Design and Implementation of a Library Metadata Management Framework and its Application in Fuzzy Data Deduplication and Data Reconciliation with Authority Data

Martin Czygan
Leipzig University Library
Beethovenstraße 6
04107 Leipzig
martin.czygan@uni-leipzig.de

Abstract: We describe the application of a generic workflow management system to the problem of metadata processing in the library domain. The requirements for such a framework and acting real-world forces are examined. The design of the framework is layed out and illustrated by means of two example workflows: fuzzy data deduplication and data reconciliation with authority data. Fuzzy data deduplication is the process of finding similar items in record collections that can not be matched by identifiers. Data reconciliation with authority data takes a data source and enhances the metadata of its records – mainly authors and subjects – with available normed data. Finally, the advantages and tradeoffs of the presented approach are discussed.

1 Introduction

The system described in this paper is a result of a few product iterations on the problem of library metadata management. The problem emerged during project *finc*[1] – a project for the development of search engine based discovery interfaces for university libraries in Saxony – where a couple of heterogeneous data sources needed to be processed and merged in certain, sometimes convoluted, ways. The main contribution of this paper is to report on core requirements (section 3) of such a system, and how most of these requirements can be met by adapting a generic workflow management framework (sections 4 and 5). Subsequently, two example workflows from the library domain are described in short (section 6), followed by a discussion (section 7), where key tradeoffs are identified.

2 Related Work

There are a number of open source software products for workflow management, dependency graph handling and data processing. A prominent example of a tool that utilizes

[1]More information about the project can be found on its homepage at `http://finc.info` and in [Laz12].

a dependency graph is make[2], a build tool for software projects[3]. More data-oriented projects include Oozie[4], Askaban[5], joblib[6], Taverna[7] and luigi[8]. Features that made luigi a compelling choice[9] as the base for our system are a) a small code base[10], b) efficient command line integration and c) a single, lightweight notation to express tasks and their relationships.

3 Requirements

An easily formulated requirement of a metadata management application for libraries is the ability to work with different data sources and to combine, filter and transform the data, as it traverses the system. This requirement implies some kind of formalism of a unit, on which transformations can be applied. Typical notions for these units would be a file on the filesystem or on the web, a single record or a stream of records. This requirement is broad and leaves a lot of design decisions untouched. To add additional design constraints, we enumerate a few non-functional requirements.

Consistency Transformations on data should happen in a transactional manner or should in other ways be guaranteed to be atomic[11], that is a transformation should either be applied to a data element or not. Atomicity should be enforced at different data granularity levels, from a single field in a single record, that is rewritten, to sets of potentielly tens of millions of records, that need a transformation to be applied on all of its elements.

Composability It is desireable to design the data elements or the processing units in ways that support reuse. There are several approaches to this: A certain data package can be used to build various workflows atop. In that way, workflows do not need to recompute certain states or shapes of the data over and over again. Another way of reuse would be

[2]`http://www.gnu.org/software/make/`

[3]Other open source projects with similar scope are – among others – rake, ant, maven, gradle, scons, waf.

[4]From `https://oozie.apache.org/`: “Oozie is a workflow scheduler system to manage Apache Hadoop jobs. Oozie Workflow jobs are Directed Acyclical Graphs (DAGs) of actions.”

[5]From `https://azkaban.github.io/`: “Azkaban is a batch workflow job scheduler created at LinkedIn to run Hadoop jobs. Azkaban resolves the ordering through job dependencies and provides an easy to use web user interface to maintain and track your workflows.”

[6]From `https://pythonhosted.org/joblib/`: “Joblib is a set of tools to provide lightweight pipelining in Python.”

[7]From `http://www.taverna.org.uk/`: “Taverna is an open source and domain-independent Workflow Management System – a suite of tools used to design and execute scientific workflows and aid in silico experimentation.”

[8]From `https://github.com/spotify/luigi`: “Luigi is a Python package that helps you build complex pipelines of batch jobs. It handles dependency resolution, workflow management, visualization, handling failures, command line integration, and much more.”

