SN Computer Science (2024) 5:449
https://doi.org/10.1007/s42979-024-02737-0

SURVEY ARTICLE

The Lakehouse: State of the Art on Concepts and Technologies

Jan Schneider[1] · Christoph Gröger[2] · Arnold Lutsch[2] · Holger Schwarz[1] · Bernhard Mitschang[1]

Received: 9 October 2023 / Accepted: 19 February 2024

Abstract
In the context of data analytics, so-called *lakehouses* refer to novel variants of data platforms that attempt to combine characteristics of data warehouses and data lakes. In this way, lakehouses promise to simplify enterprise analytics architectures, which often suffer from high operational costs, slow analytical processes and further shortcomings resulting from data replication. However, different views and notions on the lakehouse paradigm exist, which are commonly driven by individual technologies and varying analytical use cases. Therefore, it remains unclear what challenges lakehouses address, how they can be characterized and which technologies can be leveraged to implement them. This paper addresses these issues by providing an extensive overview of concepts and technologies that are related to the lakehouse paradigm and by outlining lakehouses as a distinct architectural approach for data platforms. Concepts and technologies from literature with regard to lakehouses are discussed, based on which a conceptual foundation for lakehouses is established. In addition, several popular technologies are evaluated regarding their suitability for the building of lakehouses. All findings are supported and demonstrated with the help of a representative analytics scenario. Typical challenges of conventional data platforms are identified, a new, sharper definition for lakehouses is proposed and technical requirements for lakehouses are derived. As part of an evaluation, these requirements are applied to several popular technologies, of which frameworks for data lakes turn out to be particularly helpful for the construction of lakehouses. Our work provides an overview of the state of the art and a conceptual foundation for the lakehouse paradigm, which can support future research.

Keywords Data lakehouse · Data lake · Data platform · Data analytics

Introduction

As part of the digital transformation, data is becoming a key business resource for enterprises. Along the entire value chain, often huge amounts of heterogenous data about different products, factories, customers and other entities are generated [1]. By analyzing this data, enterprises pursue to gain insights about their performance and learn how they can optimize their business practices, possibly resulting in competitive advantages. For example, in manufacturing, the collected data can be used to improve product designs, while in the retail sector, data about the customer behavior may support the planning of marketing campaigns. In this context, *data platforms* take a crucial role: They allow to store and manage data from all kinds of data sources for analytical purposes and hence form the foundation for data collection, data processing and the provisioning of data to analytics applications [2]. While the field of data platforms has been dominated by data warehouses [3] and data lakes [4] in the past, so-called

This article is part of the topical collection "Recent Trends on Enterprise Information Systems" guest edited by Joaquim Filipe, Michał Śmiałek, Alexander Brodsky and Slimane Hammoudi.

✉ Jan Schneider
Jan.Schneider@ipvs.uni-stuttgart.de

Christoph Gröger
Christoph.Groeger@de.bosch.com

Arnold Lutsch
Arnold.Lutsch@de.bosch.com

Holger Schwarz
Holger.Schwarz@ipvs.uni-stuttgart.de

Bernhard Mitschang
Bernhard.Mitschang@ipvs.uni-stuttgart.de

1 University of Stuttgart, Universitätsstraße 38, 70569 Stuttgart, Germany

2 Robert Bosch GmbH, Borsigstraße 4, 70469 Stuttgart, Germany

Published online: 18 April 2024

lakehouses have emerged in recent years, which claim to combine features and characteristics of data warehouses and data lakes [5]. This development promises major improvements in terms of operational costs and the quality of analysis results, as conventional enterprise analytics architectures require the operation of separate data warehouses and data lakes in order to serve all kinds of analytical workloads [6]. As a result, these architectures (a) tend to become rather complex, expensive and maintenance-intensive, (b) encourage the redundant storage of multiple copies of the same data on several data platforms, which prevents the formation of a single source of truth and (c) require the development and maintenance of possibly error-prone and slow data pipelines for synchronizing the data between the data platforms, which may lead to stale, inconsistent and less trustworthy analysis results [5]. The prospect of being able to address these issues and to make data analytics more accessible raised high expectations [7]. Consequently, many vendors try to take advantage of this trend by expanding and promoting their products accordingly. However, since precise definitions and distinguishing criteria are missing, it remains unclear what challenges lakehouses actually address, how lakehouses can be characterized, what benefits they provide in comparison to conventional data platforms and which technologies actually support their implementation.

In our previous work [8], we reviewed existing definitions for lakehouses, proposed a new definition, specified a preliminary selection of technical requirements for lakehouses and evaluated several technologies against these requirements from a high-level perspective. This paper revises our previous work and significantly extends it to provide a more complete overview of the state of the art of the lakehouse paradigm and a stronger conceptual foundation. It includes the following key contributions:

- Extensively discusses lakehouse-related definitions, architectural approaches and implementations that are proposed in literature;
- Identifies five common challenges of conventional data platforms;
- Proposes a new definition that outlines lakehouses as a distinct architectural approach for data platforms that can address these challenges;
- Derives and describes eight technical requirements for lakehouses;
- Systematically evaluates 21 combinations of popular technologies by comparing them against these requirements;
- Introduces a representative analytics scenario with a baseline and a lakehouse-based implementation that support and illustrate our findings.

The remainder of this paper is structured as follows: The section "Overview of Data Platforms" provides general background information about data platforms and discusses existing concepts and definitions for lakehouses from literature. In the section "Retail Scenario and Baseline Implementation", a representative analytics scenario from the retail sector is introduced, as well as an implementation based on conventional data platforms. Supported by this scenario, the section "Challenges of Data Lakes" identifies and describes common challenges of current data lakes. The section "Definition and Requirements for Lakehouses" then represents the conceptual part of this paper, in which our new definition for lakehouses is presented and technical requirements are derived that lakehouses must fulfill in order to comply with our definition. As a demonstration of these concepts, also a lakehouse-based implementation of the retail scenario is described. The section "Technology Review and Evaluation" focuses on technologies related to lakehouses by reviewing several implementations that are proposed in literature and evaluating associated technologies against the lakehouse requirements. Finally, the section "Discussion" discusses threats to validity and points to future research directions, while the section "Conclusions" concludes our work.

Overview of Data Platforms

This section introduces data warehouses and data lakes, the two most established types of data platforms, compares them and describes how they can be integrated within enterprise analytics architectures. In addition, it discusses concepts and definitions for lakehouses, a supposedly new type of data platform, which are available in literature.

Data Warehouses and Data Lakes

Data warehouses and data lakes represent the two most prominent and still most established kinds of data platforms. Table 1 summarizes and compares their key properties.

Emerged from relational databases as a more convenient solution for large-scale data analysis, *data warehouses* represent the more established type. They typically employ well-defined and possibly multi-dimensional data schemas [9] that are continuously enforced throughout all data operations, guarantee ACID properties [10] and often provide advanced management capabilities like time travel and zero-copy cloning [5]. Modern data warehouses transfer these concepts to public clouds [11] in order to leverage their benefits, such as higher scalability and lower operational efforts. Due to their static, use-case specific data models and the storage of aggregated data instead of raw data, data warehouses are primarily suitable for answering analyses questions that are known in advance [6], such as in the scope

Table 1 Comparison of data warehouses and data lakes

Property	Data warehouse	Data lake
Typical workloads	Reporting, OLAP	Advanced analytics
Users	Business users, data analysts	Data scientists
Data access	Query language, data export	Direct access on storage
Data independence	Physical, partly logical	Weak
Guarantees	ACID	Weak
Schema enforcement	Explicitly on all data operations ("on write")	Implicitly when data is read ("on read")
Type of data	Mainly structured	All types
Data addressing	Via relational structures	Via metadata
Data granularity	Aggregated	Raw and aggregated
Data storage	RDBMS	Polyglot
Flexibility	Low	High
Management features	Advanced	Rudimentary

of reporting and online analytical processing (OLAP) [12], and barely for any kinds of advanced analytics [13].

Around 2010, these limitations gave birth to the idea of *data lakes* [14], which leverage a potentially polyglot infrastructure that may consist of several different data storage and processing systems, such as relational or NoSQL databases, batch and stream processing engines and event hubs. This renders data lakes not only suitable for the management of pre-processed and aggregated data, but also for the storage of raw data as it is provided by the data sources, which can then be gradually processed [6, 15]. Analogous to the Extract-Transform-Load (ETL) process of data warehouses, this approach is often referred to as Extract-Load-Transform (ELT). Due to their low operational costs, high scalability and flexibility, distributed file systems or object storages, such as the Hadoop Distributed File System[1] (HDFS) or Amazon S3[2], are commonly employed within data lakes for storing raw and processed data. Furthermore, open and column-oriented file formats like Apache Parquet[3] or Apache ORC[4] are often utilized, which align it column-wise to enable more efficient aggregations. These approaches render data lakes more flexible for different types of analyses than data warehouses, as no decisions regarding the modelling and processing of data have to be made in advance. Instead, the storage of raw data preserves all options regarding future analyses. However, this flexibility comes at the cost of lower robustness, as the raw data can barely be validated against a pre-defined schema on ingestion. In addition, the utilization of open file formats results in reduced processing efficiency in comparison to proprietary data warehouses, while the polyglot architecture of data lakes prevents a holistic view on the data and thus also uniform data access. Furthermore, the business value of the stored data can only be exploited when metadata management is performed [16].

In summary, it can be stated that both types of data platforms show rather contrary properties and target different types of analytical applications. Hence, companies often need to leverage both of them [17]. In our previous work [8], we discuss four basic integration patterns for combining the capabilities of data warehouses and data lakes.

The currently most commonly applied integration pattern appears to be the *2-tier architecture* [5]. Here, all source data is first ingested into a data lake and subsequently pre-processed and prepared for analytical evaluation. A second data pipeline then copies or moves relevant parts of the data from the data lake to the data warehouse, where it can be exploited for reporting and OLAP workloads. Optionally, another pipeline can offload data that is no longer required by the data warehouse back to the data lake in order to improve query performance and storage costs [18]. This pattern possesses severe drawbacks [5]: The additional data pipelines increase the overall complexity of the architecture and the potentially high number of involved data movement and transformation steps render them error-prone. In addition, they introduce additional delays, as it may take several days to transfer the data from the data lake to the data warehouse, possibly leading to stale data and outdated analyses results at the data warehouse.

