From Metadata Catalogs to Distributed Data Processing for Smart City Platforms and Services: A Study on the Interplay of CKAN and Hadoop

Robert Scholz(✉), Nikolay Tcholtchev, Philipp Lämmel, and Ina Schieferdecker

Fraunhofer Institute for Open Communication Systems (FOKUS), Berlin, Germany
{robert.scholz,nikolay.tcholtchev,philipp.lammel, ina.schieferdecker}@fokus.fraunhofer.de

Abstract. Smart Cities are emerging based on the idea of provisioning and processing large amounts of urban data for various use cases. Thereby, Urban Data Platforms are usually employed to accumulate and expose the large amounts of governmental (i.e. public sector), sensor, static and real-time data in order to enable the community to create valuable applications and services for future Smart Cities. Hitherto, the Open Data initiative was seen as the key driver to providing large amounts of data within a city. Open Data platforms employ so-called data registries in order to keep track of the available datasets at various sources spread throughout the city, with CKAN currently being among the most popular data catalog software worldwide. With the emergence of frameworks for large scale distributed computing and storage, such as Hadoop and the belonging distributed file systems (HDFS), there is an inherent need for bridging the worlds of metadata catalogs and distributed data processing towards the goal of providing sophisticated urban ICT services. The current paper constitutes a first attempt on this new field, by prototyping and evaluating components that enable the collaboration and interplay between CKAN and Hadoop/HDFS. This interplay is realized through extensions to CKAN and its harvesting process and its benefits are demonstrated by belonging case studies.

Keywords: Smart Cities · Open Data · Distributed processing Hadoop · CKAN

1 Introduction

One pivotal concern for the creation of real Smart Cities is the establishment of a working data processing pipeline. Typical Smart City solutions require the integration of big and diverse data on (potentially) distributed systems. A first step towards this goal was and is the ongoing process of establishing city-wide metadata catalogs that index available datasets from all the different contributing stakeholders of the Smart City environment. This creates a single point of access for most of the available (open or specifically licensed) data for a particular city.

D. Ferguson et al. (Eds.): CLOSER 2017, CCIS 864, pp. 115–136, 2018.
https://doi.org/10.1007/978-3-319-94959-8_7

In a previous work [1], the authors highlighted a lack of research efforts concerning the seamless integration of the existing metadata hubs and the available data processing engines throughout a city. As of then, required data(-sets) needed to be manually collected, transferred onto the processing system and kept up-to-date, thereby forfeiting some of the potential advantages offered by the aforementioned cataloging systems. The authors therefore suggested a novel concept for integration of these two types of systems and consequently implemented an extension that closed this gap by providing the automated integration between the *Comprehensive Knowledge Archive Network* (CKAN) [2] and the Hadoop Distributed File System (HDFS) [3], serving as exemplary systems from each domain. This concept and its belonging prototype were denoted as *HdfsStorer* extension. The prototype builds on the CKAN platform and utilizes the core structure of a CKAN extension.

The current paper is a follow-up to this previous publication from the CLOSER 2017 proceedings [1]. The present update puts a stronger emphasis on the framework of projects in which the extension was developed and highlights some practicalities encountered during development of the presented extension. Additionally, the paper describes newly added functionalities that haven't been part of the previous publication.

The aforementioned extension was developed in the late stages of the German Governmental Data-Portal (GovData) project [4], whose main aim was the creation of a unified, country-wide metadata catalog for governmental data from municipalities, city councils and other federal and state entities. The project itself was part of a series of open government projects by the chief executive body of the German government (Bundesregierung) in cooperation with the IT-Planungsrat, which is responsible for the coordination of the collaboration of the federal government and federal states in the area of IT. As a key asset defined within the Open Data strategy of the German Government [5], the portal was launched in February 2013.

Such a metadata catalog holds only references to the data, along with other attributes, such as licensing, file size and -type or time of the last update. The actual data remains available only over the web portals of the belonging institutions. The aggregation of metadata accomplished either through manual addition of metadata entries or through "harvesting" of other metadata catalogs (e.g. separate catalogs of federal states or cities) or similar sources. CKAN is a major open source platform for metadata cataloging and was used to implement the GovData metadata engine. In CKAN, harvesting is realized through a dedicated extension [6], which provides harvesting plugins for specific standardized metadata formats. Additional plugins may be developed to enable the harvesting of sources that provide their metadata in a different format.

The previously developed extension established a means for integrating CKAN as a metadata store with the powerful capabilities of Hadoop [7], in order to enable the efficient handling of large (open) datasets in urban environments. This first extension allowed for the entirety of all harvested datasets by intercepting resource addition and update events. This is now extended by a CKAN harvester plugin that provides a means for selecting the desired type of datasets for each harvested source (i.e. other metadata catalogs) in order to allow for the realization of more dedicated use cases and resource (storage space and bandwidth) usage optimization.

The HDFS was chosen as the system for data storage for a range of reasons. As integral part of the Hadoop framework for big data processing, it enables parallel data-local processing of data distributed over a cluster of machines. Efficient replication of datasets, aside of bestowing failure resistance, allows for the dynamical addition and removal of (virtual) machines during runtime and thus provides the scalability necessary for the creation of different applications that make use of ever growing datasets and serve a continuously growing user base. This can be facilitated by cluster coordinators such as Zookeeper [8]. The stored data can be processed by a multitude of HDFS-compatible software solutions, such as batch processing engines (MapReduce [9]), in-memory solutions (Flink [10], Spark [11]), integrated with stream processing (Spark Streaming, Storm [12]) or used within graph processing frameworks (Giraph [13]) as well as data base/warehousing systems (HIVE [14], Impala [15], HBase [16]). This allows for the development of small to large scale applications within Smart Cities or other environments. Hadoop-based machine learning systems may additionally benefit from the ease of creation of homogenous big datasets – ensured through restricted harvesting of many data sources - that are usually required for their initial training phase.

The remainder of this paper is divided as follows: The next section presents already existing relevant work from the fields introduced above. Sections 3 and 4 describe the envisioned solutions, namely the *HdfsStorer* and the CKAN harvester plugin for filtering. Thereupon, the implementations of both extensions are described in detail (Sects. 5 and 6, respectively). A realized proto-use case is given in Sect. 7. In complement to that, Sect. 8 outlines a possible real-world application of said extensions within an aspiring Smart City. Lastly, the authors provide a discussion of aspects presented hitherto and summarize the contributions of the paper.