[9]The system design is decribed in more detail in section 5.

[10]luigi 1.0.16 contains 5125 LOC, which included the task abstractions, a scheduler, support for hdfs, hadoop, hive, scalding, postgres, Amazon S3, date, configuration and compression handling and task histories.

[11][HR83] describes atomicity as one of the four characteristics of a transaction: “Atomicity. It must be of the all-or-nothing type [...], and the user must, whatever happens, know which state he or she is in.”

the application of a single task or processing unit on several data streams. For example, different data sources could be fetched in form of compressed files from an FTP server, so the task of fetching and extracting these files would present itself as a template task, because for several different data sources the processing would be similar up to a fixed number of parameters.

Efficiency Many data sources are updated in intervals and recomputations of data derived from that data source in turn should happen as frequently as the data source itself is updated. In general, a workflow, that encompasses a number of data sources should be executed as soon as any one of the constituent data sources changes in a significant way. Conceptually, one can view the combination of a number of data sources as a data source itself with the update times consisting of the update times of all subsumed data sources along with a total order.

Ease of deployment, agile development and change management The application should be easily relocatable[12] and should bootstrap its data environment as autonomously as possible. This proves to be useful for development since a simple setup enables developers to mirror a production environment on a development machine and therefore reduce the gap between both setups.

Development of new workflows and changes to existing workflows should not affect established setups, that might be part of a production system. It should be possible to distribute tasks sensibly among members of a team.

Testability, transparency and introspection Writing code that deals with millions of records and complicated workflows can be a challenge, as would be writing unit tests for those workflows. It would be desireable to be able to unit test components as well as to write integration tests for complete workflows with the help of mock services.

Most workflow management systems will break down a multi-stage workflow into several subtasks. One should be able to introspect the shape of the data along the way at the different steps to support transparency and debuggability.

4 Forces

There are a number of forces around the problem of library metadata management, of which a few should be introduced in this section.

[12]This is certainly interesting in a cloud setup, where machines may be spun up and down on a regular basis.

Data formats The format of the data to be processed varies widely. While there are standardized data formats for libraries such as MARC[13], MAB[14] or RDA[15], there is still enough room for variation. Other data sources, which might be of interest to the library community, have not yet been standardized or are not necessarily curated with the goal of standardization. Typical formats are XML (which in turn can come in many flavors to express similar things), JSON, HTML, N-Triples, CSV or plain text. All these formats can furthermore be compressed or span multiple files.

Data access As we described for the data formats the accessibility of the data varies widely, too. Typical examples are FTP servers, metadata harvesting protocols such as OAI-PMH [CLW08], Web-APIs, E-Mail attachments, database or index connections.

Update intervals and update signalling Typical for the use case, data sources are updated in intervals. The update intervals can range from a few seconds over days to many weeks, months or years. In other words they can range from real-time to a singular update. In these remarks of we do not consider real-time, but only cases, where the update granularity does not underrun 24 hours[16].

An interesting observation concerns the way, the availability of an update is signalled. This can happen in many ways as well, with two broad categories being explicit or implicit. An example for explicit signalling would be a file name containing a date or some other indicator of succession, such as a serial number. Implicit signalling means that the data source will not convey the availability of updates, unless it is preprocessed to reveal this information. An example for implicit signalling would be OAI-harvesting, where one would need to perform an actual request and a comparison with existing records to find out, whether updates are available[17].

Another aspect during update processing is the way in which an update to a data source is expressed. There are at least two ways in which this can be achieved in practice. A simple way to express an update to a data source is to provide a complete dump.