The pattern of an *integrated architecture* represents a single data platform that combines desirable characteristics and features of data warehouses and data lakes, such that all analytical workloads can be served from an integrated platform. This way, no data replication or pipelines for transferring the data between platforms are necessary.

There is currently no final consensus on which of these integration patterns actually constitute "lakehouses". While

[1] https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html.

[2] https://aws.amazon.com/de/s3/.

[3] https://parquet.apache.org.

[4] https://orc.apache.org.

Table 2 Prevalent definitions of lakehouses

Source	Definition
Armbrust et al. [5]	Data management system based on low-cost and directly-accessible storage that also provides traditional analytical DBMS management and performance features such as ACID transactions, data versioning, auditing, indexing, caching, and query optimization
Gartner [31]	Converged infrastructure environment that combines the semantic flexibility of a data lake with the production optimization and delivery of a data warehouse
	Supports the full progression of data from raw, unrefined [to] optimized data for consumption
Oracle [32]	Modern data platform" that "takes the flexible storage of unstructured data from a data lake and the management features and tools from data warehouses, then strategically implements them together as a larger system
Hansen [24]	Architectural approach for managing all [types of data] and supporting [all] data workloads (Data Warehouse, BI, AI/ML, and Streaming)

many authors consider the integrated architecture as a lakehouse [5, 19, 20], others argue that also variants of the 2-tier-architecture, possibly extended by a layer for uniform data access, already represent lakehouses [18, 21, 22].

Concepts and Definitions for Lakehouses

After the term "lakehouse" was presumably used for the first time by Alonso [23] and in a company presentation by Jellyvision [24], the lakehouse idea primarily took shape with the emergence of the open source framework Delta Lake[5]. It intends to allow the construction of integrated data platforms by enhancing data lakes, which are based on distributed file systems or object storages, for typical features and characteristics of data warehouses (cf. section "Data Warehouses and Data Lakes"). The underlying concepts are explained in the paper by Armbrust et al. [25], which refers to Delta Lake as a novel kind of "*ACID table storage layer over cloud object stores*", and in their subsequent paper [5], which discusses issues of typical enterprise analytics architectures and emphasizes the benefits of an integrated lakehouse platform in comparison to the established 2-tier architecture. Harby and Zulkernine [26] take up on these concepts and compare the characteristics of such lakehouses with those of typical data warehouses and data lakes.

Other perspectives on the lakehouse paradigm exist as well: Oreščanin and Hlupić et al. [18, 21] use the term "lakehouse" to describe an architecture similar to the 2-tier approach, which combines self-contained data warehouses and data lakes. According to Azeroual et al. [27], lakehouse-like characteristics can be achieved by combining data lakes with practices of data wrangling. Others share the opinion that modern, cloud-based data warehouses already represent lakehouses, since they have adopted common features of data lakes [24, 28]. In contrast, Inmon et al. [29] and Shiyal [20] argue that lakehouses are always built on top of data lakes. Due to these different views, Raina and Krishnamurthy [30] conclude that the term "lakehouse" should be used to describe the general vision of combining data warehouses and data lakes, rather than for categorizing individual tools. Similarly, Eckerson [28] considers the term "lakehouse" just as a metaphor that in principle applies to every modern data management offering and some authors even use the term interchangeably for their entire enterprise analytics architecture or as a generic term for data platforms in general.

However, this paper argues that lakehouses can indeed provide value over other enterprise analytics architectures and possess characteristics that distinguish them from common data warehouse and data lake solutions. Nevertheless, there is still no generally accepted definition of the term "lakehouse" that considers these distinct characteristics. Table 2 summarizes the main aspects of existing attempts to define lakehouses. As discussed in our previous work [8], they impose shortcomings, as they are often: (a) formulated by vendors of data management technologies and are hence *driven by marketing* and the characteristics of individual tools that they intend to promote, (b) referring almost arbitrarily to a set of characteristics with *unclear relevance*, motivation and lacking reasonings, (c) focus solely on functional features and *lack statements regarding the underlying architecture* or (d) are *too abstract* for actually distinguishing lakehouses from other types of data platforms and hence cannot highlight their specific benefits for enterprises. Due to these issues, we proposed a novel definition for lakehouses [8], whose derivation and underlying considerations are extensively described in the section "Definition and Requirements for Lakehouses".

Recently, several works have been published that extend and adapt lakehouse concepts for specific domains: For example, Lui et al. [33] propose a reference architecture for lakehouses in the Internet of Things, Zhang et al. [34] and Errami et al. [35] present concepts for the efficient management of geospatial data, while Vox et al. [36] investigate optimizations for vehicle sensor data. However, as these

[5] https://delta.io.

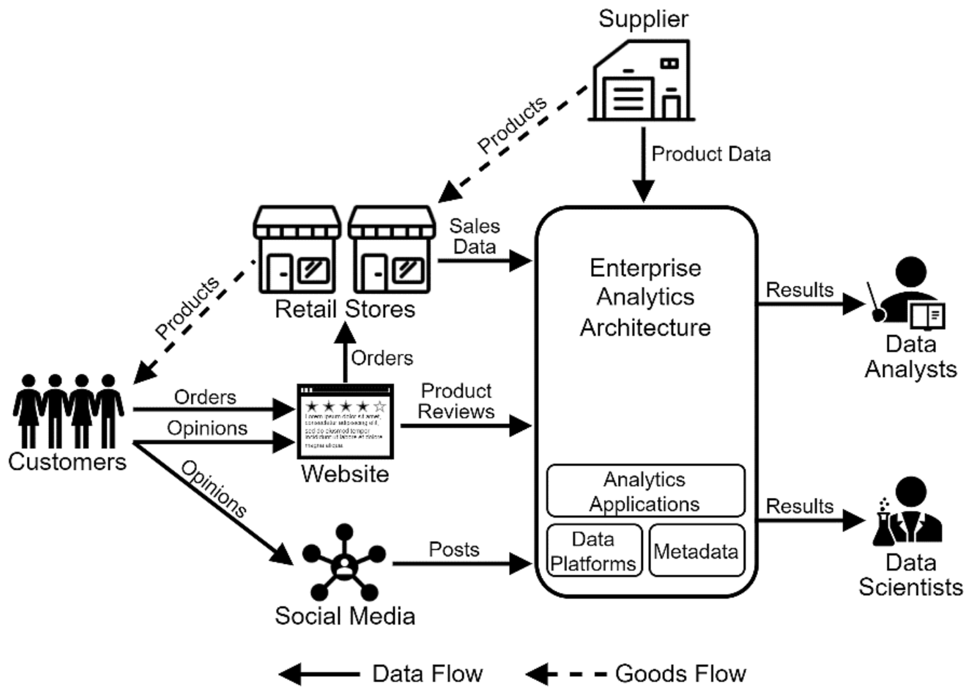

Fig. 1 High-level overview of the retail scenario

works are limited to their domains and not generally applicable, they are not considered further in this paper.

Retail Scenario and Baseline Implementation

This section introduces a representative analytics scenario from the retail business field that serves as a running example throughout this paper and is supposed to demonstrate the reasonings and conceptual choices. In the remainder of this paper, two implementations for this scenario are presented and discussed. While the *baseline implementation* utilizes conventional data platforms, i.e., a data warehouse and a data lake, the second one represents a *lakehouse-based implementation* that leverages a lakehouse that complies with our definition. With the help of the retail scenario and these two implementations, the different characteristics of the data platforms and their implications for enterprise analytics architectures can be vividly compared. The analytics scenario presented in this section is based on typical conditions and challenges of the retail business field [37, 38], with corresponding business processes [39] and data sources [40], including operational IT systems, such as Enterprise Resource Planning (ERP) systems and openly accessible data sources like social media platforms. Retailing can significantly benefit from data analytics [41–43]. For example, analyzing sales data may provide insights on the shopping behavior of customers and allow to align the pricing of products dynamically. Our scenario covers a variety of such use cases, including descriptive, diagnostic, predictive and prescriptive analysis questions [44]. The section "Scenario Outline" outlines this scenario, while the section "Baseline Implementation with Data Warehouse and Data Lake" presents the baseline implementation that leverages a conventional data warehouse and a data lake.

Scenario Outline

The scenario is about a retail chain that sells sports equipment. Figure 1 depicts a high-level overview, including all involved entities and the flows of goods and data. For simplicity, it abstracts from the operational IT systems that are leveraged by the entities.

The chain operates two physical *retail stores*, which are located in London and Stuttgart, and acquires its *products* from a *supplier*, whose offerings and prices may change on a weekly basis. By visiting a *website* that is offered by the retail chain, *customers* can browse through the assortment of the stores, place orders and rate their purchases by writing textual comments. In addition, it is assumed that the customers also spread their opinions on individual products via *social media* platforms.

The enterprise behind the retail chain pursues to apply data analytics for improving its business model and hence needs to design and implement an *enterprise analytics architecture* that allows to collect, manage, prepare and analyze data from the different sources. This architecture includes *data platforms*, which are either a data warehouse and a data lake (cf. section "Baseline Implementation with Data Warehouse and Data Lake") or a lakehouse (cf.