2 Related Work

A multitude of projects worldwide – such as the Open Data portal of Japan [17] or the municipal data hub in Rio de Janeiro [18] - use CKAN for metadata cataloging. Its major competitor is Socrata [19]. Proprietary solutions such as Konema [20] exist as well. CKAN instances find their application on all levels of geographical and administrative granularity, ranging from the regional (i.e. in Berlin Open Data Portal [21]), over the federal level (GovData [4]) to multinational communities (such as the European Data Portal [22]). Fraunhofer FOKUS has played a major role in the conceptualization and development of these latter three Open Data portals. These portals aim to provide a single point of access to predominantly governmental data, whereby the data itself remains on the web portals of the belonging institutions (also denoted as data providers). GovData indexes data from a multitude of providers i.e. municipalities, city councils, or federal institutions such as the Federal Statistical Office of Germany, which are either directly publishing the data (i.e. via custom Application Programming Interfaces [APIs]) or providing access to it through their own metadata catalogs. The European Data portal is one position downstream and itself harvests the metadata hubs of the individual countries, such as those of Germany (GovData.de), Portugal (dados.gov.pt) or Estonia (opendata.riik.ee). Private entities such as companies also

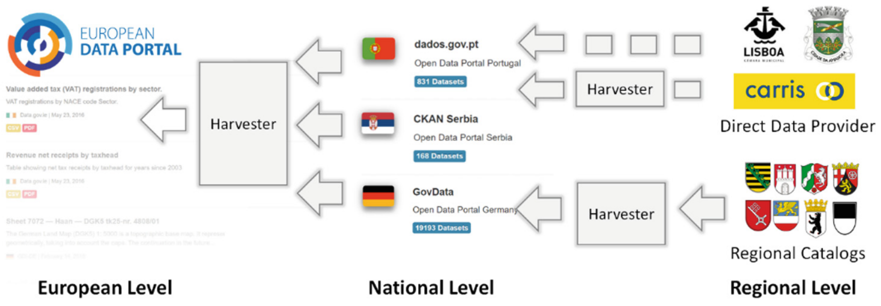

Fig. 1. Schematic overview of the metadata accumulation for the European Open Data Portal. Regional catalogs receive exclusively direct input (not shown). Direct Data Provider may either provide their data in a harvestable format (akin to the regional catalogs) or their data can be manually registered in the corresponding catalog (dotted arrow).

provide their data to some of the harvested data hubs upstream. This harvesting pipeline is shown in Fig. 1.

To enable the efficient integration of metadata from different data hubs and data sources, a set of specifications and standards pertaining harvesting protocols, interfaces and metadata schemes have been developed. These include the Open Archives Initiative Protocol for Metadata Harvesting (OAI-PMH) [23] and Object Exchange and Reuse (OAI-ORE) standards [24], the open government data (ODG) metadata scheme [25], the Infrastructure for Spatial Information in the European Community (INSPIRE) specification [26] for geospatial data and the Dublin-Core [27] and Machine-Readable Cataloging (MARC) [28] metadata vocabularies. Harvest sources usually provide their

data in one of those standardized formats and different harvester types or plugins are needed for automated metadata import.

GovData supports harvesting of three types of sources. Firstly, a large number of CKAN based platforms are harvested over the belonging CKAN-Representational State Transfer (REST) interfaces by a standard CKAN-Harvester. Geospatial data compliant with the INSPIRE specification is captured via CSW interfaces by a dedicated harvester. The inclusion of such geospatial data requires the CKAN spatial extension. Lastly, a number of data providers come up with own metadata representations provided over REST services that output JavaScript Object Notation (JSON) strings. Those are imported through custom harvester plugins. The harvested metadata is then transformed into OGD metadata scheme, which constitutes the base for capturing metadata within GovData.

The OAI-PMH standard was originally developed in the context of publication retrieval and later on taken up by further institutions, such as the Internet Archive [29], to serve different purposes. It can make use of the Dublin-Core metadata vocabulary for object and document description, but also supports further formats such as MARC. The OAI-ORE standard builds on the OAI-PMH stack and adds the possibility of defining links between different documents and associated alternative formats and version in a so-called resource map, akin to the package description of CKAN. The CKAN harvester implementation for the OAI-PMH [30] allows for selective harvesting of only a subset of datasets based on certain *spec* attributes, which may be comparable to tags. These *specs* have to be set by the data source for each dataset individually. As only a few harvest sources are compliant with this standard – i.e. none of the harvest sources from GovData - and out of those that are compliant not all come with a complete set of specs, a more general way for filtering or selection is desirable.

The HDFS was preferred over other available distributed storage systems, because they are either focused on a specific type of data (i.e. wide-columnar databases such as Apache Cassandra [31] or document stores such as Open Stack Cinder [32]), proprietary (Amazon S3 cloud [33] and the Ceph File System [34]) or appeared to be to a lesser degree in focus of the developer community (OpenStack Swift [35]). Furthermore, data stored on either of these other sub-systems can be readily made available to multitude of Hadoop solutions [36, 37]. More efficient transfer of big data files between repositories can be achieved through usage of the Remote Direct Memory Access protocol over Converged Ethernet (RoCE) [38] that overcomes a weakness of the Transmission Control Protocol (TCP) regarding resource consumption that becomes apparent when transferring big amounts of data.

Streaming data - i.e. coming in real-time from various sensors in the city - can be already readily persisted on the HDFS through tools such as Kafka [39] or integrated later on in-memory at the level of processing engines (such as Spark or Flink) with static data that has been previously imported to the HDFS. Static data so far had to be manually transferred onto the file system, even if it was already cataloged within the corresponding metadata engine. The *HdfsStorer* – proposed by the authors - closed this gap by enabling the automated data import of indexed resources/files in those metadata catalogs to the HDFS. Filtering of the imported data allows for resource economization and benefits the topicality of the imported data as it allows for more frequent harvesting cycles.

The following sections will give an overview about the structure of these two extensions and highlight their possible utilization through both a realization of a proto-use case, as well as a description of a potential real-world application in a Smart City context.

3 Requirements for the HdfsStorer and the Filter Plugin

Various requirements have to be met by the data import procedure in order to be used sensibly in future Smart City as well as for Big Data applications. These are motivated by the nature of the different environments - determined by the diversity of use cases and infrastructure deployments within different Smart Cities - in which systems like CKAN and Hadoop operate:

Req. 1: Smooth Interplay and Integration between the *HdfsStorer* and CKAN. User experience and established operational perception should not be affected by the emerging *HdfsStorer* extension running in the background.

Req. 2: Tracking and Being Up-to-Date with Resource Changes. Given that a new resource was created, or an existing one was changed or deleted, the belonging updates should then be transferred directly to the employed distributed file system (HDFS).

Req. 3: Network Bandwidth Optimization. HDFS entries (i.e. files) should only be updated in case the original resources have changed, such that the network utilization is kept minimal.

Req. 4: Handling Large Data Files. The HDFS extension should be able to import files to the HDFS, irrespective of their size and format. Size limitations - as in the case of the CKAN-internal FileStore - should not be present.

Req. 5: Import of Datasets, which are Already Registered with CKAN. Given that the *HdfsStorer* is activated after a number of datasets have already been cataloged within the belonging CKAN, it should be possible to automatically import the data resources for those already registered datasets.

Req. 6: Filtering. It should be possible to set specific exclusion as well as inclusion filters to restrict the range of datasets imported through the harvesting procedure.

Req. 7: Flexibility of Filters. It should be possible to define a separate set of filters for each harvest source.

Based on these key requirements, the next sections proceed with devising the architecture of the *HdfsStorer* extension and the *Harvester Filter Plugin*, evaluating it based on a prototype, as well presenting the belonging case study and measurements.

