SEDAR: A Semantic Data Reservoir for Heterogeneous Datasets with Versioning and Semantic Linking

Sayed Hoseini
Hochschule Niederrhein
sayed.hoseini@hs-niederrhein.de

Marcel Thiel
Hochschule Niederrhein
marcel.thiel@stud.hn.de

Alexander Martin
Hochschule Niederrhein
alex@kugma.de

Christoph Quix
Fraunhofer FIT & Hochschule Niederrhein
christoph.quix@fit.fraunhofer.de

ABSTRACT

During recent years, data lakes emerged as a way to manage large amounts of heterogeneous data for modern data analytics. Some data lake frameworks have been proposed, but requirements like data versioning, linking of datasets, semantic metadata management, or schema evolution are not supported in one system. We demonstrate *SEDAR*, a comprehensive data lake system that includes support for data ingestion, storage, processing, and governance. The generic ingestion interface can deal with any external data source ranging from files to databases and streams. The ingestion provides change data capture as well as data versioning and automatic metadata extraction. To prevent the system from turning into an inoperable data swamp, metadata management is enabled at various levels. The data catalog based on a generic metadata model can capture metadata for many use cases, e.g., provenance, versioning, lineage, semantic annotations, and links to knowledge graphs. The demo will showcase how the system allows for various ingestion scenarios, metadata enrichment, data source linking, tagging, profiling and data processing with either a graphical user interface for transformation workflows or computational notebooks.

1 INTRODUCTION

Data lakes have been addressed intensively in research and practice in the recent years as they provide the required data for data science and machine learning projects. Data lakes are scalable schema-less repositories to ingest raw data in its original format from heterogeneous data sources; thus, only a minimal effort is required for ingesting data into a data lake which makes a data lake an efficient tool to collect, store, link, and transform datasets. While first implementations for data lakes aimed at processing 'big data' efficiently using distributed, scalable systems like *Hadoop*, the need for proper management of metadata and data quality in data lakes has also been recognized [4, 5, 7].

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

This demand is endorsed by a current smart factory use case [8] at our institute, where different experimental data needs to be captured and catalogued, in particular streams, APIs, data bases, images and model checkpoints. As these external sources change in time, the systems needs to track these continously and efficiently. This heterogeneity impedes the adherence to the FAIR principles.

A basic function for metadata management in data lakes is metadata extraction during ingestion [12], but also more advanced functionality (such as semantic enrichment, data indexing, link generation, data polymorphism, data version, and usage tracking [13]) or data quality features [16] have been defined as requirements. These requirements can only be met partially by existing data lake systems [13].

In this demo, we present *SEDAR* as a comprehensive data lake system that addresses the aforementioned requirements. The system addresses especially the research question on how we can track the evolution of a dataset within the data lake system. Our system is able to track different versions of the dataset, to detect and highlight the changes between two versions, but it is also able to establish links between semantically related datasets. The latter aspect is supported by a semantic metadata management, i.e., metadata can be linked to knowledge graphs (ontologies) that provide more semantic meaning to the data in the lake. The system is built on existing open-source technologies in the area of big data management, but extends these components by the following more sophisticated functionalities:

- A semantic layer is added to the metadata which enables semantic annotations of all metadata based on the Linked Data principles [3].
- Data is indexed to enable efficient full-text search across all datasets.
- Usage of datasets in user queries is tracked to enrich existing metadata and to link datasets.
- A dataset can be stored in different physical storages, thereby supporting data polymorphism.
- Provenance and different versions of datasets are supported by using an efficient delta store [1].
- Data governance is supported by an extensible data quality monitoring and profiling tool [14].