Table 3 Overview of the analytics use cases that are reflected in the retail scenario

#	Analytics type	Interval	Description
U1	Descriptive	Monthly	Reporting in terms of sales and revenues
U2	Descriptive	Near real-time	Live dashboard for sales
U3	Descriptive, diagnostic	Irregular	OLAP for identifying and assessing patterns in the buying behavior and preferences of customers
U4	Diagnostic	Irregular	Data mining for explaining the buying behavior and preferences of customers
U5	Descriptive, diagnostic	Near real-time	Sentiment analysis of product reviews and social media posts for detecting trends early
U6	Predictive	Irregular	Prediction of the success of marketing campaigns
U7	Prescriptive	Weekly	Recommendation of optimal product prices

Table 4 Overview of the data sources that are available in the retail scenario

Name	IT system	Type	Provided data
London store	ERP[a]	Batch or Stream	Data about the sales made in the retail store in London; prices in GBP
Stuttgart store	ERP	Batch or Stream	Data about the sales made in the retail store in Stuttgart; prices in EUR
Supplier	SCM[b]	Batch	List with details about the products that are currently offered by the supplier
Customers and marketing	CRM[c]	Batch or Stream	Details and materials on past marketing campaigns as batch; customer reviews created on the website as stream
Social media	n/a	Stream	Social media postings related to products that are sold by the retail chain
Central bank	n/a	Batch	Current exchange rates for currencies

[a]Enterprise Resource Planning System

[b]Supply Chain Management System

[c]Customer Relationship Management System

section "Lakehouse-based Implementation of the Retail Scenario"), tools for managing associated *metadata* and *analytics applications* that can exploit the data that is available on the data platforms. Overall, seven analytics use cases are supposed to be implemented, which are summarized in Table 3. Here, the second column lists the types of analytics that are involved in each use case, while the third column indicates how often the analyses are run: either continuously in near real-time, automated at fixed time intervals, or manually at irregular times. With U1, the enterprise wants to automate the creation of business reports in order to gain more recent and more detailed insights into sales figures and revenues, while U2 aims at reporting for sales in near real-time, which would allow to detect trends in the sales data more quickly. In addition, the enterprise pursues to gain insights on the shopping behavior of the customers, i.e., with respect to external events like Christmas, analyzing the data with OLAP to identify and assess preliminary patterns in accordance with U3, as well as by generating data mining models that may be able explain the behavior and preferences of customers (cf. [45, 46]), as specified by U4. The purpose of U5 is to enable the early detection of trends regarding the demand and popularity of certain products by applying sentiment analysis to product reviews and social media posts (cf. [47, 48]). Furthermore, U6 should allow the enterprise to optimize and personalize marketing campaigns by predicting their success on the basis of previous campaigns (cf. [49]), while the goal of U7 is to generate recommendations for the optimal pricing of products (cf. [50, 51]) based on various factors, such as the store location and the expected supply and demand.

In total, six data sources are available in the retail scenario, which provide the data that is necessary for the implementation of the different use cases and are described in Table 4. The second column lists the type of operational IT system in which the enterprise manages the data before it is extracted and loaded onto a data platform. The table also indicates whether the data sources provide the data as a batch of records on request, or whether it can be delivered as a continuous stream, so that each new record is transmitted as soon as it becomes available at the source (cf. Akidau et al. [52]).

Since the retail store in London uses British Pounds (GPR) as currency in its data and the retail store in Stuttgart relies on Euros, the sales records must be converted by using the exchange rates of the central bank before meaningful analyses can be carried out. In addition, the sales data of both retail stores may be provided in different formats, which is why further transformation and cleansing steps are required. In addition, as it is unclear in advance which extracts of the data may be of relevance for the advanced

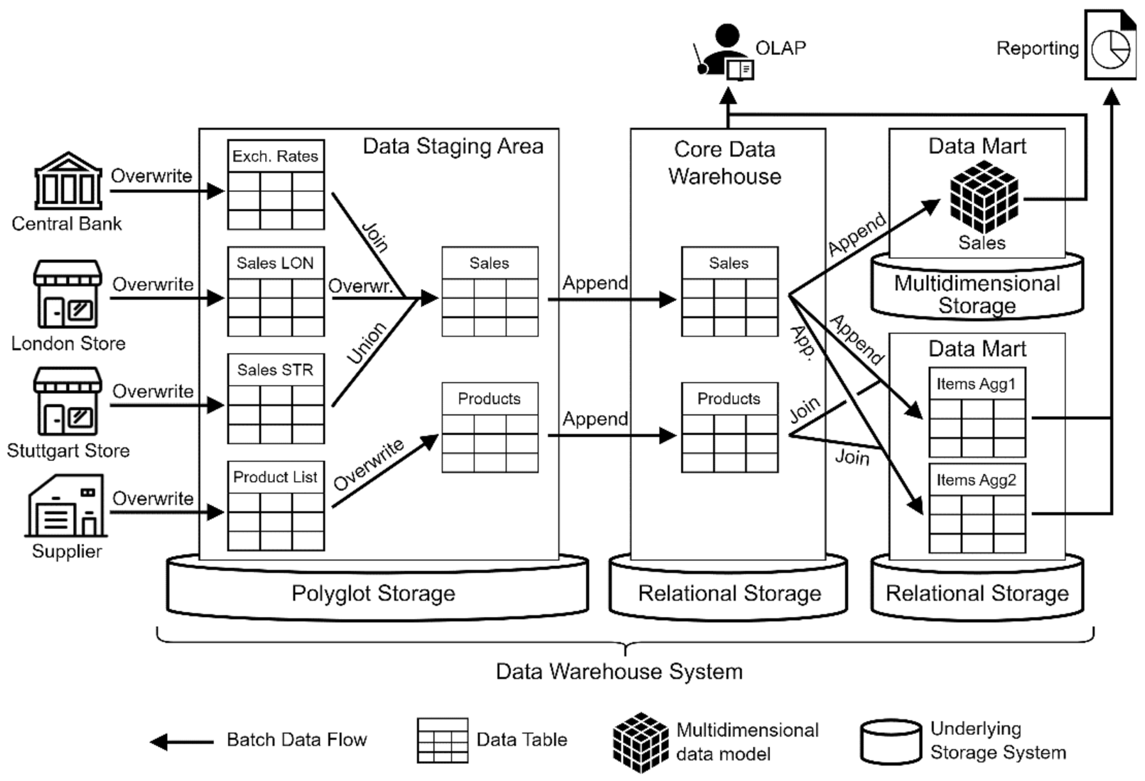

Fig. 2 Baseline implementation of the use cases U1 and U3 with a data warehouse

analytics use cases, the raw data must be stored next to aggregated data.

Baseline Implementation with Data Warehouse and Data Lake

As discussed in the section "Data Warehouses and Data Lakes", there are several ways to implement the retail scenario within an enterprise analytics architecture. This section illustrates an implementation based on traditional data platforms, i.e. a conventional data warehouse and data lake. As this implementation leverages commonly applied architecture patterns and technologies [6, 53], it can be considered as representative for many enterprise analytics architectures that are employed in practice. The baseline implementation applies the integration pattern of the *parallel architecture* [8], for which a data warehouse is operated next to a data lake and relevant data is ingested into both data platforms in parallel, where necessary. Since the use cases U1 and U3 represent reporting and OLAP workloads, they should preferably be implemented on the data warehouse due to its stronger support for these types of analytics, while the remaining use cases fall into the field of advanced analytics and hence should be carried out on the data lake.

Figure 2 illustrates a possible implementation of U1 and U3 on a data warehouse with particular focus on the data flows. The underlying architecture is based on the reference architecture by Bauer and Günzel [54], but omits some components that are not necessary for explaining the basic approach. In accordance with this reference architecture, it can be considered as a *data warehouse system*, which in addition to the *core data warehouse* features two other important components: The *data staging area* [54, 55] represents the core of the ETL process, which is responsible for ingesting new data from the data sources into the data warehouse. As part of this process, the data is extracted from the sources and temporarily stored in the data staging area in order to be cleaned, prepared and transformed for the target schema, before it is loaded into the data warehouse. While the core data warehouse is usually based on relational storage, various types of storage systems can be used for the data staging area, including NoSQL databases and file systems [12]. In the retail scenario, the sales data of the retail stores in London and Stuttgart, the current list of products that are offered by the supplier, as well as the daily updated exchange rates for currencies are extracted as batches from the corresponding sources and dumped into the data staging area. There, the data of both retail stores is normalized and merged into a common table, with all values in GBR being converted into Euros according to the exchange rates from the central bank. In addition, the product list of the retailer is flattened and transformed into the target schema of the data