4 Internal Architecture of the CKAN HdfsStorer and TheHarvester Filter Plugin

The CKAN platform provides different integration points for extension development. These integration points constitute a set of programming interfaces, which are to be utilized by Python extensions running on top. In general, the programming interfaces are event triggered and are related to the lifecycle of a data resource, encompassing (1) the initial creation of a data resource, (2) its updates, and at the end (3) its deletion/removal. A belonging Python interface is provided, which encompasses hooks to catch and process these events on top of the CKAN harvesting platform. The relevant interface for resource updates is denoted as *IResourceController* and the main hooks of interest are called *after_create*, *after_update* and *before_delete*.

When working with the CKAN metadata catalog, a number of special features related to the metadata and the belonging process of dataset registration should be taken into account. Within CKAN, the concept of a *package* captures the metadata descriptions of a particular dataset along with all its attributes and a list of belonging data resources. The resources stand for single files or data dumps, but can also point to service endpoints on the Internet. Irrespective of the presence of a service, file or a data dump, the resources are referenced by an URL and are accessible over the Internet. In case resources and packages are removed from CKAN, these are not really deleted but rather marked as hidden within the corresponding PostgreSQL [40] database, i.e. a single attribute is changed preventing the datasets from being visible on the belonging portals/platforms and being searchable over the CKAN platform interfaces. Indeed, datasets that are marked for deletion are only visible for administrator users with appropriate authorization. In order to fully remove the corresponding *packages* from the system, a *purging* process needs to be initiated, either through the CKAN portal graphical user interface or by directly accessing the database underneath, in order to delete resource and package metadata entries. It has to be noted that there is no possibility to purge “deleted” metadata over an API. Similarly, there is no possibility to intercept a purge even – i.e. a hook called “after_purge” does not exist. Hence, a design decision was made to completely remove a dataset from the HDFS storage every time the *before_delete* function of the *IResourceController* is triggered. This leads to a construction where each *package* deletion (i.e. marking of a *package* as pending for deletion) leads to removing the belonging data from the HFDS storage.

Datasets come from a range of sources that each have different quality standards and most importantly index very diverse data, from which only a subgroup may be required for a particular use case. Therefore, it should be possible to create separate sets of filters for each data source (see **Requirement (6)**). This ensures a maximum degree of flexibility and does not restrict the range of potential future use cases. In order to harvest a specific data source, a separate harvest job has to be created and data source specific parameters can be set. The CKAN harvesting extension allows for the development of specific harvester plugins. Those plugins are derived from the *Har-vesterBase* class and override or extend certain provided methods, corresponding to different harvest stages. Usually such plugins are developed to enable the harvesting of a different kind of data source. We can use this capability to introduce a means for filtering of imported datasets.

Based on the above generic considerations, the next two sections proceed with introducing the internal components of the emerging *HdfsStorer* extension and *Harvest Filter Plugin*.

4.1 Components and Dynamic Aspects

Figure 2 depicts the general architecture of the extensions including the flow of information (as sequence operations) in an enumerated manner. In this scope, it is clearly described which components are accommodated within the CKAN platform and which are to be viewed as related to the HDFS.

Filtering takes place within the harvesting procedure (more specifically during the *gather stage*) shortly before step (1b). The harvesting procedure is subdivided into three main stages: During the first stage - the *gather stage* - a list of available datasets is retrieved from the harvest source. This is followed by the *fetch stage*, wherein the corresponding metadata packages to these datasets are accumulated to be then - during the *import stage* - integrated into the local catalog (triggering e.g. resource creation or update events). Each of these methods can be altered independently in each harvester plugin implementation. Practically, the *gather and fetch stage* are often regarded as a single process, with all the functionality implemented in the *gather_stage* method. Filtering should take place as early as possible to reduce the load on the entire system. The required attributes for filtering are part of the metadata packages and filtering can thus already take place during the *gather and fetch stage*. Because a multitude of specific harvester plugin implementations override the original *HarvesterBase* class gather method entirely, the authors decided to alter the *gather stage* method of a specific harvester implementation (the general CKAN harvester) instead of altering the method of the base class.

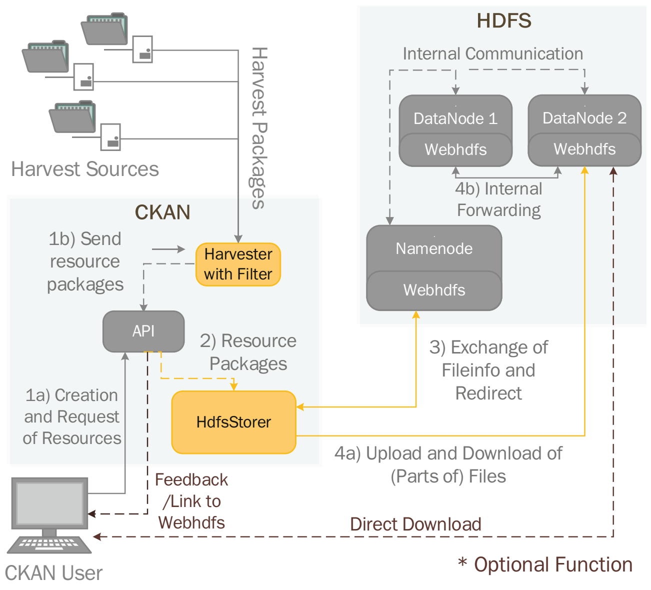

Fig. 2. Extension and Plugin Architecture (highlighted). Figure modified from [1].

Provided that a resource is imported and created or updated with respect to its metadata in CKAN, through either manual interaction or the harvesting procedure – operations (1a) and (1b), all included resource files need to be newly uploaded or refreshed within the HDFS. This flow is given by operations (2), (3) and (4a-b) in Fig. 2.

Furthermore, in case a package or single resources are marked for “deletion” from CKAN, the belonging data files have to be removed again from the HFDS. The identification of the files to delete is performed over the resource IDs, given that for each ID a directory on the HDFS is maintained. Indeed, the CKAN platform generates deletion events and triggers corresponding API calls for the methods implemented by *HdfsStorer* extension based on the *IResourceController* interface. After each CKAN data handling event – such as creation, update and deletion - the corresponding ID and a reference to the latest version of the resource are passed as parameters to the *HdfsStorer*. During the deletion of CKAN-packages, the corresponding ID is given, which leads to a database lookup for obtaining and identifying the resources for removal from the HDFS.

Various communication and data exchanges between the HdfsStorer extension and the HDFS (in the upper right part of Fig. 2) are realized using the WebHDFS protocol [41], which is a REST-API for Hypertext Transfer Protocol/TCP based manipulation of resources kept within the HDFS. The belonging WebHDFS operations are provided in Table 1, including functions such as (1) checking whether a file or directory exist, (2) download and access to resources/files or parts (i.e. chunks) of files/resources, (3) the creation of directories, (4) obtaining a resource/file handle for overwriting an existing resource/file, (5) appending data to an existing resource file and (6) the removal of resource files and directories from the HDFS.

Table 1. List of WebHDFS Operations as described in [1].

Method	Operation	Fields	HTTP return type
GET	liststatus		200 (OK) + JSON
GET	open		200 (OK) + FILE
PUT	mkdirs		200 (OK) + JSON
PUT	createfile	data = ‘ ’	203 (redirect)
POST	append	data; content-type	200 (OK)
DELETE	delete		200 (OK) + JSON

5 CKAN-Based Extension Implementation

The following paragraphs elucidate in detail the *HdfsStorer* extension implementation, starting with the procedure for resource creation and update.