For SEDAR, we have designed a modular and flexible architecture based on microservices which can be extended easily with additional functionality, making the system suitable

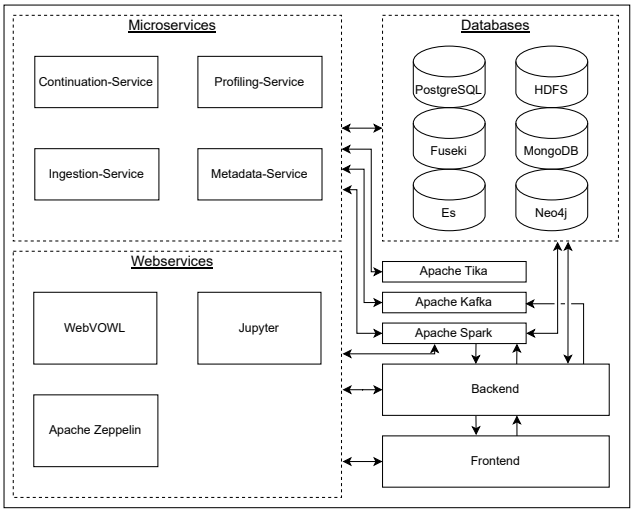

Figure 1: The *SEDAR* system architecture

as a workbench for further research in data lakes. The key components of the system are the generic ingestion interface and the data catalog, which are presented in section 3 and 4, respectively. Before, we will give an overview of the system architecture in section 2.

2 SYSTEM ARCHITECTURE

SEDAR is a web application with a modern, responsive user interface and several data management technologies in the backend. The functionality of the backend system is exposed by a REST API. The system architecture is illustrated in Figure 1 and is based on a four layer architecture: ingestion, storage, transformation and interaction [9]. The *ingestion layer* with its generic interface is described in detail in 3.

The *storage layer* is composed by different storage systems, which can be extended to support efficient storage for specific data structures. The idea is that queries on specific data structures like tables, JSON documents, graphs are best supported in systems that are optimized for queries on these structures (i.e., relational, document-oriented or graph database systems). In the current version, we utilize a *PostgreSQL* database for structured relational data, *MongoDB* stores nested documents (e.g., JSON and XML), and *Neo4j* for graphs. The storage systems are glued together by the query engine of *Apache Spark*.

Different requirements emerge during the implementation of a generic metadata management system. By utilizing both *MongoDB* and *Neo4j* simultaneously, we aim at leveraging the best of each technology. *Neo4j* has shown to be more flexible for the integration of semantic technologies due to its native graph-based data model, which allows graph-based algorithms. *Neo4j* supports immediate derivation and visualization of schemas in its native form, and further information about relationships between datasets can be represented as edges in the data model. The query engine of *MongoDB* is useful and practicable for other use cases. Knowledge graphs are persisted in a *Fuseki* database which supports typical Semantic Web formats like RDF/XML or Turtle and provides a SPARQL Endpoint. Finally, *Hadoop* with its file system HDFS is used to store files with any arbitrary structure.

The *transformation layer* allows data access to perform various transformation operations, e.g., cleansing operations, format transformations, or joins. Here, we rely heavily on *Apache Spark*, a unified analytics engine for large-scale data processing with implicit data parallelism and fault tolerance. The main reason for choosing *Apache Spark* is the query engine that supports SQL-like queries across heterogeneous data stores. Furthermore, *Spark* provides *DataFrames* a uniform data representation for relational and nested data structures. *DataFrames* can be used to transform, to integrate, or to analyze data using an easy-to-use API or the declarative query language *SparkSQL*. The *Spark* Cluster is controlled by the main backend written in the *Python* programming language, which receives requests from the interaction layer and manages all metadata and data processing task. Metadata is extracted at various stages: we use *Spark*, which provides basic information about schemas and data types, *Apache Tika* to extract further information like language identification or MIME-types and *Amazon's Deequ* [14] for data profiling and data quality indicators in large datasets.

The *interaction layer* is composed of a User Interface (UI) written in *JavaScript* and *React*. In addition to the data catalog, described in detail in the section 4, it provides a UI to define complex data integration and transformation workflows (see figure 3). Furthermore, we have integrated some web-based applications as microservices for specific tasks: *WebVOWL* [11] for the visualization of ontologies and *Apache Zeppelin* as well as *Jupyter* for data analytics with computational notebooks.