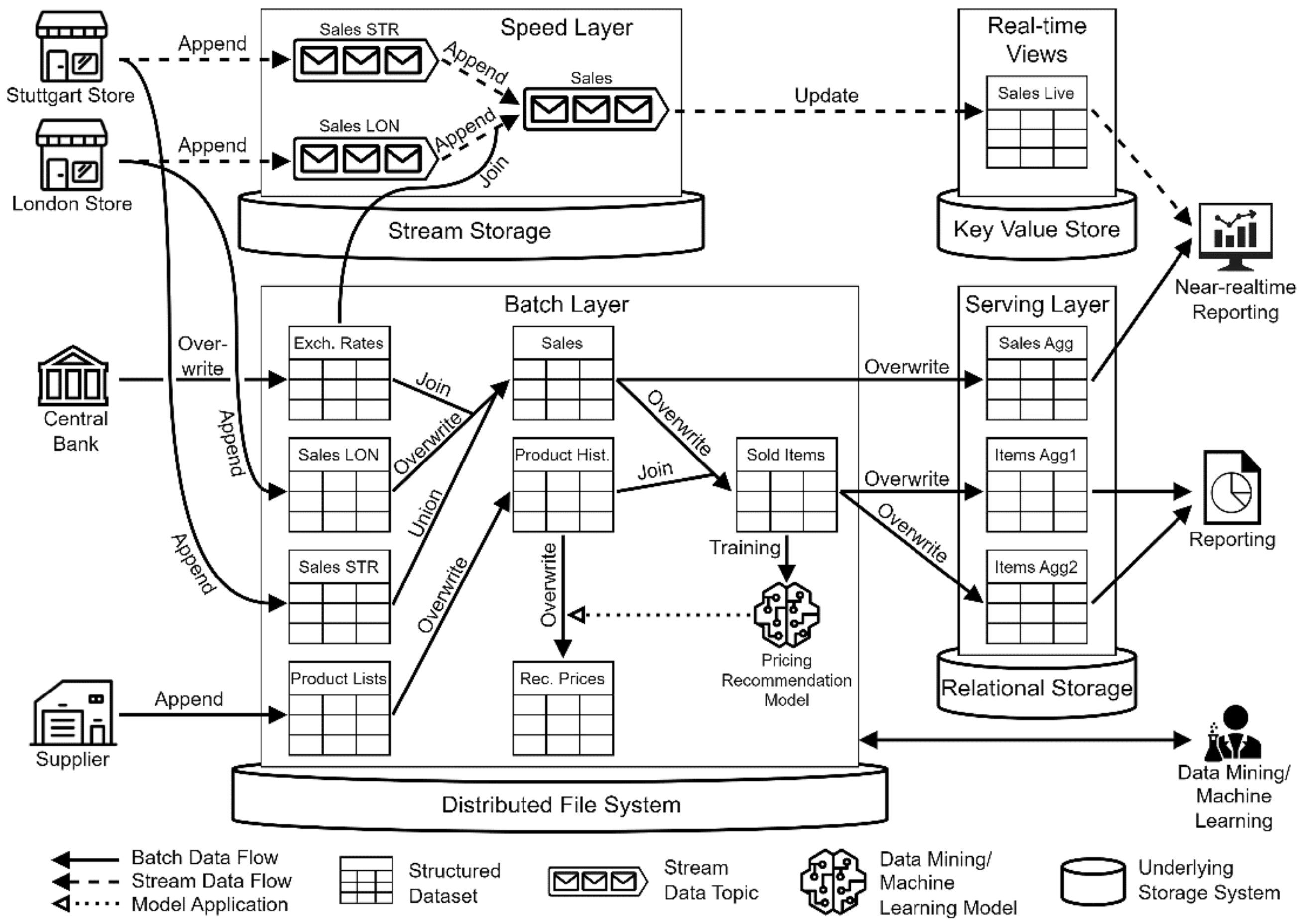

Fig. 3 Baseline implementation of the use cases U2, U4 and U7 with a data lake

warehouse. From the data staging area, the data is incorporated into the core data warehouse, which collects all data and makes it available for queries.

Due to the large amounts of data that has to be managed, it is reasonable to outsource extracts to so-called *dependent data marts* [12] in order to increase the efficiency of the analyses. This is particularly appropriate for the data that is relevant to the reporting use case U1, as the structure of the reports usually stays the same over long periods of time and hence leads to re-occurring analyses that can be performed on pre-defined extracts of the data. The OLAP use case U3 can benefit from data marts, as they may support multi-dimensional data schemas and hence allow to represent the data as OLAP cubes [56] that can be queried with the help of high-level query languages.

It is worth noting that all data movement and transformation steps in the data warehouse are carried out via periodically executed batch processing tasks, as data warehouses and the underlying relational storage systems are usually optimized for the batch-like processing of huge amounts of data and less performant in handling incremental changes with high frequency. Therefore, the use cases U2 und U5, which require stream processing in near real-time, but also the use cases U4, U6 and U7, which involve data mining or machine learning and hence require access to raw data, should rather be implemented on a data lake than on a data warehouse. Figure 3 indicates how such an implementation of the use cases U2, U4 and U7 may look like.

This data lake consists of four different types of storage systems: (a) a stream storage system such as Apache Kafka[6] that can ingest, persist and deliver huge amounts of streaming data; (b) a NoSQL key value store like Redis[7] that supports efficient reads and writes for storing and serving real-time data; (c) a distributed file system like the HDFS that provides high flexibility and (d) a relational database that stores and serves the batch processing results. The architecture of the data lake follows the popular Lambda architecture pattern [57], in which incoming data is processed separately

[6] https://kafka.apache.org.

[7] https://redis.io.

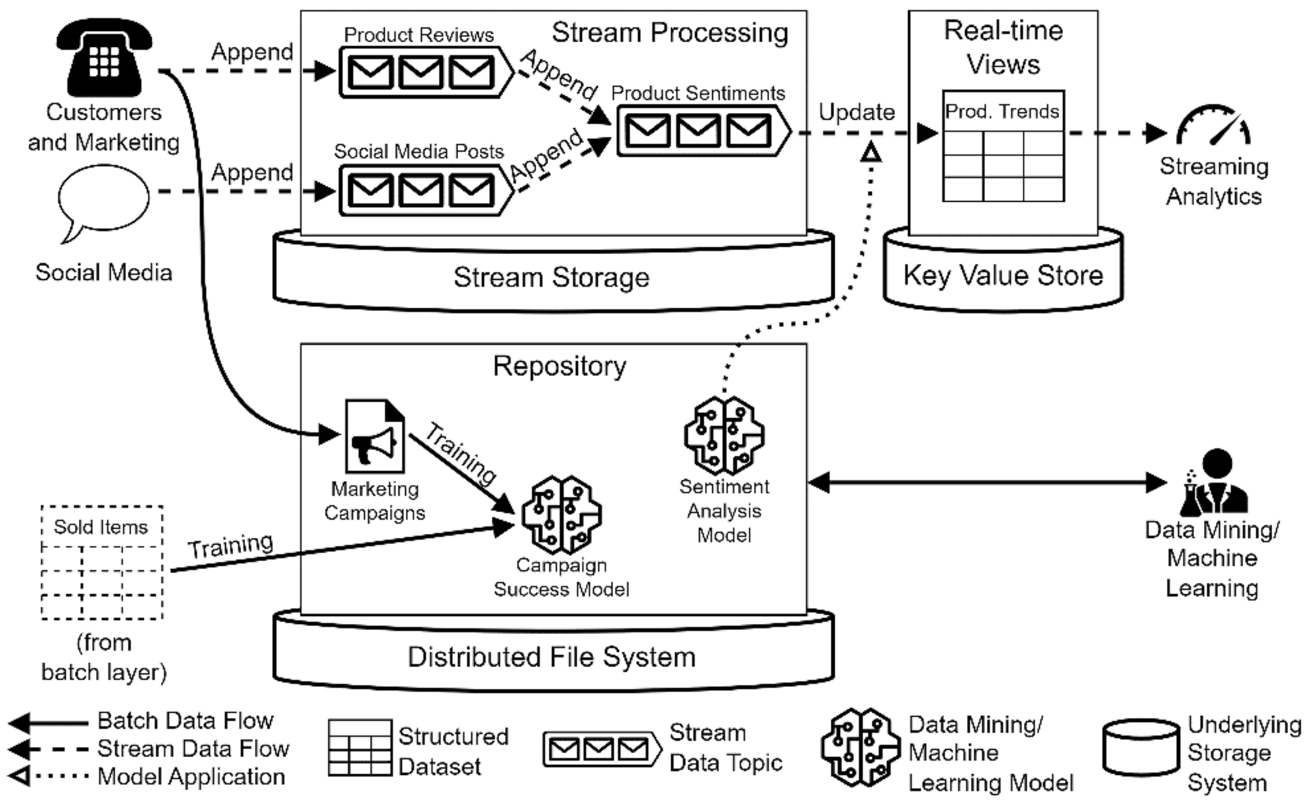

Fig. 4 Baseline implementation of the use cases U5 and U6 with a data lake

on a speed and a batch layer. While the *speed layer* processes only the most recent data and incrementally updates its results in near real-time, the *batch layer* stores all data as a master dataset and periodically re-computes its results entirely. This way, the speed layer can provide up-to-date, but potentially error-prone results in near real-time, while the batch layer also incorporates all historical data and hence delivers more reliable results, but only between larger time intervals. Analytics applications must combine the results of the batch and speed layer in order to obtain a complete view on the data.

In the depicted data lake implementation, the Lambda approach applies to the sales data of both retail stores, which is processed in the speed as well as the batch layer. The stream layer utilizes the stream storage to temporarily persist the stream data from the retail stores under predefined topics, the batch layer on the other hand stores the extracted data as datasets on the distributed file system in table-like formats, such as Apache Parquet. Since the exchange rates of the central bank can only be imported as batches, it is stored on the batch layer as well, where it can be accessed by stream processing engines as part of a stream-static join[8]. Finally, the results of the processing steps in the stream layer are incrementally updated in the key value store and the results of the batch layer are periodically written to the relational storage that acts as the *serving layer* of the Lambda architecture. A dashboard, as required for use case U2, can read and combine the results of both layers and hence display current sales data in near real-time. In addition, the raw and processed data of the batch layer that is stored on the distributed file system can be accessed by data scientists for data mining, such as clustering or association rule generation, which satisfies the use case U4. By joining the sales data with the detailed product information of the supplier, training data for a machine learning model can be generated, which is supposed to recommend optimal product prices in accordance with the use case U7. In this implementation, the trained model is periodically applied to the most recent product data of the supplier and the resulting price recommendations are stored in a separate dataset on the batch layer where they can be further processed or incorporated into data mining analyses. In addition to the use cases U2,

[8] https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html.

U4 and U7, the processed sales data of the data lake could also be used for implementing the reporting use case U1, as indicated in Fig. 3. However, data warehouses typically offer more extensive support and more convenient features for reporting than traditional databases and are hence preferred by business analysts.

The implementation of the remaining use cases U5 and U6 can be accomplished in a similar way on the data lake, which is illustrated in Fig. 4. Here, the distributed file system primarily acts as a repository for machine learning models and raw data. For use case U5, the stream of product reviews that are created on the company's website is pre-processed and combined with the pre-processed stream of social media post that have been identified by a social media crawler as related to one or multiple products offered by the retail stores. The individual customer messages are then classified using a machine learning model for sentiment analysis, from which trends regarding the sentiment of customers with respect to individual products can be derived. These results are then updated in the key value store and from there can be consumed by streaming analytics applications. The machine learning model itself is again stored in the distributed file system of the data lake, where it can be configured, trained, tested and maintained by data scientists. Since the distributed system is flexible enough to store not only store structured datasets, but data files of any format, it is also suitable for storing the heterogeneous data of past marketing campaigns, which may include structured information, as well as documents, images and video files. Data scientists can correlate this data with the processed data about the sold products and use it for data mining, but also treat it as training data for machine learning models, which may be able to predict the success rates of future marketing campaigns in accordance with use case U6.