Using deltas is also common practice. Given an initial dump or a delta against an empty state, subsequent changes to the data source are expressed as changes, which need to be applied to the previous state of the data source. The upside of this approach is the limitation of the update to the affected records, the downside – which sometimes results in significant overhead in the amount of processing – is the need to keep all the metadata packages available if one wants to recompute some state of the data source after a number of deltas. The way in which the deltas are formulated can differ between data sources. One data source might provide an update or deletion list as an extra file, while other data sources ship updated or deleted records with a flag set at a predefined record field. The

[13]`http://www.loc.gov/marc/`

[14]`http://www.dnb.de/DE/Standardisierung/Formate/MAB/mab_node.html`

[15]`http://www.rda-jsc.org/rda.html`

[16]The workflow framework ships with a parameters class, that allows update intervals of 60 minutes. Below that, the inherently batch-oriented nature of the framework might not prove supportive.

[17]A *from* parameter in the request is required, which in turn can only be derived from some previous harvest.

variety of these notions does not assist in the definition of a consistent abstraction layer, when one tries to generalize the event of an update over a bigger number of data sources[18].

Processing speed It is desirable to utilize modern multicore computers by implementing systems, which take advantage of parallel processing whenever feasible. It should be noted that while certain parts of a workflow might not be suitable for parallelization, others might and the processing framework would ideally be able to identify these parts by itself and schedule the tasks accordingly.

5 Design

In this section, the overall system design is outlined along with the description of how the implementation addresses the requirements and forces.

Building blocks The system described in this report is based upon an open source software library called luigi[19], developed by Spotify[20] – a music streaming service. This library provides basic abstractions for workflow management and execution[21] and embodies many of the desired properties listed in section 3. The contribution of this report is the description of how this generic framework for implementing workflows can be adapted for the library domain.

The basic building block of the framework is the notion of a task, which is both simple and powerful. A task has a couple of essential properties: It has zero, one or more requirements, which are tasks themselves; furthermore a task has a body of code, which corresponds to the business logic that needs to be executed; finally every task has one or more targets[22]. Aside these three core notions each task has zero or more parameters. Every workflow is decomposed into a number of tasks. Tasks are implemented and wired together in the source code itself. Figure 1 gives an impression of an emerging depedency graph. Workflow execution is realized through command line integration, which supports interactive use as well as automation with standard tools such as cron[23].

A central scheduler, implemented as a deamon, handles task scheduling and visualisation. While the notion of a target supports many different implementations[24], the majority of the artefacts are files.

[18]This fact suggests, that the notion of an update to a data source alone might not be the abstraction level sweet spot.

[19]`https://github.com/spotify/luigi`

[20]“Luigi was built at Spotify, mainly by Erik Bernhardsson and Elias Freider, but many other people have contributed.“, `https://github.com/spotify/luigi`

[21]The framework comes with more than just the basic abstractions. Most notable there are wrappers around HDFS [Bor08] and Hadoop [Whi09] to support cluster computing as well.

[22]While it is possible to create tasks with multiple targets, it is recommended, that each task has exactly one artefact as output.

[23] `http://unixhelp.ed.ac.uk/CGI/man-cgi?cron`

[24]The community implemented targets for Amazon Redshift, MySQL, Postgres, Elasticsearch.