5.1 Creation and Update of Resources

When a new resource is created within CKAN, the *before_create* method is invoked for CKAN extensions that have registered over the corresponding hooks of the *IResourceController* interface. A new resource can be either created manually (e.g. through a Command Line Interface or through the CKAN portal interface) or can be the result of a data harvesting procedure. Thereby, the available implementations of the *before_create* method are invoked with the appropriate data (dictionary) structure filled with information for the new resource as a main parameter. Subsequently, the resource is internally cataloged (i.e. added to the database) and the corresponding *after_create* method is invoked. Similar procedures are followed in the course of resource updates and removals.

Figure 3 details the overall procedure within the *HdfsStorer* extension. The call of the *after_create* method - provided by the *IResourceController* interface and implemented by the extension – results in the creation of an HDFS directory named after the ID of the resource. This directory is located on the resource storage folder, which is prepared in advance on the HDFS and passed to the overall system via a CKAN configuration parameter. Furthermore, the mirroring of the remote resource is prepared by creating an empty file carrying an identical name as the name of the remote resource. In order to enable the subsequent appending of data, a redirect is provided to the HDFS DataNode hosting the newly created empty file. On this basis, the original data resource is read chunk-by-chunk and the chunks are appended to the previously created empty file. This continuous piecewise process of chunking and appending allows for transferring large (larger than the machine’s memory) amounts of data from their original locations to the HDFS – during the experimentations the authors managed to transfer files of roughly 30 GB in size. Hence, it can be claimed that **Requirement (4)** is fulfilled. In case the resource size is larger than the block size specified by HDFS, then the remaining data is automatically redirected to a different DataNode where a new block is created. In all cases, data replication is automatically conducted in the background.

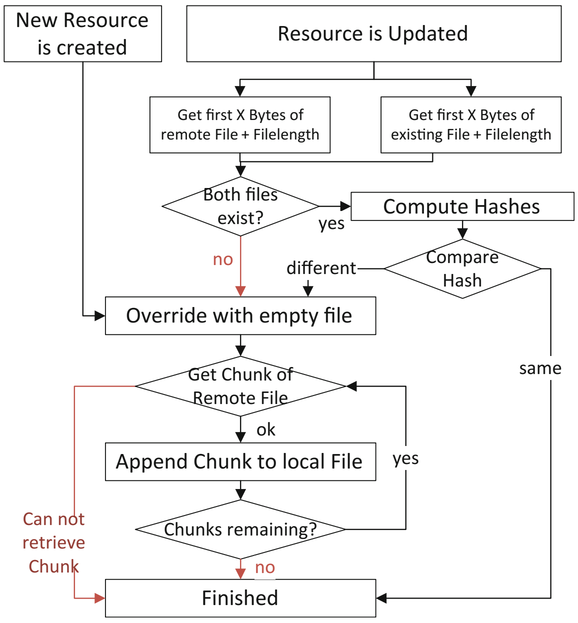

Fig. 3. Resource Addition and Update onto the HDFS as described in [1].

Some very important aspects of the current processes – as described here – are tackled through the usage of hashing concepts. Given that only such resource files should be harvested (i.e. transferred to HDFS) which have changed since the last harvesting processes, a hash check was put in place as visualized on the right in Fig. 3, thereby addressing **Requirement (3)**. A typical approach would be to calculate the checksum (i.e. hash) using the entire file in question. In order to conduct such a checksum computation, the complete files need to be locally in place, i.e. on one single machine. Since it cannot be assumed that data providers will provide appropriate checksums for their data resources, this circumstance would require the files to be additionally downloaded to a single machine in order to compute the hashes and compare the resulting checksums. Unfortunately, this would lead to additional network traffic and contradict **Requirements (3)** and **(4)**, as all files would be downloaded irrespective of their novelty and single machines may not accommodate enough storage space for very big data files. The optimization of this process is a potential topic for further research. However, a first proposition on how to deal with this challenge is given in the following paragraph.

Based on the files' characteristic according to the various contexts of application, two possible approaches can be considered: (1) the comparison procedure is completely omitted and each resource is always harvested and uploaded to the HDFS, or (2) the comparison of the resources is conducted thereby using a partial checksum. Within the (CKAN-)*HdfsStorer* extension the second approach was considered, given that the HDFS and the *HdfsStorer* are meant to be utilized in Big Data generating urban environments. A side remark: the name of a resource (including a checksum based on it) is not considered as an indicator for a resource update given that a simple renaming of a file does not necessarily imply a change in the belonging contents. Another good indicator for changes in large resources is provided by changes in the file size for a resource. Hence, the file sizes can serve as one constituent for the hash key. However, there are also cases where the size doesn't hold as a good indicator. For example, log files – constituting a big share of the resources to be processed by big data engines – are usually changed on a *log-rotate* principle, which means that the file size remains constant, whilst the oldest data is removed and the newest is appended. By selecting a number of bytes at either the beginning or the end of a file (or both), and including these bytes as further parameters for the hash key, changes to such files would be reliably detected and taken into account. If the number of bytes is of a large enough amount, then for smaller files (e.g. configuration files or images) the entire checksum is computed within the presented approach. The downside is that resources which do not differ in file size after update and are larger than the defined chunk size, with static header (beginning of the file) and footer (last bytes of the file) will be omitted as resources that have changed and need to be harvested anew. Such files are not very suitable for distributed computing and processing, given that they are normally hard to split. In addition, the (rare) case of data with only minor differences in a large splittable, identically sized resource will be omitted. Given the large size of the expected datasets, individual items should be only of minor importance to the final result (after processing the entire dataset) and the more single items are updated, the higher is the probability that a difference in file size can be detected. Hence, the above described drawback can be considered acceptable for the majority of scenarios.

5.2 Parallel Upload of Data

In order to avoid negative influence on the performance of the CKAN system - thus addressing **Requirement (1)** - the design decision was made to avoid parallel running data uploads to the HDFS, since - based on the specific setting - many parallel data upload processes may use up the entire network bandwidth to the server. This should be considered when setting the repetition period for data/metadata harvesting, because a single harvesting job might be significantly slowed down and might result in an ever-increasing queue of harvesting jobs leading to outdated and invalid data. Provisioning of a second CKAN server only for the purpose of harvesting is a possible solution, thereby establishing a periodical synchronization of its database with the main CKAN server. This is expected to enable parallel data upload without influencing **Requirement (1)** in a negative way.

5.3 Deletion of Resources

The process of resource and package deletion is visualized in Fig. 4. The removal of a single resource from CKAN leads to the call of the *before_delete* function of the *IResourceController* interface, which is implemented and utilized by the *HdfsStorer* extension. As previously explained, package removal in CKAN does not lead to the deletion of the respective resources, and hence does not result in the call of any additional functions from the *IResourceController* interface. Correspondingly, package removal has to be intercepted and handled by the (CKAN-) *HdfsStorer* extension, which is done by implementing the *after_delete* function from the *IPackageController* interface (depicted in Fig. 4). It is required to retrieve all corresponding resource IDs in both of these two functions and initiate the deletion of the specific directories on the HDFS and their contents by utilizing the belonging WebHDFS API. This fulfills **Requirement (2)**.