Challenges of Data Lakes

The section "Data Warehouses and Data Lakes" described the limitations of traditional data warehouses, which finally resulted in the emergence of data lakes. However, the development and operation of data lakes poses challenges as well, which are also reflected in the baseline implementation and can be considered representative for many data lake in industrial practice [17]. This section discusses some of these challenges and thereby demonstrates the relevance and motivation for the development of novel architectural approaches for data platforms.

C1: Complex Architecture

The proposed architecture of the data lake in the baseline implementation consists of at least four different storage systems and, in addition, at least one processing engine that supports batch and stream processing. Typical data lakes may incorporate even more different technologies and also leverage different formats (e.g. Apache Parquet, Apache ORC[9], CSV) for storing structured data on the distributed file system. On the one hand, this heterogeneity means that the data can be stored efficiently in storage systems and formats that are specifically tailored towards the individual characteristics of the data and its use cases. For example, for storing the results of the speed layer, a storage system that supports efficient random reads and writes can be chosen, while the results of the batch layer can be stored in a relational system that is optimized for the ingestion of batch data and also offers a query language. On the other hand, however, this heterogeneity also raises the need for several data transportation and transformation steps, as the data must be moved between the different storage systems and each system may support different data models, data types and formats. This can lead to inconsistencies, e.g. when precision is lost during the conversion of floating-point numbers or when complex data types differ in their semantics. In addition, it introduces high operational costs and a large maintenance overhead, as all the involved technologies, their accompanying technical infrastructure and the associated code artifacts must be maintained. For example, the addition of a new data source or even simple changes to the structure of the data may need to be reflected in lots of different storage and processing systems, as they are tightly coupled and depend on each other. All of these aspects increase the error-proneness and maintenance demands of the data lake.

C2: Combining Batch and Stream Processing

The Lambda architecture and its variants and derivatives (e.g. Lambda + [58]) still represents a very commonly used system architecture for data lakes that need to combine batch and stream processing and is also utilized in the proposed implementation of the retail scenario. An inherent issue of the Lambda architecture is its need for separate batch and speed layers, which usually have to be implemented with different processing engines and storage systems. As a consequence, major parts of the application logic have to be implemented twice, once for batch and once for stream processing. This applies to the actual processing logic, such as projection, selection, and aggregation operations, but also to the additional logic that is required for the extraction, validation and de-duplication of data as well as for writing the processing results to the target storage systems. In the implementation of the retail scenario, the logic for retrieving and validating the sales data of the retail stores, the currency

9 https://orc.apache.org.

conversion based on the exchange rates, the necessary data aggregations and also the transformations and materializations of the results on the storage systems must therefore be implemented twice. As a consequence, two different code-bases that represent almost the same logic have to be maintained, leading to additional development and maintenance efforts and tedious troubleshooting. In addition, the operation of two parallel processing branches, where one branch periodically re-computes all data, is expensive.

The Kappa architecture [59], which attempts to implement the entire application logic as possibly concurrently executed stream processing jobs and does not employ a batch processing branch at all, can remedy these drawbacks to a certain extent, as only a single code-base has to be maintained. However, the Kappa architecture also possesses severe limitations, as many complex data processing tasks that go beyond simple relational algebra are difficult to implement using stream processing. This includes the training of machine learning or data mining models and other non-trivial analytics tasks, but also computation-intensive tasks, such as joins over large datasets, where the processing engines would need to keep huge amounts of data in memory.

Modern processing engines, such as Apache Spark and Apache Flink[10], support batch and stream processing and pursue to transfer the semantics of batch processing operations to the processing of unbounded data streams. Furthermore, they allow to read streaming data from and to write streaming data to typical batch storage systems, such as distributed file systems or object storages. The ability to use the same technologies and similar application logic for batch and stream data reduces complexity and contributes to the unification of batch and stream processing. However, some of these processing engines lack important features for storing and organizing incoming stream data efficiently and reliably on distributed file systems and object storages, such as write compaction [25]. Since most processing engines treat data files on distributed file systems or object storages as immutable, they create new data files for each incoming bulk of new data, which tends to spread stream data across a large number of very small files. Without compaction or comparable measures, this large number of data files may reduce the efficiency of downstream processing tasks or queries. On the other hand, some compaction techniques may introduce additional delays for write operations, leading to a trade-off between write latency and query performance [25]. These and further aspects must be considered when designing a data lake and hence constitute challenges for combining batch and stream processing.

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

C3: Incremental Processing

The immutability of data and the complete re-computation of all data in the batch layer are core principles of the Lambda architecture and were also followed for the implementation of the retail scenario, e.g. for the processing of the retail store's sales data. While a complete re-computation is reasonable in order to increase error tolerance and produce reliable results, it also imposes challenges. Especially in the case of the Lambda architecture, where the re-computation is not only performed in exceptional cases, but periodically and frequently, it leads to an enormous computational burden, which can consume a lot of resources and time and may lead to large costs. For this reason, Marz and Warren [57] suggest to apply various optimization techniques, such as partial re-computation and integrating incremental processing into the batch layer in order to achieve suitable trade-offs between efficiency and fault-tolerance or to completely switch to an incremental batch layer where all computations are performed incrementally. However, common processing engines like Apache Spark and Apache Flink do not support incremental batch processing on distributed file systems or object storages. Under the hood, these engines represent datasets as a collection of multiple data files, each of which contains segments of the dataset's data and stores it in open formats like Apache Parquet. Together, all of these data files reflect the data of the corresponding dataset. In order to reduce the error-proneness when processing the datasets in multi-worker clusters of the processing engines or accessing them concurrently as part of multi-step data pipelines, data files are typically considered immutable, which means that they can only be added, deleted, or overwritten in their entirety. Hence, random updates or deletes of individual records in these data files are not possible and instead would always require the entire data file to be rewritten [60]. As overwriting existing data files is inefficient and may interfere with concurrent processes that read from the dataset, processing engines tend to rely on a processing model that only allows to append new data to existing datasets and does not support other modifications [61, 62]. As a consequence, this not only prevents incremental batch processing, but also makes manual updates to the data sets more difficult, for example when errors in the data need to be corrected or when the schema needs to evolve.

By using stream instead of batch processing, some kind of incremental processing can be achieved, as the processing engine keeps an internal state in memory due to which only the new incoming data has to be processed and incorporated into the results. However, writing the results to a distributed file system or an object storage typically again requires to overwrite existing datasets [63]. Furthermore, it may still be necessary to combine the "incremental" stream processing with batch processing, as stream processing in its current state is not able to replace batch processing entirely. As already discussed for C2, this combination remains challenging.

C4: Consistency Enforcement

The implementation of the retail scenario leverages a distributed file system like the HDFS as underlying storage for the batch layer, which represents a very common design choice for data lakes. Distributed file systems and also many object storages basically behave like a simple, but highly scalable data repository, into which almost unlimited amounts of arbitrary data can be dumped as files of any formats. Unlike many relational or NoSQL databases, these systems typically do not offer any dedicated features for validating the internal consistency of the stored datasets. In the example implementation, the distributed file system can for instance neither ensure that the data in the currency exchange rates dataset (*Exch. Rates)* actually complies to the intended data schema and possesses all required attributes and the expected data types, nor that the dataset exists at the intended storage path at all. Accordingly, all of these basic validation tasks must be implemented manually in the application logic of the processing steps, which results in additional development efforts and huge amounts of boilerplate code. For example, the Spark SQL module[11] allows the definition of schemas within Apache Spark against which new data can be validated. However, without using a Hive Metastore, these schemas are not stored or registered globally on the level of the underlying dataset and thus not implicitly enforced [64]. Hence, they need to be explicitly specified and enforced in each processing step that accesses the dataset. In addition, the information about the structure of the data is spread across several code bases this way, leading to low transparency for developers, increased error-proneness and high efforts when changes to the structure of the data have to be performed.

C5: Assurance of Atomicity and Isolation

In many data lakes and also in the implementation of the retail scenario, several sequential processing steps must be executed that depend on each other. For example, the processing step that generates the dataset *Sold Items* depends on the processing steps that produce the datasets *Sales* and *Product Hist.*, as these datasets constitute the input data. Similarly, also the processing step that processes and converts the sales data of the retail store in London (*Sales LON*) within the speed layer depends on the processing step that extracts the currency exchange rates from the central bank. In order to be able to ensure that the interdependent processing steps can be executed correctly and produce the expected results, ACID properties similar to the processing of transactions in database systems are desirable. Otherwise, typical concurrency anomalies such as lost updates may occur, which can lead to either no results or, even worse, incorrect results. Since the processed data forms the basis for analytics and thus possibly for important business decisions, incorrect results may have fatal consequences.

Durability (the "D" of ACID) is typically provided by the underlying storage infrastructure, which in terms of the baseline implementation is a distributed file system, such as the HDFS. The consistency (the "C" of ACID) of the data, i.e. the guarantee that a consistent state is maintained after the execution of a processing step has been completed, must be ensured in two ways. First of all, it requires a correct implementation of the processing steps, so that they leave the data in a consistent state with respect to certain domain-specific constraints and rules, as the underlying storage system may not be able to provide additional consistency guarantees as known from relational databases (cf. challenge C4). Secondly, the atomic execution of the processing steps is another necessary precondition for maintaining a consistent state. However, ensuring atomicity (the "A" of ACID), i.e. that the execution of a processing step either completely succeeds or does not make any changes to the data, poses a major challenge: For example, the popular batch processing engine Apache Spark does not guarantee atomicity for its processing tasks and also does not utilize any locking mechanisms for datasets in file storages [65]. In Spark's *Overwrite* mode, in which an existing dataset is supposed to be overwritten by updated data, this lack of atomicity becomes obvious: If errors occur in the application logic of a currently executed processing step, these will not only lead to the abortion of the execution, but also corrupt the original dataset, since it was deleted before the new data is written. This violates the atomicity condition, as after the abortion of the processing step the dataset is neither in the expected post-execution state, nor in the original pre-execution state, which can lead to fatal consequences for dependent processing steps downstream. For stream processing, such as within the scope of Apache Spark's Structured Streaming module, where data is ingested into the storage system with high frequency, these problems may become particularly noticeable. In addition, ensuring isolation (the "I" of ACID) poses another challenge: Processing engines like Apache Spark attempt to address it by considering datasets immutable and writing processing results as files into a temporary directory, which is finally moved to the target location once the execution of the processing step concluded. The successful relocation of the temporary directory corresponds to the commit of the processing step and can be achieved with the help of an atomic rename operation, which is available on

[11] https://spark.apache.org/docs/latest/sql-programming-guide.html.