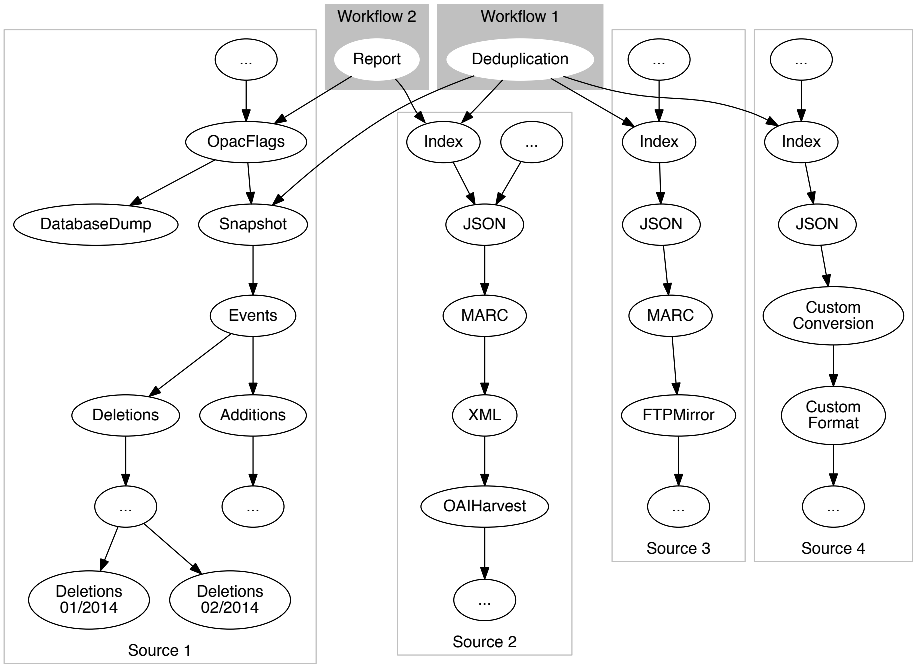

Figure 1: An impression of a dependency graph (with simplified task names and without task parameters). Different sources will differ in their internal complexity, depending on various factors, such as update intervals and signalling. Workflows can be built upon tasks from many sources.

Consistency The framework itself does not enforce consistency, but makes it easy to implement it. Following the rule, that each task will create an artefact only if the task has been run successfully, it is possible to carry over the consistency from tasks to workflows. A workflow (which is just a set of tasks) is successfully completed, only if every task executed successfully. When dealing with files, the atomicity can be enforced in the task by using the POSIX rename interface [IG13][25].

On the other side, it is possible to repeatedly execute workflows. Since each task has a defined target (e.g. a file), it will not execute the task again, if it has been run successfully in the past (e.g. if the output file exists) with the same set of parameters.

Idempotence and atomicity in the building blocks aid in reducing the number of states the system can be in, which benefits the resilience of the overall system and also helps to reason about what happened and what failed.

[25]“This rename() function is equivalent for regular files to that defined by the ISO C standard. Its inclusion here expands that definition to include actions on directories and specifies behavior when the new parameter names a file that already exists. That specification requires that the action of the function be atomic.“

Composability The notion of a task underlines the provision of small units. While there is no limitation in what a task can do and how many lines of code it can contain, it is convenient to keep the amount of work and responsibility for a single task limited. In an evaluation of about 300 tasks, that implement several different workflows in our system, the average number of lines of code (LOC) inside the body of a task was 13.7. Figure 2 shows the distribution of the lines of code over the tasks. The number of tasks that constitute a workflow ranges between a few tasks up to several hundreds of tasks.

The nature of the framework supports composability also by letting the developer implement generic tasks, that can be reused by other tasks by adding these generic tasks to the workflow or derive special tasks from them. Examples for generic tasks, that have been implemented in our system are tasks for mirroring FTP files and directories, harvesting OAI-PMH interfaces or indexing documents.

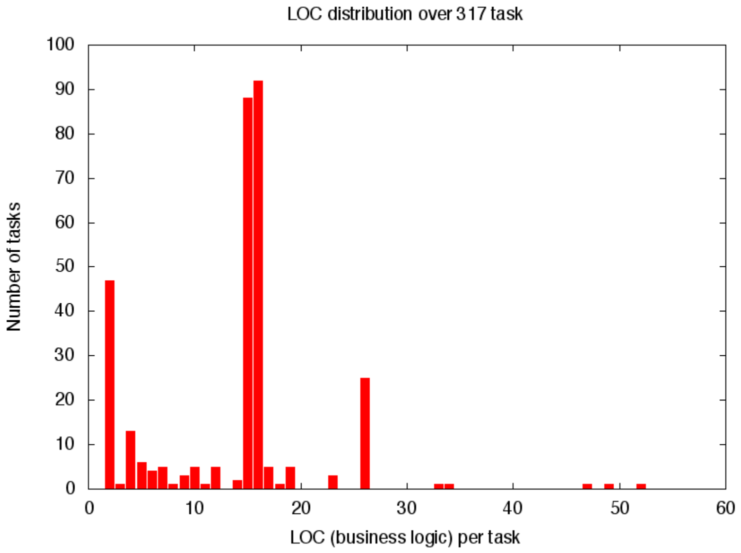

Figure 2: Distribution of lines of code (LOC) over 317 tasks. The average LOC per task (business logic only) is 13.7.

