify it should this be required.

Line 5 then reads the input, which in this case is a list of file paths that point to TIFF images. This list is then streamed through Fits in line 6, generating a list of XML validity reports that are stored in the variable ``fitsValidation''. In line 7, the FOREACH operator is called on this list, meaning that the XPath UDF is executed on each XML validity report in turn. This extracts a nested list of true/false values that reflect the validity of each image. In line 8, this list is flattened for future processing and written into a file in line 9.

4.4 Discussion

Our chosen approach for the PPL-translator rests on Apache Pig as this allows us to define complex workflows in a scripting environment, which are both optimized and independent of the underlying execution platform. Based on the preservation workflow that we have outline in Section 2.3, we have developed a solution for the invocation of local tools that utilizes Apache Pig streaming functions. Stated goal of the Apache Pig project is to expand this streaming support from the invocation of command line tools to UDFs written in programming languages other than Java. As both this project as well as comparable projects are currently undergoing heavy development, it is likely that the research community will produce more efficient methods for calling non-Java UDFs in the near future. By basing our approach on a project that is actively supported by the data parallel processing community we believe that this will allow our approach to benefit from these developments and be maintainable after the conclusion of the SCAPE project.

The same benefits are also likely for the continued expansion of supported dataflow elements in Apache Pig. Our chosen approach enables us to make use of joins, decisions and possibly iterations as these are being developed by the Apache Pig community. We preview that as both the Taverna and the Apache Pig development advance, they will naturally meet, so that arbitrary workflows defined in Taverna will be easily expressible in Apache Pig. Our work on the *taverna-to-pig* compiler demonstrates this and serves as an important connector between these works.

5 Conclusion

We have investigated the large-scale execution of preservation workflows and their optimization using the MapReduce paradigm. We have given a detailed analysis of our optimization approach and conducted a large-scale experimental evaluation on industry datasets. Based on the results of our experiments, we have implemented the *taverna-to-pig* compiler that converts Taverna workflows to Apache Pig.

Some points of discussion remain, which we have addressed in this deliverable and to which we make recommendations for future directions. Firstly, even using a higher order abstraction some workflow elements remain difficult to model and optimize. In this deliverable, we have focused on iterations as an important workflow element that lies beyond what currently established distributed processing paradigms can effectively model. We have outlined a method of sidestepping the need for iterations using broadcast-joins which is applicable to some workflows, but is difficult to generalize to a principled solution for arbitrary iterative workflows. We have however also shown that custom optimization is possible and very good execution time results can be achieved if the effort is expended. We believe that such research - and indeed recent research from a number of projects that complement or extend MapReduce - will hold the key to more principled approaches for distributed execution of iterations and other workflow elements.

Secondly, from a perspective of easily deploying distributed workflows on MapReduce, the current state-of-the-art favours Java-based approaches, as MapReduce itself is written in Java. The crucial point is that Java-based approaches can be effectively executed in a cluster of machines simply by distributing the JAR at MapReduce start-up. When calling tools that lie outside this framework, the situation becomes more complex, as now the execution must deal with external dependencies.

Crucially, this means that non-Java tools need to be installed on *every node* in a cluster of machines, with exactly the same setup, before a workflow can be executed. Reading our desideratum of ``ease-of-use'' a preferred way would be to execute workflows without such an administrative effort of setting up software and installing libraries on large clusters of machines. This, however, is only possible for pure-Java solutions.

Current development by the research community is addressing both these points. By basing our approach for the data parallel execution of Taverna workflows on Apache Pig, a project that is actively supported by the industry and research community, we will benefit from these developments. Our solution will therefore be maintainable after the conclusion of the SCAPE project and is well-placed to leverage future breakthroughs in parallel processing.

6 References

[Ama12] Xavier Amatriain. Building industrial-scale real-world recommender systems. In Proceedings of the sixth ACM conference on Recommender systems, pages 7-8. ACM, 2012.

[ATM12] Alexander Alexandrov, Kostas Tzoumas, and Volker Markl. Myriad: scalable and expressive data generation. Proceedings of the VLDB Endowment, 5(12):1890{1893, 2012.

[BL07] James Bennett and Stan Lanning. The netflix prize. In Proceedings of KDD cup and workshop, volume 2007, page 35, 2007.

[Bor07] Dhruba Borthakur. The hadoop distributed file system: Architecture and design, 2007.

[BPE+10] Spyros Blanas, Jignesh M Patel, Vuk Ercegovac, Jun Rao, Eugene J Shekita, and Yuanyuan Tian. A comparison of join algorithms for log processing in mapreduce. In Proceedings of the 2010 ACM SIGMOD International Conference on Management of data, pages 975-986. ACM, 2010.

[DG08] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplied data processing on large clusters. Communications of the ACM, 51(1):107-113, 2008.

[GNHS11] Rainer Gemulla, Erik Nijkamp, Peter J Haas, and Yannis Sismanis. Large-scale matrix factorization with distributed stochastic gradient descent. In Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining, pages 69-77. ACM, 2011.

[GRFST11] Zeno Gantner, Steen Rendle, Christoph Freudenthaler, and Lars Schmidt-Thieme. Mymedialite: A free recommender system library. In Proceedings of the fifth ACM conference on Recommender systems, pages 305-308. ACM, 2011.

[HKV08] Yifan Hu, Yehuda Koren, and Chris Volinsky. Collaborative Filtering for implicit feedback datasets. In Data Mining, 2008. ICDM'08. Eighth IEEE International Conference on, pages 263-272. IEEE, 2008.

[KBV09] Yehuda Koren, Robert Bell, and Chris Volinsky. Matrix factorization techniques for recommender systems. Computer, 42(8):30-37, 2009.

[Lin13] Jimmy Lin. Mapreduce is good enough? if all you have is a hammer, throw away everything that's not a nail! Big Data, 1(1):28-37,2013.

[OAF+04] Tom Oinn, Matthew Addis, Justin Ferris, Darren Marvin, Martin Senger, Mark Greenwood, Tim Carver, Kevin Glover, Matthew R Pocock, Anil Wipat, et al. Taverna: a tool for the composition and enactment of bioinformatics workflows. Bioinformatics, 20(17):3045-3054, 2004.

[TMG12] Christina Teflioudi, Faraz Makari, and Rainer Gemulla. Distributed matrix completion. In ICDM, pages 655{664, 2012.

[Zha04] Tong Zhang. Solving large scale linear prediction problems using stochastic gradient descent algorithms. In Proceedings of the twenty-first international conference on Machine learning, page 116. ACM, 2004.

[ZWSP08] Yunhong Zhou, Dennis Wilkinson, Robert Schreiber, and Rong Pan. Large-scale parallel collaborative Filtering for the netflix prize. In Algorithmic Aspects in Information and Management, pages 337-348. Springer, 2008.