We introduce the concept of workspaces to the system which reaches deep into the physical layer and generally enables data security and supports data governance. Every user is assigned to a workspace and may add other users, but generally cannot access (meta) data from other workspace unless it is made publicly available. Various query options are provided in the interaction layer. We utilize *Elasticsearch* to index source data, *SparkSQL* for custom queries upon datasets and SPARQL for knowledge graphs.

For the actual deployment of the application, all of the aforementioned individual components and services will be run as microservices to enable easy deployment, failure resilience and robust software packaging.

3 GENERIC INGESTION INTERFACE WITH VERSIONING

For the development of the ingestion layer, we defined several requirements to ensure the generality of the implementation. First, the ingestion interface must be independent of any formats and data sources. As *Apache Spark* does not support every possible data format/data source directly, it needs to be extended to allow custom configuration to process data. We have implemented a plugin mechanism to have a flexible and

extensible option to support custom data sources and formats. The demo shows, that only a few lines of *Python* code are required to implement a plugin. For example, accessing data from a JSON-based web service is not supported, but can be configured via a plugin. The ingestion must allow this configuration at run time, i.e., without changes to the source code.

Moreover, it should be possible to ingest external sources continuously. These can be data streams or timed batch executions, but the system must ensure that parallel actions do not interfere. Further, the ingestion must allow data versioning, i.e., the system must provide a functionality to perform change data capture between two version of a dataset [10]. By doing so, the system can incrementally process the updates instead of loading the dataset as whole. On the other hand, the data versioning mechanism enables access to previous states of a dataset. The information about ingested datasets, their versions, updates, etc. are extracted automatically (as far as possible) and store in the data catalog.

The basic architecture of the ingestion layer is split into four services: ingestion, API, continuation, and message service. We utilize *Apache Kafka*, an open-source distributed event streaming platform and message-broker, to register and queue ingestion events and distribute messages between services. At the same time, it is utilized to ingest data streams. The API service provides the REST endpoints that give access to the various functions of the ingestion interface.

The ingestion service has the task to create the *Spark* job for the execution of an ingestion which means starting, monitoring and adjusting the status of the event. This includes defining the process for loading data into a *DataFrame*, change data capture, storing the data as well as coordinating the parallel executions. As mentioned above, we use a plugin mechanism to support any type of data source. The plugins are triggered at two different points during the ingestion process. The first point is, as described, the loading of data. More precisely, this means that the plugin is in charge of creating the *Spark DataFrame*, which is later passed on to the storage layer. The second point for executing a plugin is after the loading of a *DataFrame*, to transform the data before it is stored in the data lake. Strictly speaking, this contradicts the principle of storing data unchanged in its raw state. However, such post-processing is useful in case data is ingested from *Kafka*, because it represents all data objects as a byte array, storing this data in the data lake would mean to store data which is not directly usable for a query engine. Therefore, a plugin can be used to transform the byte array into a more meaningful representation (e.g., a table or JSON document) and store it in a suitable storage system.

For the versioning of datasets we integrated DataBrick's *Delta Lake* API [1], which is an extra storage layer that can be applied on top of HDFS. Data is stored in so-called delta tables with additional metadata and logs. A protocol is used that allows multiple clients to read simultaneously, but only one to write at a time and supports both batch processing as well as streams and full integration with *Spark*. By default,

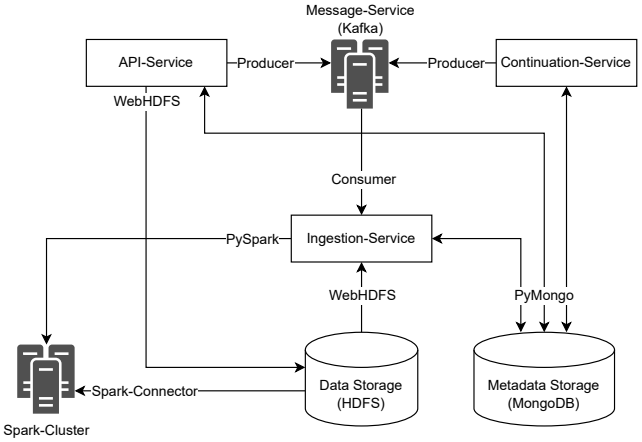

Figure 2: Architecture diagram for the ingestion interface

the most recent version is always used when reading, but it is also possible to specify a specific version.