Efficiency Two important types of efficiency are efficiency concerning tasks and efficiency in the context of parallelism. Efficiency with regard to tasks means that tasks and workflows are only executed, if there has been an update to the data source in question. As described in section 4, the update intervals can vary widely between data sources. It can be sufficient to parameterize a task by date and to have it execute in regular intervals (e.g. in weekly, monthly or quarterly intervals). Other tasks might have some latest state, which can only be evaluated by looking at the data source itself. For these cases, a special parameter class has been implemented, that is able to map any given date to the closest date in the past, on which an update occured for a data source. In this way a task can respond with an correct output artefact for arbitrary dates. The business logic of the task is only executed if absolutely necessary, therefore contributing to the efficiency of the overall setup.

The second type of efficiency is handled by the framework, which can use any number of workers. A worker corresponds to a separate OS process. Since the tasks in the system form a dependency graph, the framework is able to determine, which tasks are parallelizable and will spawn separate processes to execute them, given the user allows more the one worker to run.

Ease of deployment, agile development and change management Most of the system's artefacts are files, which are organized under a single directory. This makes it relatively easy to backup or replicate the metadata – both original and derived – to other machines by copying a single directory. The software consists of a single python package that can be installed with standard tools. One strength of the setup is the ability to almost completely bootstrap a metadata infrastructure, given a valid configuration has been provided. This, again, is possible because the tasks form a dependency graph. This feature supports agile development as well. A developer does not need to know, how a certain state or artefact of the metadata (e.g. a list of ISBNs from a few different data sources or a list of records from a data source, that have been updated in the last three weeks) is provided, as long as there is a task in the system that provides that specific artefact.

During development it has been our observation, that certain tasks are referenced more often than other. One could call these tasks *stable points*, since they underpin more workflows than other tasks. On the other hand, there are tasks, that merely serve as an intermediate step in some longer computation, making them less important in the overall setup. These kind of metrics[26] can also inform development. We saw that investing time in the refinement of the stable points in the architecture yielded performance benefits for many workflows, whereas it was often possible to contract less important intermediate steps into a single larger task for the sake of performance – without sacrificing too much conceptual clarity by agglutinating many steps within a single task.

Testability, transparency and introspection As described in section 5, the average number of LOC for a task is relatively low. Furthermore, a task has a defined set of inputs (often just a single file) and outputs (per recommendation just a single file). This helps writing unit tests, since a task resembles a function with a number of arguments and most of the time no or only a few references to any kind of state outside itself. Since the tasks themselves can be tested with moderate efforts, complicated workflows, which span hundreds of tasks, become manageable as well.

The mostly file-based setup of the system facilitates another aspect: transparency. Most tasks produce a single output and this output has a location[27] on the file system. In our

[26]Each task records its performance and reports it to the user. The benchmarks are implemented with python decorators, making it easy to extend this system into a metrics database that could monitors performance characteristics over time, should the need arise.

[27]We extended the built-in file system abstractions of the framework to be able to almost complety hide the

system, we wrote small wrapper scripts, that help to inspect tasks with a single command, abstracting from the real filename, referring to the task only by its name and parameters[28]. Another addition concerns documentation. By combining the documentation facilities[29] and the dynamic nature of the python programming language with the implicit dependency information encoded in the task, it is possible to autogenerate comprehensive command line documentation, similar to a UNIX man page[30].