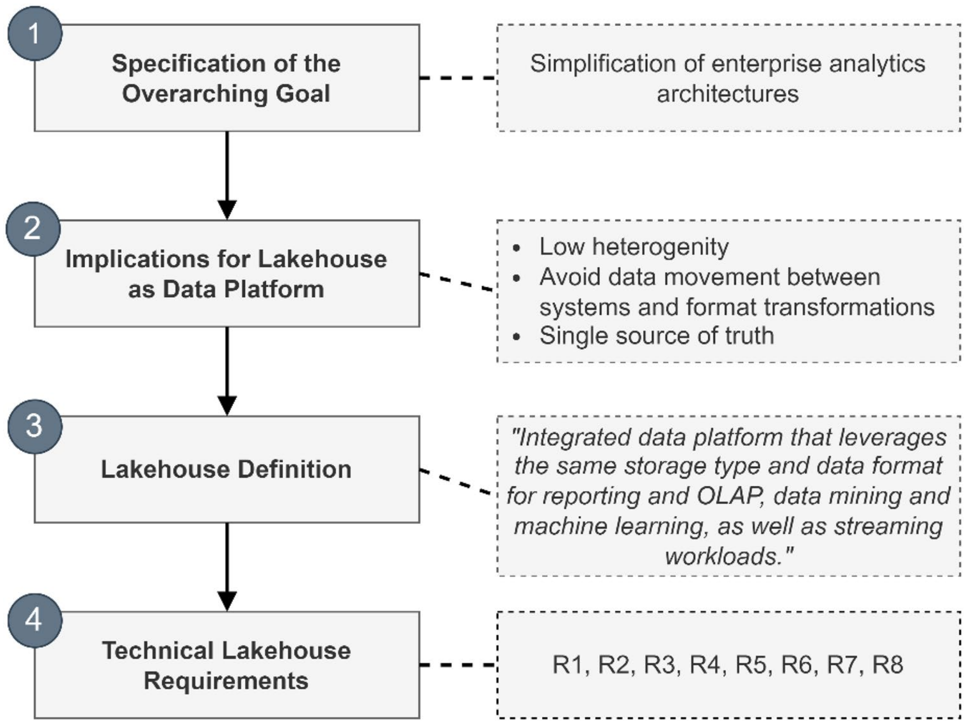
1
Specification of the Overarching Goal
Simplification of enterprise analytics architectures
2
Implications for Lakehouse as Data Platform
• Low heterogenity
• Avoid data movement between systems and format transformations
• Single source of truth
3
Lakehouse Definition
"Integrated data platform that leverages the same storage type and data format for reporting and OLAP, data mining and machine learning, as well as streaming workloads."
4
Technical Lakehouse Requirements
R1, R2, R3, R4, R5, R6, R7, R8

Fig. 5 The top-down method for creating a definition for lakehouses

many file systems and also represents a prerequisite for the HDFS [66]. This way, processing steps in execution cannot see intermediate results of other processing steps running in parallel and isolation is granted. However, other file systems, despite similar to the HDFS, may relax some of these requirements. For example, the object storage Amazon S3, which among object storages from other cloud service providers is commonly utilized as a cloud-based replacement for the HDFS, does not provide atomicity for directory renames [67] and lacks "put-if-absent" semantics [68]. Consequently, additional efforts may be required for ensuring isolation, e.g. by implementing locking mechanisms within the cluster of the processing engine or by leveraging additional storage systems that provide atomic operations [68].

In summary, it can be stated that atomicity and isolation generally cannot be taken for granted when implementing an analytics scenario on a data lake. Instead, when choosing storage and processing systems, explicit attention must be paid to whether and to which degree these technologies guarantee atomicity and isolation and what additional precautions must be taken to achieve them.

Definition and Requirements for Lakehouses

This section proposes a novel definition for lakehouses that pursues to overcome the issues of the existing definitions as discussed in the section "Concepts and Definitions for Lakehouses". In order to illustrate the reasoning behind this definition and to demonstrate how such a data platform can address the previously identified challenges, the section "Lakehouse-Based Implementation of the Retail Scenario" presents a lakehouse-based implementation for the retail scenario and compares it to the baseline implementation. Next, the section"Analytical Workload Characteristics" characterizes the different types of workloads that are mentioned in the new definition, while the section "Derived Technical Requirements" uses these insights to derive a total of eight mandatory technical requirements for lakehouses.

Defining the Lakehouse

As indicated in Table 1, most of the characteristics of typical data warehouses and data lakes are rather contrary, for example with respect to data access, data independence and the storage type. Hence, a straightforward combination of both concepts into one universal data platform that preserves all desirable properties is not possible. Instead, data warehouses and data lakes typically need to give up or relax some of their characteristics in order to be able to adopt features from the respective other data platform. For instance, a data warehouse that is supposed to provide direct access to its data to support data mining and machine learning must give up parts of its data independence. Similarly, a data lake can only integrate management features from data warehouses, such as ACID characteristics, if it restricts the freedom for its users to store and access the data arbitrarily and instead introduces protocols for read and write operations. However, the combination of both types of data platforms into a lakehouse may also lead to new, emergent characteristics, such as the possibility to create a single source of truth by serving OLAP, reporting and advanced analytics from a single data platform.

In our observation, some of the existing definitions for lakehouses (cf. section "Concepts and Definitions for Lakehouses") follow a *bottom-up approach*, where the desired characteristics of such a data platform are specified first and then a definition is derived from them. However, this leads to the issue that it remains an open question which overarching goal a lakehouse should actually serve and what benefits it provides for enterprises in comparison to existing data platforms and architectures. Accordingly, it is also unclear why the specified characteristics are important and which of them are mandatory. In order to address this issue, we instead applied a *top-down approach*, which is illustrated in Fig. 5. Here, we first specify the overarching goal and the benefits that lakehouses in our opinion provide to enterprises. Subsequently, we determine the implications of this goal for the functional and architectural aspects of data platforms and derive a corresponding definition. Finally, we use this definition to identify technical requirements that can be used to design lakehouses and to evaluate whether given data platforms constitute lakehouses that comply with our definition. By applying this method, we can first shift the focus to a higher level of abstraction and later derive individual characteristics.

Despite the different existing perspectives on the lakehouse paradigm, it appears to be one of the fundamental motivations for this type of data platform to *simplify existing enterprise analytics architectures* by reducing their complexity. We consider this the overarching goal of lakehouses, as they will not be able to functionally supersede both data warehouses and data lakes in their respective analytical domains at the same type due to their contrary characteristics. However, they can provide additional benefit for enterprises in comparison to the more traditional data platforms by keeping the overall enterprise analytics architecture as simple as possible and hence avoiding a high number of involved data platforms, data pipelines and other infrastructural components that lead to high development and maintenance efforts and increased operational costs. Similarly, they can also contribute to simplify the daily work of data analysts and data scientists by providing a single access point to the data and enabling users to transparently access data through uniform interfaces, e.g. in terms of using the same data formats, query languages and dialects, for different types of analytical workloads. With this goal in mind, especially new companies and startups can benefit from lakehouses, as they no longer need to develop and maintain complex and expensive infrastructures for performing ordinary analytical tasks. However, also larger enterprises can benefit from consolidating data platforms within their departments, as it may allow to save costs and give analysts more intuitive access to data.

From our perspective, this goal of simplifying enterprise analytics architectures leads to three implications: First of all, such a data platform must show an inherently *low heterogeneity* in terms of the utilized technologies and data formats, as a high number of different components tends to increase the overall complexity as they typically support different data types, data structures, interfaces, and tend to differ in many other characteristics as they have been designed with different use cases in mind. Secondly, the data platform should *avoid data movement between different storage systems and formats*, as these require the development and maintenance of data pipelines and may become prone to errors if the source and target systems differ strongly in their characteristics and supported features. For example, when developing such a data pipeline that transfers data between different types of storage systems, aspects such as exactly-once semantics, eventual consistency, internal representations of data types, and the additional computational load utilization must be considered, which can easily lead to undesirable behavior and the loss or inconsistency of data. Similarly, also transformations between different data formats, such as CSV, JSON, Apache Parquet and Apache ORC should be avoided, as these formats may support different primitive and complex data types and transformations can therefore also be a source of errors. Furthermore, dealing with different data formats means additional difficulties for data analysts and data scientists, as they may have to perform transformations ad-hoc in order to be able to consolidate and analyze data together. Finally, lakehouses should constitute a *single source of truth*, so that data analysts and data scientists have a single point of access with uniform interfaces for the data they need, rather than having to choose between different sources that contain similar but possibly not identical data due to differences in terms of data formats, domain of the data and the time windows in which the data was collected. A single source of truth, however, increases confidence in the stored data and in the analysis results, contributing to simplify the architecture and the analysis processes. Based on these considerations, we propose the following definition:

Definition 1 A lakehouse is an integrated data platform that leverages the same storage type and data format for reporting and OLAP, data mining and machine learning, as well as streaming workloads.

Reporting and OLAP refer to the primary workloads of data warehouses, while the combination of data mining and machine learning, as well as streaming constitute advanced analytics and hence represent typical data lake workloads.