Delta calculation for different versions of the same dataset is implemented for two distinct cases. First, in order to transfer changes to a delta table, the external data source may provide exact information about each data entry that has been added, changed, or deleted. The format used here must exactly match the schema of the original data, so that only explicit changes are processed and stored. Second, to not rely on an external change data capture system, there is an internal solution based on the algorithm by [10] that can be applied to all (semi-)structured data within the system.

Lastly, the continuation service ensures the correct execution of continuous ingestions, for example data streams or timed batch executions with delta determination. The service allows time-controlled continuous tasks, similar to cronjobs in UNIX systems.

4 DATA CATALOG

A data catalog is fundamental for any organization that stores big data, enabling data scientists to manage datasets effectively and efficiently. Yet, setting up a data catalog in a data lake remains a complicated task. *SEDAR* is an implementation to bridge this gap in a flexible and extensible manner. Any data lake design must integrate a metadata storage strategy to allow users to search, query, locate and understand all assets stored in the lake. The catalog is designed to provide a single source of information about the contents of the data lake. It contains structural metadata (schema information), descriptive metadata (e.g., information about author and creation), as well as administrative metadata (e.g., access rights). The data catalog must ensure ease-of-use for non-technical users as well as provide means to manage access to sensitive information, i.e., privacy-preserving measures.

The overall data structure of the data lake is the concept of workspaces, which enables distinct spaces for users to create own datasets and share them with others. Every workspace is equipped with a standard ontology, for this demo we are using the *EXPO* Ontology, which describes scientific experiments.

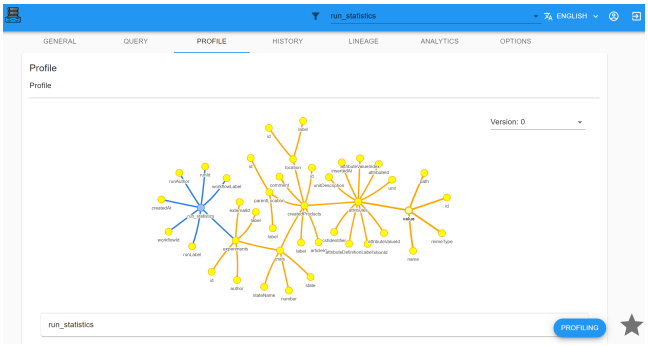

Figure 3: Screenshot of the Data Catalog, where the *Profile* tab is selected.

The system is not limited to specific ontologies; any user may add more ontologies to a workspace. Querying and storage is realized via *Fuseki*'s SPARQL endpoint, and *WebVOWL* is utilized as web service for visualizations.

For each dataset in a workspace, the system displays status and schema information. When a dataset is made available in a workspace, we enforce the specification of at least one semantic annoation, composed of a name and an ontology element, to leverage the power of the semantic layer. Additionally users may enter a language, possible authors, latitude and longitude, relevant time periods and other custom dataset attributes. Free text description can be defined in a *Markdown* editor. The dataset can be made publicly available or only to single users. Additionally, one may choose to index the source data via *Elasticsearch* and may start the profiling task if desired.

Datasets can be found by a simple keyword search or by exploiting the links that have been created between datasets. Querying is supported by *SparkSQL* queries; the derived schema and the resulting *DataFrame* are shown next to the query editor. In this way the user may query arbitrarily complex data structures, for example nested JSONs or *NoSQL* databases. The profile tab displays the schema and allows to switch between versions of the same dataset. The catalog supports annotating the schema with various options. One may add a semantic annotation to each attribute, entity, or dataset; primary and foreign key relationships to attributes of the same or other datasets can be defined and columns can be marked to contain sensitive information. *Deequ*'s profiling results are provided containing initial statistical information as well as quality measures. These quality measures can be obtained even for semi-structured data, if data is processed by a flattening operation as *Deequ* supports only relational data. The history tab displays information about the individual versions of a dataset and allows to perfrom an update using the change data capture mechanism. The lineage tab provides information about previous transformations from which the dataset might have emerged; this information is used to create links between datasets as well. The analytics tab allows to create computational notebooks, currently *Apache Zeppelin* or *Jupyter* notebooks. The corresponding code to access the datasets as a *Spark DataFrame* is added automatically to the notebook such that the user may start working with the dataset to create, e.g., visualizations straight away. Inspired by JUNEAU [15], the search bar is reachable from within a notebook in order to increase the availability of datasets.