Deployment setup The software is a single python package, splitted into two modules: One contains the data sources, another the higher level workflows. The data source modules take care of the data from its origin (e.g. FTP servers, web sites, databases), to some stable point, e.g. in our case this is mostly a single JSON file that represents the current state of the data source or a corresponding Elasticsearch index. The workflow modules work with those data sources, modeling a variety of things, such as data deduplication, reporting or data enrichment and reconciliation. The python package itself has a couple of external dependencies for database access, index access, ISBN handling, tabular data manipulation among other things. Common tasks and utilities have been factored out into a separate package.

6 Example workflows

Two workflows, that have been implemented atop various data sources are briefly described in this section. The important observation in both examples is the fact, that non-trivial tasks, that originate in the library data domain, can be made manageable, by builing upon a powerful data structure like a dependency graph.

6.1 Fuzzy data deduplication

As a first example we describe briefly the process of fuzzy data deduplication. We selected three data sources with about five million records as a first target. There are three workflows involved to get each data source into an Elasticsearch index. Two data sources work with regularly updated files on FTP-Servers, one data source uses a more involved scheme involving dumps, deltas and additional lists for deletions. Of the two data sources that use FTP, one conveys updates in form of complete dumps, whereas the other uses a set of deltas. As described in section 5, a stable point in our setup is an Elasticsearch index with a document structure, that is relatively homogenous across data sources. These indices are kept up-to-date as the data source commands it.

underlying directory structure from the developer or user.

[28]This feature is especially useful, when a new data source is added to the system and needs to be explored with the help of a few initial tasks.

[29]`http://legacy.python.org/dev/peps/pep-0257/`

[30]`http://manpages.bsd.lv/history.html`

For the fuzzy deduplication a dual approach of cosine and ngram similarity is used. First a list of ids is generated, for which similar items are collected with the help of the a heuristic query[31], based on record titles. The titles and authors of the candidates are then broken into trigrams and compared. Records with a similarity score above a threshold are then reported as fuzzy duplicates. Additional fields are evaluated in a postprocessing step. Records, for example, that do not share a similar edition statement are removed from the list of duplicates[32].

6.2 Data reconciliation with authority data

Another requirement concerns data enrichment and data reconciliation. Since our setup uses over 30 data sources, the quality of the metadata is not necessary alike. Especially there are data sources, that can benefit from the knowledge already encoded in form of library metadata in other data sources. One example is the use of high quality authority data produced and published by the German National Library in form of the Integrated Authority File (GND) [BNP11]. This authority file contains among other things naming variants for entities, for example authors or composers. On the other hand there are data sources in our metadata landscape, that carry a single naming variant of a person, thus making it impossible to find the record by using a valid name that is not part of the record. The workflow here takes the entries from the GND and successively compares the naming variants to names and entities found in the less rich metadata source, generating suggestions for metadata enrichment. In a quality control postprocessing step, the suggestions are checked for plausibility, by evaluating additional information about the subject[33].

7 Discussion

There are a couple of tradeoffs involved in a setup, as described in this report. The first and probably most significant is the space overhead of a file based approach. Similar to a map-reduce [DG08] style processing, where each intermediate step is persisted to disk, before another phase in the workflow picks it up, the disk space overhead is considerable. Depending on the concrete workflows, the increase in storage requirements in our setup can be up to tenfold – compared to the size of the raw original data source. It is possible to view the data generated by the tasks of the workflow as a cache, computing parts of the process once and reducing the number of computations required for subsequent steps. One example for this incremental computing[34] approach is the handling of data sources,

[31]This query uses Lucene's *MoreLikeThis* facility which uses a *bag-of-words* model to find similar documents.

[32]The definition of a duplicate will vary in different situations.

[33]We implemented this workflow for the University of Music and Theatre Leipzig. We could utilize information about the profession of a person, encoded in the GND taxonomy, to weigh suggestions. For example there are many people named Bach, of which not all are composers. In a second step, we used biographical data, also encoded in the GND, to further clean the list of suggestions.