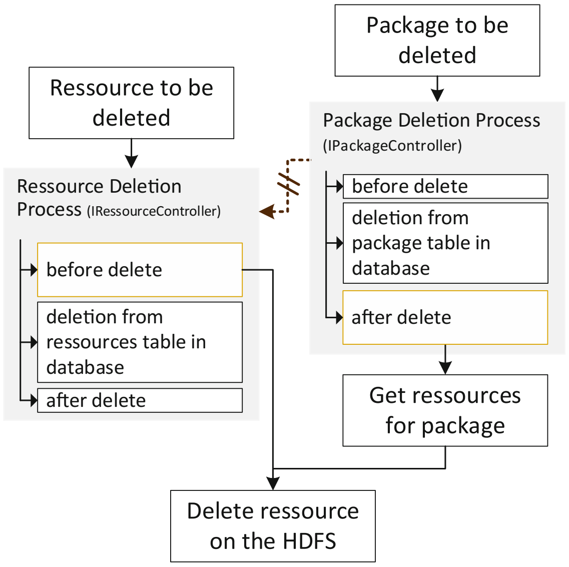
Package to be deleted
Ressource to be deleted
Package Deletion Process (IPackageController)
before delete
deletion from package table in database
after delete
Ressource Deletion Process (IRessourceController)
before delete
deletion from ressources table in database
after delete
Get ressources for package
Delete ressource on the HDFS

Fig. 4. Deletion of a Package or a Resource. Package deletion does not usually result in resource deletion and hence had to be explicitly implemented.

5.4 Backwards Compatibility

Given the need to address **Requirement (5)** - referring to handling existing CKAN entries - another module was designed and implemented. This module is mainly responsible for reading the internal CKAN database and the transfer of the referenced files to the HDFS, whilst ensuring in parallel the consistency of the data. The belonging process flow is demonstrated in Fig. 5. The information for each resource registered

within CKAN is obtained from the PostgreSQL-database in the backend and subsequently uploaded to the HDFS DataNodes. Packages and resources which have been marked as deleted (i.e. they wait to be purged) are excluded from uploading to the HDFS.

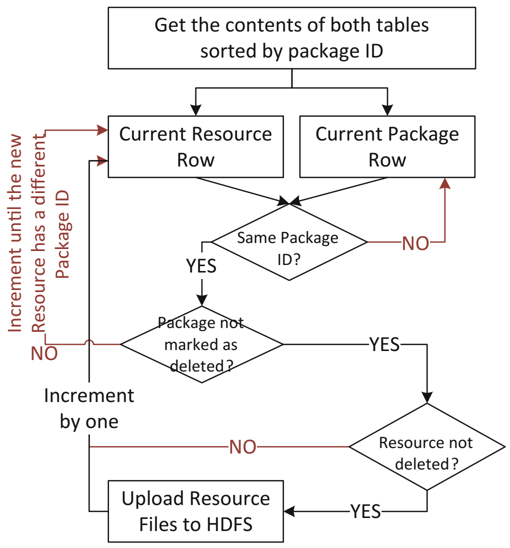
Get the contents of both tables sorted by package ID
Current Resource Row
Current Package Row
Same Package ID?
NO
YES
Increment until the new Resource has a different Package ID
NO
Package not marked as deleted?
YES
Increment by one
NO
Resource not deleted?
YES
Upload Resource Files to HDFS

Fig. 5. Resource Import from the CKAN-internal Database as described in [1].

6 Implementation of the Harvester Filter Plugin

A way of filtering datasets has been implemented through extending one of the available harvest plugins of the CKAN harvest extension. In order to satisfy **Requirements (6)** and **(7)**, a means for the definition of the set of filters for specific harvest sources and the actual application of the filters - within the harvesting procedure – were realized.

6.1 Initialization of Filters

In order to harvest a specific data source, a *harvest job* has to be defined in the CKAN backend by a user with the appropriate privileges. This definition includes the URL under which the harvest source is reachable, the harvester plugin to be used (which depends on the type of the data source), the periodicity of harvest iterations and an optional dictionary including further configuration settings. We can pass therein the desired filter settings that are then available to the harvester plugin. In order to ensure a required degree of quality, these settings are validated upon creation of the harvesting job and an error can be thrown in case faulty or conflicting parameters were provided.

Mainly two types of filters can be used: (1) *Inclusion filters* define certain criteria that have to be met by an item, in order to successfully pass the selection process. In contrast to that stand (2) *exclusion filters* that pass all items that do not meet the filter criteria. The filter type can be set through the *filter* attribute of the harvest job configuration dictionary.

Possible filtering criteria include restrictions on file formats, file size or string matching within the file name. Similar to the filter type, these criteria are also defined in the harvest job configuration dictionary. For instance, to harvest only images exceeding a file size of 3 MB, the following configuration would have to be used: *{filter: 'inclusion', size: ' > 3mb', file_type: 'jpg,png,gif'}*. Disjunction of multiple attributes for inclusion filtering can be achieved by the creation of multiple harvest jobs working in a pipeline, each of which would be configured with one of the disjunct inclusion criteria as filtering settings.

6.2 Application of Filters

During the *gather stage*, the configuration dictionary that has been passed during the creation of the harvest job is parsed and the different filtering parameters extracted and compared against the "to-be-gathered" datasets. Resources and packages that do not belong to the group specified by the filters are excluded from further harvesting stages and thus are not imported. This process is illustrated in Fig. 6.

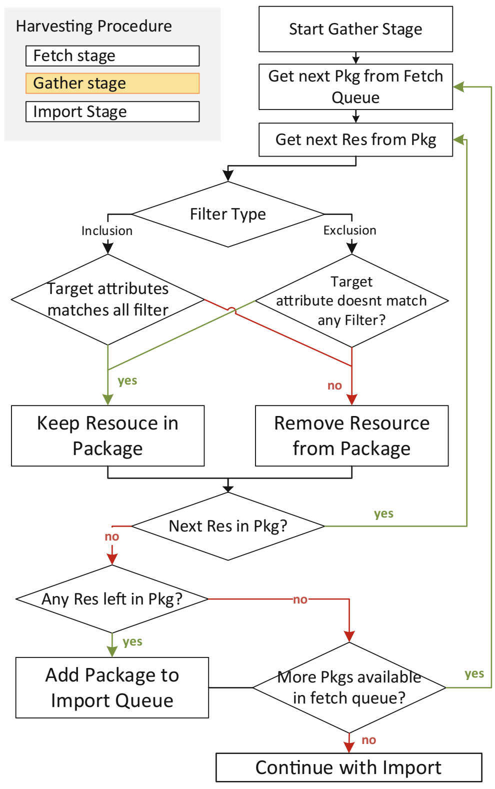

Fig. 6. The Filtering Procedure implemented in the Harvester Plugin. Res: Resources, Pkg: Package.

In order to realize the logic from the previous paragraph, the list of available packages is iterated through. For each of the resources, it is then ascertained that they either do - in case of inclusion filters - or do not - in case of exclusion filters - match the defined filter criteria. Shouldn't this be the case, the resource is removed from the current package. The package – given that some resources remain contained within – is then put onto a stack that is later handed over to the *import stage* of the *HdfsStorer*. Empty packages are discarded.