5 DEMONSTRATION

The video demonstrates a use case from the smart factory environment, where experiments to improve properties of chemical formulations are conducted and data integration poses a significant problem. We showcase the ingestion of the sources, followed by designing a transformation workflow, enrichment of dataset metadata via the data catalog, and the execution of an update with integrated change data capture.

6 ACTIVE DEVELOPMENT AND OUTLOOK

We plan to release the code and "open-source" the project by the time of the conference, but several active development activities should be integrated before. One drawback of the existing system is the absence of advanced data quality management tools. We are working on an extended integration of the *Deequ* library for automated data cleaning operations and units tests for data which is almost completed.

Finally, to tackle the problem of data integration for heterogeneous sources, we are implementing an interface for Ontology-based data access (OBDA) [2] to utilize the power of the semantic layer not only for annotations, but also to make data access more expressive and inter-operable.

REFERENCES

[1] Armbrust, M., and et. al. Delta lake: High-performance ACID table storage over cloud object stores. *Proc. VLDB Endow.* (2020).

[2] Belcao, M., Falzone, E., Bionda, E., and Valle, E. D. Chimera: A bridge between big data analytics and semantic technologies. In *International Semantic Web Conference* (2021).

[3] Bizer, C., Heath, T., and Berners-Lee, T. Linked data - the story so far. *Int. J. Semantic Web Inf. Syst.* (2009).

[4] Farid, M., Roatis, A., Ilyas, I. F., Hoffmann, H.-F., and Chu, X. Clams: Bringing quality to data lakes. In *Proc. SIGMOD* (2016).

[5] Hai, R., Geisler, S., and Quix, C. Constance: An intelligent data lake system. In *Proc. SIGMOD* (2016).

[6] Hai, R., Quix, C., and Jarke, M. Data lake concept and systems: a survey. *arXiv e-prints* (2021), arXiv–2106.

[7] Halevy, A. Y., Korn, F., Noy, N. F., Olston, C., Polyzotis, N., Roy, S., and Whang, S. E. Managing google's data lake: an overview of the goods system. *IEEE Data Eng. Bull.* (2016).

[8] Hozdić, E. Smart factory for industry 4.0: A review. *International Journal of Modern Manufacturing Technologies* (2015).

[9] Jarke, M., and Quix, C. On warehouses, lakes, and spaces: The changing role of conceptual modeling for data integration. In *Conceptual Modeling Perspectives* (2017).

[10] Labio, W., and Garcia-Molina, H. Efficient snapshot differential algorithms in data warehousing.

[11] Lohmann, S., Link, V., Marbach, E., and Negru, S. Webvowl: Web-based visualization of ontologies. In *EKAW* (2014).

[12] Quix, C., Hai, R., and Vatov, I. Metadata extraction and management in data lakes with GEMMS. *Complex Syst. Informatics Model. Q.* (2016).

[13] Sawadogo, P., and Darmont, J. On data lake architectures and metadata management. *JIIS* (2021).

[14] Schelter, S., and et. al. Unit testing data with deequ. In *SIGMOD Conference* (2019), ACM.

[15] Zhang, Y., and Ives, Z. G. Juneau: data lake management for jupyter. *Proceedings of the VLDB Endowment* (2019).

[16] Zhao, Y., Megdiche, I., and Ravat, F. Data lake ingestion management. *arXiv preprint arXiv:2107.02885* (2021).