[34]*Reactive Imperative Programming with Dataflow Constraints* [DFR11] describes incremental computing as follows: "When the input is subject to small changes, a program may incrementally fix only the portion of the

that ship updates with deltas. In one case, each delta package needs a few processing steps (e.g. extraction, conversion, filter). In another step all deltas and their derived artefacts are considered to create a snapshot of the data source. The amount of work that needs to be done when a new delta arrives is minimal, once the derived artefacts of the previous deltas are in place – making it possible to quickly create snapshots of that data source.

This extensive caching facilitates the solution of other problems in a straightforward manner. One example would be the computation of change reports for data sources. Since many stages of the metadata has been or can be recomputed, the comparison of the state of the metadata between two arbitrary time instances becomes surprisingly simple. One just calls the same task with two different date parameter values and extracts the difference[35].

A disadvantage of the presented approach is the discipline required to implement the tasks. Each tasks must be written in an manner, so that it supports atomicity and idempotency. Also not every task is suitable for this functional paradigm, although from empirical evidence we are able to conclude, that many different real world tasks are. Persisting intermediate steps can lead to inefficiencies, since performance could benefit from keeping data in memory for a certain amount of time or for a certain chain of tasks[36].

8 Summary

In summary, the advantages of a system like the one described in this report are the following: A workflow composed of a number of smaller tasks facilitates testable, reusable components. If it is easy to reason about a single task, complex workflows spanning multiple data sources can stay manageable. Supplying tasks with parameters for time can help with the isolation of update behaviours and can shield the developer from the idiosyncrasies of a data source by providing stable points in the architecture, which enable a multitude of workflows. Finally, trading extensive space requirements for a gain in simplicity and composability seems to be a worthwhile tradeoff in the library metadata domain.

References

[BNP11] Renate Behrens-Neumann and Barbara Pfeifer. Die Gemeinsame Normdatei – ein Kooperationsprojekt. *Dialog mit Bibliotheken*, (1):37–40, 2011.

[Bor08] Dhruba Borthakur. HDFS architecture guide. *HADOOP APACHE PROJECT http://hadoop. apache. org/common/docs/current/hdfs design. pdf*, 2008.

output affected by the update, without having to recompute the entire solution from scratch. “

[35]If, for example, the two tasks emit single column data, one can load the two artefacts into two sets and perform a set difference.

[36]This line of critique seems to apply to the map-reduce style framework Hadoop as well [Hea14]: “Spark is able to execute batch-processing jobs between 10 to 100 times faster than the MapReduce engine according to Cloudera, primarily by reducing the number of writes and reads to disc.“ Another

[CLW08] Michael Nelson Carl Lagoze, Herbert Van de Sompel and Simeon Warner. The Open Archives Initiative Protocol for Metadata Harvesting, 2008.

[DFR11] Camil Demetrescu, Irene Finocchi, and Andrea Ribichini. Reactive imperative programming with dataflow constraints. In *ACM SIGPLAN Notices*, volume 46, pages 407–426. ACM, 2011.

[DG08] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[Hea14] Nick Heath. Faster, more capable: What Apache Spark brings to Hadoop, 2014.

[HR83] Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys (CSUR)*, 15(4):287–317, 1983.

[IG13] The IEEE and The Open Group. The Open Group Base Specifications Issue 7, IEEE Std 1003.1, 2013 Edition, 2013.

[Laz12] Jens Lazarus. *Projekt finc Ein Open Source Discovery System für sächsische Hochschulbibliotheken*, volume BIS - Das Magazin der Bibliotheken in Sachsen 5(2012)2 S. 72 - 76. Saechsische Landesbibliothek- Staats- und Universitaetsbibliothek Dresden Dresden, 2012. article.

[Whi09] Tom White. *Hadoop: The Definitive Guide: The Definitive Guide*. O'Reilly Media, 2009.