7 Proof of Concept

In order to demonstrate the feasibility and the processes around the *HdfsStorer* extension, a scenario was worked out making use of an established technique from the field of Machine Learning. Thereby, the techniques are employed on top of two different processing engines for classification of multi-dimensional data. The utilization of two different processing engines demonstrates the variety of further utilization the data can be used for (e.g. based on the free choice of processing engines), once it has been imported to the HDFS. In this scope, the *HdfsStorer* is the key element for such analyses by enabling large scale Big Data and Open Data to be integrated and efficiently re-used in the scope of urban data platforms. Creating an application that has as input datasets, which are linked in the belonging CKAN-registry, essentially consists of three major steps described in the following.

7.1 Import of the Dataset(S) to the HDFS by Means of the Hdfsstorer

The utilized dataset [42] contains descriptions of phoneme properties (such as place of articulation) and the belonging classification (i.e. classification tags) of those into phoneme classes. It was necessary to split the data into training and a test subset for evaluation purposes. Both subsets were stored in different files. Once the files are cataloged within CKAN, either through manual addition or file type-specific harvesting of a data source, they are automatically imported to the HDFS by the *HdfsStorer*.

7.2 Selection of the Appropriate Processing Engine and Program Logics

Both Standard Hadoop Mapreduce and Its in-Memory Counterpart Spark Were Used to Separately Train an Artificial Neural Networks (ANN) on The Training Subset. Thereby, a Standard Backpropagation Algorithm Was Used to Perform The Training Procedure. The Details of The Employed Back Propagation Algorithm Can Be Found in [43]. The Trained Anns Were Subsequently Used as Data Point Classifiers for The Evaluation Set.

7.3 Job Execution and Result Retrieval

The executed jobs were triggered over the command line. The belonging data classification results - based on the *HdfsStorer* data imports - can be in turn retrieved from the HDFS filesystem. As the goal of this work is not given by the evaluation of the classification quality of different ANN implementations, only the training step is considered in the following text, with the goal to look into the performance of two key Big data technologies (Hadoop MapReduce and Spark) on top of the *HdfsStorer* results, i.e. the imported datasets.

The performed execution time measurements are provided in Fig. 7. The execution time was obtained as the time difference between job application submission (be it a Spark or a Hadoop job) and job termination. The Spark and Hadoop execution times in Fig. 7 show that for the dataset in question - which was imported over CKAN and the *HdfsStorer* extension - not only Spark execution times are by far shorter (based on experiments with various number of ANN iterations), but also rise more slowly than Hadoop execution times. This can be attributed to the much lower Spark overhead for each iteration.

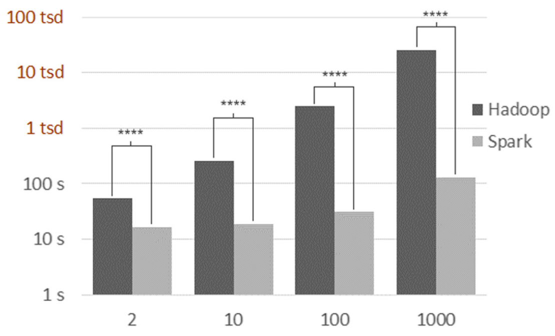

Fig. 7. Execution times of Spark and Hadoop compared. Stars indicate the level of significance (p < 0.0001) as described in [1].

In addition, Fig. 8 describes the average memory utilization of both Hadoop and Spark during idle time and job execution with 100 iterations on top of the open dataset that has been imported to HDFS over the *HdfsStorer* extension and CKAN. The vertical axis displays the memory usage in Mega-Bytes. Because of the comparably small size of the dataset, expected characteristics (e.g. the much higher expected memory utilization of Spark) in the absolute statistics have not been evident. The difference between idle time and work intensive job execution periods is greater for both slaves in the Spark deployment than that of the slaves in the Hadoop deployment. This is a clear sign for the stronger memory dependence of Spark during data processing.

The above evaluation gives an idea of how important it is to choose the right processing engine for the overall efficiency of a Big Data driven Smart City service. The usage of the HDFS thereby facilitates the free choice of the processing engine, given that HDFS is a common platform for distributed processing in modern data centres. The processing engine evaluation can be done on a broader basis or can be targeting specific datasets within particular Smart City scenarios. Overall, this constellation is made possible by the *HdfsStorer* extension, which has been specified and prototyped in the current work.

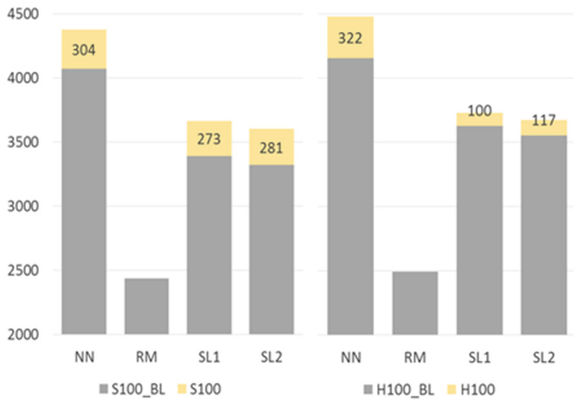

Fig. 8. Comparison of ANN Training Memory Usage of Hadoop (H) and Spark (S) Memory Usage between idle (“Baseline”, BL) and work intensive Periods for 100 Training Iterations as described in [1]. NN: NameNode/MasterServer, RM: ResourceManager, SL1/2: Slaves.

8 A Smart City Scenario Using Hadoop and CKAN

Having shown the general architecture and process flows of the extension, and having evaluated its principal applicability, a natural next step is to apply the *HdfsStorer* for the purpose of realizing a more complex scenario. We target a use case relating to the public transport system of a forthcoming Smart City, which requires to be streamlined and optimized. This includes various aspects such as schedule improvements based on the dynamic identification of peak traffic hours combined with delay and occupancy prediction integrated with trip planning [1]. The ultimate goal is to provide a better travel experience to passengers.

The data needed for the above scenario is given by two different types: static and real-time. The transport schedule and history of occupancy and punctuality statistics together with the belonging history of road and weather condition records are distributed over different data stores as static data. Correspondingly, these datasets would be indexed in a CKAN-catalog. Periodical harvesting based on CKAN mechanisms would guarantee that this catalog is up-to-date and the *Harvester Filter Plugin* would ensure that only relevant data is indexed by the local catalog and consequently transferred to the Hadoop cluster. Data regarding the current weather, road and traffic conditions as well as the amount of passengers — e.g. measured by sensors inside the transport vehicle - are provided as streaming data. The interplay between these two types of data within Hadoop is sketched in Fig. 9. The *HdfsStorer* extension takes the role of the key component responsible for importing static data to the HDFS according to metadata provided within the CKAN-catalog (e.g. URL of the original dataset). Consequently, various processing engines - that can make use of the files stored on the HDFS and of data streams provided through message brokers - integrate the data and enable the envisioned correlation and data integration towards sophisticated urban services.