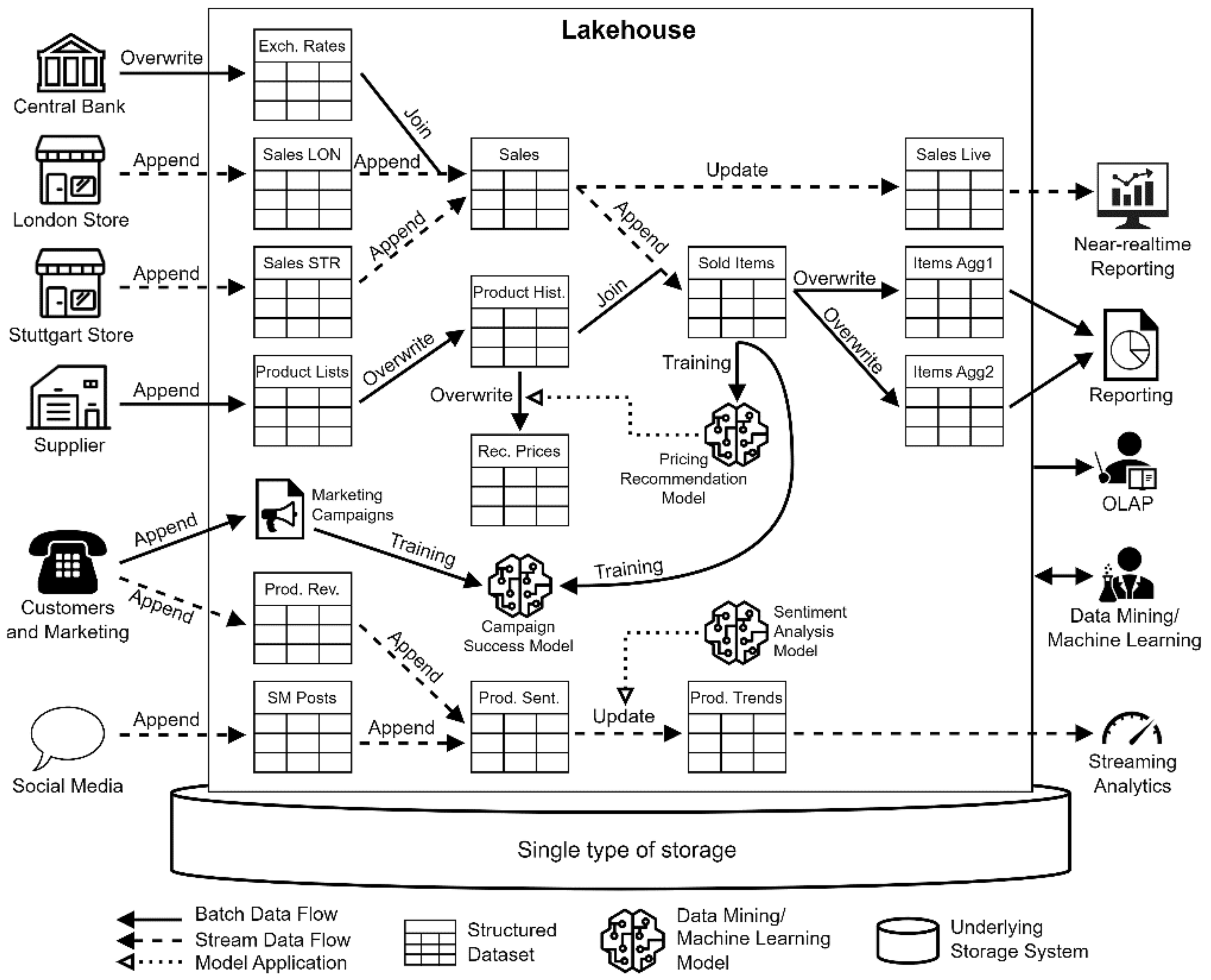

Fig. 6 Lakehouse-based implementation of the retail scenario

All three workloads are discussed in detail in the section "Analytical Workload Characteristics" and subsequently used to derive the technical requirements. With its additional constraints, the definition ensures that lakehouses use the same type of storage, e.g. object storages, and the same data format, e.g. Apache Parquet, to serve all of the listed workloads. As a consequence, the data must not be replicated to different types of storages or transformed into other formats for these purposes. This reflects the previously described implications for reducing heterogeneity, avoiding additional data movement and data transformation steps and the formation of a single source of truth. Consequently, we argue that only the integrated architecture (cf. section "Data Warehouses and Data Lakes") allows the construction of lakehouses, since the other patterns utilize multiple types of storages, which leads to additional data movement and transformation steps between the data platforms and/or impedes the formation a single source of truth. The section "Lakehouse-Based Implementation of the Retail Scenario" illustrates the reasoning and benefits of this definition with the help of the retail scenario, which is then implemented on a lakehouse that complies with this definition. The definition deliberately does not make any statements about nonfunctional properties of lakehouses, since these mainly distinguish more suitable from less suitable lakehouse systems for the respective application scenario at hand, but have no major influence on the general characteristics of the data platform.

Lakehouse-Based Implementation of the Retail Scenario

In the section "Scenario Outline", a representative retail scenario was introduced, for which the section "Baseline

Implementation with Data Warehouse and Data Lake" already described a baseline implementation that leveraged a data warehouse and a data lake as underlying data platform. However, while being representative for many established enterprise analytics architectures of the real world, this implementation suffered from several problems and challenges, which were discussed in the section "Challenges of Data Lakes". In Fig. 6, another possible implementation of the retail scenario is depicted, which now utilizes a lakehouse as data platform that complies with the previously introduced definition and its underlying goals. Especially due to the challenges described in the section "Challenges of Data Lakes", this implementation is not feasible out-of-the-box with many conventional technologies and concepts and would require the application of more recently emerged technologies. However, at this point, no specific technologies are discussed, as the lakehouse-based implementation of this section is supposed to illustrate the reasoning behind our lakehouse definition and its impact on enterprise analytics architectures if such a lakehouse is leveraged instead of traditional data warehouses or data lakes. Accordingly, the depicted implementation abstracts from specific technologies and instead reflects the essence of our lakehouse definition in a technology-agnostic manner. The individual technical requirements that must be fulfilled by a data platform in order to constitute a lakehouse and hence to enable the practical implementation of such an architecture are identified and described in the section "Derived Technical Requirements".

In accordance with our definition, the lakehouse in the implementation that is shown in Fig. 6 is an integrated data platform that leverages only one single type of storage system and one single data format to serve all three different types of analytical workloads *OLAP and reporting* (cf. U1, U3), *data mining and machine learning* (cf. U4, U6, U7) and *streaming* (cf. U2, U5). In principle, any type of storage system and any data format can be used for this purpose, although some appear to be more suitable, as discussed in the following sections of this paper. As a consequence of these constraints, it is no longer allowed for the architecture to use different storage systems (e.g. object storages in combination with relational databases) or different data formats (e.g. CSV and Apache Parquet) for different types of data. Therefore, all data, including structured data like datasets with a table-like-structure, as well as semi-structured and unstructured data like documents and images, must be stored on the same type of storage system and in the same data format. Accordingly, in the implementation, not only the datasets for currency exchange rates, the sales data of the retail stores and the product lists of the supplier, which all represent structured data, but also the unstructured data about the conducted marketing campaigns and the generated machine learning models, which are often described by semi-structured data, are stored on the same storage system, which could be a file system or an object storage. This also applies to all downstream datasets that are produced in the course of the processing pipeline, such as the datasets *Sales Live* and *Items Agg1* that contain pre-aggregated data.

In addition, the data must also use the same type of storage regardless of whether it is processed as batch or stream and hence batch and stream processing must be combined on a shared storage system. When technically supported, this constraint greatly reduces the complexity of the overall architecture by avoiding a Lambda architecture with separate batch and speed layers. In addition, it urges for a unification that allows to use the same datasets selectively for both batch and stream processing, since it would be inefficient to store the same data in the same format on the same storage system twice. In the lakehouse-based implementation, the impact of this unification in comparison to the former implementation is particularly noticeable with respect to the sales data, which was previously stored and processed redundantly on the stream and batch layers. Instead, this data is now primarily processed via stream processing and either supplemented with batch data (e.g. the currency exchange rates) or partially processed with batch processing (e.g. the *Sold Items* dataset) where necessary. This combination of batch and stream processing also allows to process most of the data of the lakehouse-based implementation incrementally as a stream (i.e. with append and update operations), avoiding entire re-computations (i.e. with overwrite operations) using batch processing in many cases. However, if necessary, re-computations on datasets should still be supported, for example for particular complex aggregations, as they are required in the lakehouse-based implementation to derive the datasets *Items Agg1* and *Items Agg2* from the *Sold Items* data, or for the training of machine learning models, such as the *Price Recommendation Model* and the *Campaign Success Model*. Due to this combination and mutual complementation of batch and stream processing, the shown implementation does not leverage any of the traditional processing architectures, such as the Lambda or Kappa architecture. Instead, this approach with a shared storage for batch and stream processing is commonly referred to as *Delta architecture* and requires specific technologies and concepts for its implementation, which are discussed in more detail in the following sections. According to our definition, the lakehouse must be able to serve reporting and OLAP workloads, which, among other aspects, require that the datasets located on the lakehouse can be queried and analyzed interactively with the help of query languages, similar to tables in relational databases. For data mining and machine learning, it must provide direct read access to the stored datasets so that data scientists can exploit them directly on the underlying storage system with the help of appropriate tools and algorithms, without needing to export the data to a separate storage system first, as this would contradict the lakehouse definition and result in additional data movement and transformation steps. For streaming workloads, such as near-realtime reporting (cf. U2) and streaming analytics (cf. U5), it is necessary that stream processing engines can efficiently read and write the stored data of the lakehouse and detect changes in the individual datasets timely so that they can then further process the data and, for example, update dashboards and prepare analyses results.