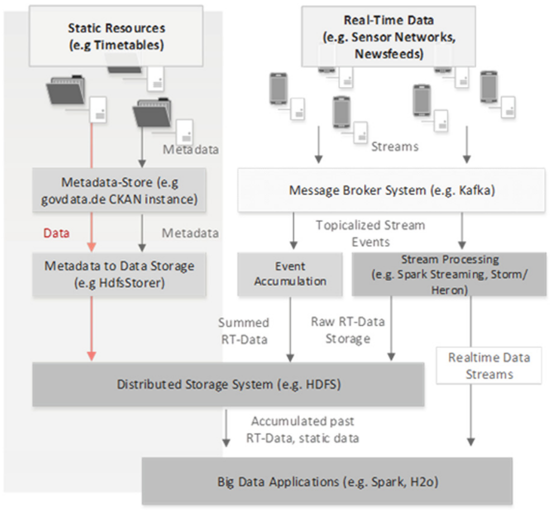
Static Resources (e.g Timetables)
Real-Time Data (e.g. Sensor Networks, Newsfeeds)
Metadata
Streams
Metadata-Store (e.g govdata.de CKAN instance)
Message Broker System (e.g. Kafka)
Data
Metadata
Topicalized Stream Events
Metadata to Data Storage (e.g HdfsStorer)
Event Accumulation
Stream Processing (e.g. Spark Streaming, Storm/ Heron)
Summed RT-Data
Raw RT-Data Storage
Realtime Data Streams
Distributed Storage System (e.g. HDFS)
Accumulated past RT-Data, static data
Big Data Applications (e.g. Spark, H2o)

Fig. 9. Integration of both static and streaming Data coming from different Sources on the Level of distributed Storage and Processing Engines, as described in [1].

A realization of a similar process can be found on the H2O.ai github-page [44]. Thereby, Spark is utilized in combination with an H2O extension (=Sparkling water) to create a prediction system for flight delays based on historical data and current weather information. As compared to our use case, the dataset in question is presumed to be in place on the HDFS right from the beginning, e.g. through a manual upload. The current work enhances such traditional solutions by a more convenient way for data import to HDFS, thereby making use of a widely applied data cataloging system (i.e. CKAN) for Open Data in the Smart City context.

9 Discussion

In this paper, the authors described the implementation of two extensions to the CKAN metadata storage system. The first allowed for the automated transfer of the actual data files that are described by the cataloged metadata entries to a designated data storage and processing hub, exemplified by the HDFS. The second extension allows to restrict the range of datasets imported to CKAN through its harvesting procedure. In combination, these two extensions enable the rapid import of (only) relevant data, which has been referenced in various metadata catalogs, onto a scalable distributed file system. This imported data can then be directly used as the basis for a diverse range of Smart City as well as general data science applications.

The creation of applications on the basis of Open Data and the possibly resulting informational and economic gain is one of the main motivations for making previously closed data available to the public [45]. However, making data available to the public remains a heavy burden for the required institutions. Normally this comes at the cost of having to invest additional labour and exposing their inner workings to the scrutinizing eye of the general public. This results in the current situation, where even though the European Commission has issued corresponding regulations years ago (European PSI directive 2003 [46]), the amount of available Open Data across Europe remains far behind the identified potentials. A further factor that may contribute to the currently apparent shortcomings of Open Data, in both the private as well as the governmental sector, could be the lack of quality assurance mechanisms. To ensure the quality of the publicized data, providers are required to go through a process of continuously updating these datasets and belonging metadata. Currently, they are not obliged to this commitment. This has the consequence that, after an initial dataset is published, it very often is not continuously updated since it requires an additional effort from the data provider's side. Potential users thus may find incomplete, outdated or erroneous data and refrain from using it altogether for the development of new applications or as a reliable source of information. This interactive but manual procedure could be complemented by automated periodic quality checks on the basis of either actual data or metadata, addressing issues such as completeness, accuracy, timeliness, as well as privacy concerns and other quality parameters. The frequency of these checks could be coupled to the harvesting frequency or run independently at regular intervals, possibly defined separately for each type of dataset or data source, or estimated by analysis of the time course of previous dataset and version changes. The described data import extension could additionally empower such automated quality checks, as these could make use of the already imported data on the distributed file system without the additional overhead of having to download (parts of) the data files a second time and furthermore run in parallel by virtue of the distributed nature of the HDFS and associated processing engines.

Also, other fields outside the scope of Smart Cities may benefit from the outlined extensions, especially those that require the accumulation of big corpora of textual or other data files. These include computational linguistics and other data/machine learning-heavy disciplines. The reduction or corpora reuse - through enabling also small research and development groups to create their own corpora – and the introduction of more diverse sources may enable better generalization of applications such as speech generation and translation engines [46]. Dedicated corpora that can be accumulated from multiple data sources through more filtered selection (i.e. by language, text type or image/video resolution) of datasets could furthermore enable more detailed insights - i.e. into situative or group-specific media and language usage or geographically and time-restricted phenomena and interactions - as well as the creation of more dedicated applications. Certain types of advanced filtering may imply the necessity for downloading parts of the actual data files and possibly the need for conducting expensive (non-distributed) computations. Thus, it may be more reasonable in some use cases to carry out such second step filtering after data import.

The insights gained during the development of both extensions, and the above considerations regarding the success of Open Data initiatives and application development, will be taken up as input for emerging and running national and international projects on *Urban Data Platforms*.

10 Summary and Conclusions

Smart City projects are currently on the rise, as major and minor municipalities strive to be on the forefront of state-of-the-art technologies and smart solutions in areas such as mobility, energy and ICT. Correspondingly, ample funding is available. In parallel, initial expectations on the impact of Open Data remained unmet and have been dampened. The current work described a possibility of re-establishing and strengthening the link between these two fields – Smart Cities and Open Data - by means of integrating metadata hubs with data processing engines. Thereby each aspect represents a key component in its corresponding field. The authors hope that this link may have some contribution in transferring the momentum that Smart City concepts currently enjoy also to the field of Open Data and furthermore also highlight how Smart Cities may benefit from the continued publication of freely useable data.

References

1. Scholz, R., Tcholtchev, N.,Lämmel, P., Schieferdecker, I.: A CKAN plugin for data harvesting to the Hadoop distributed file system. In: 7th International Conference on Cloud Computing and Services Science (CLOSER) (2017). http://dx.doi.org/10.5220/0006230200470056
2. CKAN Association: CKAN Overview. http://ckan.org
3. Shvachko, K., Kuang, H., Radia, S., Chansler, R.: The Hadoop distributed file system. In: 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies, MSST2010 (2010). http://dx.doi.org/10.1109/MSST.2010.5496972
4. Helene, M.: GovData - Das Datenportal für Deutschland. In: Hill, H., Martini, M., Wagner, E. (eds.) Transparenz, Partizipation, Kollaboration: Die digitale Verwaltung neu denken, pp. 109–116. Nomos Verlagsgesellschaft mbH & Co. KG, Baden-Baden (2014)
5. Bundesministerium des Innern: Nationaler Aktionsplan der Bundesregierung zur Umsetzung der Open-Data-Charta der G8. https://www.bmi.bund.de/SharedDocs/Downloads/DE/Broschueren/2014/aktionsplan-open-data.pdf (2014)
6. Mercader, A., et al.: ckanext-harvest - remote harvesting extension (2012). https://github.com/ckan/ckanext-harvest
7. The Apache Software Foundation: Hadoop Project Webpage. http://hadoop.apache.org/
8. Hunt, P., Konar, M., Junqueira, F.P., Reed, B.: ZooKeeper: wait-free Coordination for Internet-scale systems. In: USENIX Annual Technical Conference, Boston, MA, USA, p. 9 (2010)
9. Dittrich, J., Quiané-Ruiz, J.-A.: Efficient big data processing in Hadoop MapReduce. Proc. VLDB Endow. **5**, 2014–2015 (2012). https://doi.org/10.14778/2367502.2367562
10. The Apache Software Foundation: Apache Flink: Scalable Stream and Batch Data Processing. https://flink.apache.org/
11. The Apache Software Foundation: Apache Spark - Lightning-Fast Cluster Computing. https://spark.apache.org/
12. Iqbal, M., Soomro, T.: Big Data Analysis: Apache Storm Perspective (2015). https://doi.org/10.14445/22312803/ijctt-v19p103
13. Avery, C.: Giraph: large-scale graph processing infrastructure on hadoop. Proc. Hadoop Summit. St. Cl. **11**, 5–9 (2011)

14. Thusoo, A., Sarma, J.S., Jain, N., Shao, Z., Chakka, P., Anthony, S., Liu, H., Wyckoff, P., Murthy, R.: Hive: a warehousing solution over a Map-Reduce framework. Proc. VLDB Endow. **2**, 1626–1629 (2009). https://doi.org/10.14778/1687553.1687609
15. Bittorf, M., Bobrovytsky, T., Erickson, C.C.A.C.J., Hecht, M.G.D., Kuff, M.J.I.J.L., Leblang, D.K.A., Robinson, N.L.I.P.H., Rus, D.R.S., Wanderman, J.R.D.T.S., Yoder, M.M.: Impala: A modern, open-source SQL engine for Hadoop. In: Proceedings of the 7th Biennial Conference on Innovative Data Systems Research (2015)
16. Vora, M.N.: Hadoop-HBase for large-scale data (2011). http://dx.doi.org/10.1109/ICCSNT.2011.6182030
17. National Strategy Office of Information and Communications Technology in Cabinet Secretariat: data.go.jp. http://www.data.go.jp/?lang=english
18. Matheus, R., Vaz, J., Maia Ribeiro, M.: Open Government Data and the Data Usage for Improvement of Public Services in the Rio de Janeiro City (2014). http://dx.doi.org/10.1145/2691195.2691240
19. Socrata: Socrata - The Data Platform for 21st Century Digital Government. https://www.socrata.com/
20. Knoema: knoema.com Webpage. https://knoema.com/
21. Senatsverwaltung für Wirtschaft, E. und B.: Offene Daten Berlin. https://daten.berlin.de/
22. European Commission Directorate-General Communication: European Data Portal. https://www.europeandataportal.eu/en/
23. Lagoze, C., Van de Sompel, H., Nelson, M., Warner, S.: Open Archives Initiative Protocol for Metadata Harvesting (2015)
24. Open Archives Initiative: Object Reuse and Exchange Specifications and User Guides. https://www.openarchives.org/ore/1.0/toc
25. Marienfeld, F.: Open Government Data (OGD) - Die Metadaten-Struktur für Open Government Data in Deutschland. http://open-data.fokus.fraunhofer.de/die-metadaten-struktur-fur-open-government-data-in-deutschland/
26. Bartha, G., Kocsis, S.: Standardization of geographic data: the european inspire directive. Eur. J. Geogr. **2**, 79–89 (2011)
27. Weibel, S., Kunze, J., Lagoze, C., Wolf, M.: Dublin core metadata for resource discovery (1998). https://doi.org/10.17487/rfc2413
28. Coyle, K.: MARC21 as data: a start. Code4Lib J. **14**, 1–10 (2011)
29. Liu, Xiaoming, Balakireva, Lyudmila, Hochstenbach, Patrick, Van de Sompel, Herbert: File-based storage of digital objects and constituent datastreams: XMLtapes and Internet Archive ARC files. In: Rauber, Andreas, Christodoulakis, Stavros, Tjoa, A.Min (eds.) ECDL 2005. LNCS, vol. 3652, pp. 254–265. Springer, Heidelberg (2005). https://doi.org/10.1007/11551362_23
30. Open science and research initiative: OAI-PMH harvester for CKAN. https://github.com/kata-csc/ckanext-oaipmh
31. Lakshman, A., Malik, P.: Cassandra: a decentralized structured storage system. SIGOPS Oper. Syst. Rev. **44**, 35–40 (2010). https://doi.org/10.1145/1773912.1773922
32. McGninnis, S., et al.: OpenStack Block Storage Cinder. https://wiki.openstack.org/wiki/Cinder
33. Amazon.com, In.: Amazon Web Services S3 - Simple Cloud Storage Service
34. Watkins, N., Sevilla, M., Jimenez, I., Maltzahn, C.: Ceph: An Open-Source Software-Defined Storage Stack
35. Dickinson, J., et al.: OpenStack Object Storage. https://wiki.openstack.org/wiki/Swift
36. Nóbrega, T.: OpenStack Sahara. https://wiki.openstack.org/wiki/Sahara
37. Red Hat Inc.: Using Hadoop with CephFS. http://docs.ceph.com/docs/master/cephfs/hadoop/

38. Tierney, B., Kissel, E., Swany, M., Pouyoul, E.: Efficient data transfer protocols for big data (2012). http://dx.doi.org/10.1109/eScience.2012.6404462
39. Kreps, J., Narkhede, N., Rao, J.: Kafka: a distributed messaging system for log processing. In: Proceedings of the NetDB, pp. 1–7 (2011)
40. Momjian, B.: PostgreSQL: Introduction and Concepts. Addison-Wesley, New York (2001)
41. The Apache Software Foundation: WebHDFS REST API. http://hadoop.apache.org/docs/%0Ar1.0.4/webhdfs.html
42. Alinat, P., Pierrel, J.M.: Esprit II project 5516 Roars: robust analytic speech recognition system (1993)
43. Liu, Z., Li, H., Miao, G.: MapReduce-based Backpropagation Neural Network over large scale mobile data (2010). http://dx.doi.org/10.1109/ICNC.2010.5584323
44. H2O.ai: AirlinesWithWeatherDemo. https://github.com/h2oai/sparkling-water/tree/master/examples/
45. Klessmann, J., Denker, P., Schieferdecker, I., Schulz, S.: Open government data Deutschland. Eine Studie zu Open Government in Deutschland im Auftrag des Bundesministerium des Innern. Deutschland <Bundesrepublik>/Bundesministerium (2012)
46. Wuebker, J., Ney, H., Zens, R.: Fast and scalable decoding with language model look-ahead for phrase-based statistical machine translation. In: Proceedings of the 50th Annual Meeting of the Association for Computational Linguistics: Short Papers, vol. 2, pp. 28–32. Association for Computational Linguistics, Stroudsburg (2012)