In order to be able to use a uniform data format like Apache Parquet throughout the lakehouse as demanded by our lakehouse definition, it first may be necessary to convert this data to the target format during ingest. This conversion can be subject to errors, since the source data may use different data types than supported by the target data format, but having a uniform data format on the lakehouse still avoids all further conversions during the remaining processing and analysis process and therefore contributes to reducing the overall complexity and error-proneness of the architecture. Transforming the data to the target format of the lakehouse is usually rather straightforward for structured and hence table-like data, such as the sales data of both retail stores or the currency exchange data, as the source and target format typically differ only slightly in their supported data structures and data formats. For semi-structured and unstructured data this appears to be more challenging, as it typically either imposes more complex and nested data structures, such as found in XML or JSON documents, or comes in its very own data formats, such as binary image formats. However, this data still does not need to be stored separately or in a different format than the structured data, as it can commonly be embedded into the uniform data format as well: In the case of semi-structured data, such as XML or JSON documents, this data can be integrated as a sequence of characters (strings), while unstructured or binary data can be included as byte arrays, which are commonly referred to as binary large objects (BLOBs). Both data types are commonly supported by popular data formats such as Apache Parquet [69] or Apache ORC [70] and allow to embed the rather unhandy data into table-like structured datasets. This approach possesses some advantages over storing semi-structured and unstructured data as separate files and in separate formats, since it allows to: (a) store the actual data together with related metadata, such as recording time, file size and relevant context information, (b) query the data based on the more easily processable metadata, (c) to process and manage semi-structured and unstructured data similar to and along with structured data and (d) to query semi-structured data, such as strings of XML or JSON documents, with special query languages and extensions like XQuery[12] and Databricks' JSON operators[13] if supported by the utilized query engines.

However, also drawbacks may arise, such as a loss of read and write performance, which depends on the size of the data and the optimizations of the technology stack.

When comparing the lakehouse-based implementation of Fig. 6 with the baseline implementation as presented in the section "Baseline Implementation with Data Warehouse and Data Lake" that leveraged a separate data warehouse and a data lake, it becomes obvious that the constraints in terms of a single type of storage system and a single data format inevitably lead to a simplification of the architecture, as a lower number of data movement and data transformation steps is required. For the retail scenario, the introduction of a lakehouse as data platform reduced the number of involved types of storage systems from four to one and almost halved the number of datasets that need to be stored and managed. As a result, only one storage system and a lower number of data pipelines need to be maintained, which is likely to reduce the overall development and operational costs. Furthermore, the data does not need to replicated anymore between different systems, which leads to a single data access point for data analysts and data scientists and contributes to the formation of a single source of truth, as users do not need to choose between different data platforms for consuming trustworthy data. As the lakehouse allows to query the data for reporting and OLAP, but also provides direct access to this data for machine learning and data mining applications, data scientists can work on up-to-date data and do not need to rely on possibly slow pipelines for exporting the data. However, the lakehouse implementation is also associated with potential drawbacks: Since all data is necessarily stored on a single type of storage system regardless of its type, it can no longer be stored and managed on the systems that are strongly optimized towards this specific type of data, such as document stores or time-series databases. This may lead to a reduced query performance and a less extensive and convenient set of analysis features that can be offered to the data analysts. For this reason, a lakehouse-based implementation may not be equally suitable for all kinds of application scenarios and enterprises.

Analytical Workload Characteristics

Our lakehouse definition (cf. section "Defining the Lakehouse") demands that a lakehouse must in particular be able to serve the three types of analytical workloads *OLAP and reporting, data mining and machine learning*, as well as *streaming*. Based on the properties of these workloads and

12 https://www.w3.org/TR/xquery/.

13 https://docs.databricks.com/en/optimizations/semi-structured.html.

Table 5 Comparison of the analytical workloads that lakehouses must be able to serve

Characteristics	Reporting and OLAP	Data mining and machine learning	Streaming
Analytics types	Descriptive, diagnostic	Diagnostic, predictive, prescriptive	Descriptive, diagnostic, predictive
Users	Business users, data analysts	Data scientists	Data analysts, data scientists
Data access	Via query language	Direct access on storage	On stream storage
Processing type	Batch	Batch	Streaming
Response time	Interactive	No strict requirements	Near real-time
Type of data	Structured	All types	All types
Granularity of data	Aggregated	Raw, aggregated	Raw, aggregated
Processing complexity	High	High	Low
Concurrency	High	Low	High
Example use cases	U1, U3	U4, U6, U7	U2, U5

the other implications of our definition that were discussed in the previous sections, several technical requirements for lakehouses can be derived. This section lays the groundwork for the derivation of these requirements by characterizing the three different types of analytical workloads. Table 5 summarizes and compares the results.

Reporting and OLAP

Reporting refers to the production, delivery and management of reports [71], i.e. static or interactive overviews of business facts, such as key performance indicators (KPIs) and corresponding visualizations [72]. For the automatic generation of reports, predefined queries are typically employed and periodically executed against the stored data [71]. In the retail scenario, this workload is reflected by the use case U1, for which different reports with tabular and visual overviews on the sales of the retail chain have to be generated from the collected data on a monthly basis.

This workload is supplemented by Online Analytical Processing (OLAP), which intends to enable interactive analyses by providing fast, intuitive and scalable multi-user query capabilities based on multidimensional data models [9, 73]. In the use case U3 of the retail scenario, OLAP is applied in order to analyze the data occasionally with respect to complex interrelationships that are not permanent of interest, such as the impacts of external events like Christmas on the sales of the retail chain. As analytical workloads, OLAP and reporting share many characteristics: In both cases, data analysts usually employ query languages in order to formulate complex queries that can then be executed on the stored data in order to answer descriptive and diagnostic analyses questions, such as how big last month's revenue at each of the two retail stores was and why the revenues decreased in comparison to the previous month. Furthermore, for the sake of higher query performance, OLAP and reporting both typically operate on pre-processed and pre-aggregated batches of data rather than on raw data and cannot work well on data streams due to the general high complexity of the required analyses and aggregations. These workloads are also commonly associated with a high concurrency, as multiple data analysts may conduct OLAP in parallel to each other and to the periodically executed report generation, while the datasets are also concurrently updated by processing tasks. However, despite the previously discussed similarities, OLAP and reporting differ in terms of timing requirements: While the automated generation of reports happens periodically and is therefore non-time-critical and allowed to consume several hours or even multiple days, OLAP is supposed to be interactive and hence often expected to process even complex queries in at most 20 s [73].

Data Mining and Machine Learning

Both data mining and machine learning are broad sub-disciplines of the field of advanced analytics [13]. Data mining is the process of discovering patterns and other forms of knowledge in large datasets [74]. Typical data mining techniques include classification approaches, clustering analysis and regression analysis, for which parametrized algorithms are applied to the data in order to generate models that reflect the discovered interrelationships within the data. In the retail scenario, the use case U4 requires the application of data mining techniques in order to detect patterns regarding the buying behavior of the customers in the stored data, e.g. by performing an association analysis on products that are bought together. The results of this analysis could later be used to recommend products to customers that they may want to buy as well.

The goal of machine learning is the development and application of learning algorithms that are capable of building models from the stored data which can subsequently be used to generate predictions on new observations [75]. Although there is an overlap between the techniques used in data mining and machine learning, the focus of data mining lies on finding new patterns and inferring knowledge, while

Table 6 Overview of the identified technical requirements for lakehouses, the factors by which they were mainly influenced and the challenges that are addressed by them

Requirement		Influencing factors				Impact
#	Name	Reporting and OLAP	DM/ML	Streaming	Data management	Addressed challenges
R1	Same storage type and data format	◐	◐	○	●	C1
R2	CRUD for all types of data	◐	◐	◐	●	C3
R3	Relational data collections	●	◐	◐	◐	C2
R4	Query language	●	○	○	○	C1
R5	Consistency guarantees	●	◐	◐	◐	C4
R6	Isolation and atomicity	●	○	●	●	C5
R7	Direct read access	○	●	○	◐	C1
R8	Unified batch and stream processing	◐	○	●	●	C2, C3

● strong influence, ◐ medium influence, ○ low influence

machine learning tries to generalize from patterns in the collected data in order to generate prediction models for unseen data. The use cases U6 and U7 of the retail scenario require data scientists to iteratively train, test and optimize machine learning models that are capable of predicting the success of marketing campaigns and recommending optimal prices for the products that are supposed to be sold by the retail stores. Since most data mining and machine learning algorithms are too complex to be expressed in query languages and also due to large data volumes, data scientists and their tools usually require direct read access to the data on the infrastructure on which it is stored, for example by using scalable and possibly cluster-based processing engines, such as Apache Spark with its MLlib[14] module. Due to the high complexity of the involved algorithms, the generation of data mining and machine learning models is typically conducted via batch processing and not via stream processing, as the latter does not allow to randomly access the available data and is hence limited to specific incremental algorithms. Furthermore, data scientists tend to work on both raw and aggregated data, as analysis questions for data mining are often not known in advance and may only arise after the discovery for initial patterns in the collected data. Similarly, it is often not known in advance which aspects of the data qualify as features for machine learning models. Therefore, the data cannot always be appropriately pre-processed in advance to support data mining and machine learning tasks. In addition, the data that is supposed to be analyzed can be of arbitrary types, including semi-structured and even unstructured data [76], as data mining and machine learning techniques are not limited to structured data and can also be applied to files such as documents, images or audio files. Since the generation and optimization of models is typically an iterative process that involves a lot of human experimentation and produces results that are valid until an updated version of the model is created, there are no strict timing requirements associated with this workload.

Streaming

In the context of analytical workloads, streaming subsumes all analysis techniques for near-realtime reporting and stream analytics [77]. The goals of near-realtime reporting are similar to those of batch reporting, with the difference that the reports are usually replaced by dynamic dashboards whose business facts and visualizations must be updated within minutes. Due to the time-consuming nature of most data-driven analysis techniques, the calculated results, such as KPIs and further aggregations and overviews, cannot be continuously re-calculated and instead must be incrementally updated by stream processing engines as new data arrives. Hence, near-realtime reporting requires different techniques than batch reporting. In the retail scenario, the use case U2 corresponds to reporting in near-realtime, as the live dashboard is supposed to be updated as timely as possible with new sales and revenue data from the two retail stores.

Stream analytics refers to techniques for the continuous analysis of data that arrives in unbounded data streams, including algorithms for data filtering, pattern detecting and clustering [77], but also the application of trained machine learning models to the incoming data in order to generate predictions. Accordingly, the use case U5 of the retail scenario can be considered as stream analytics, which copes with detection of product-related trends by analyzing product reviews and social media posts. If trends are identified with regard to a particular product, business users could be automatically notified about the developments in a timely manner, which could result in the adjustment of currently running marketing campaigns, for example.

Since streaming needs to process data incrementally and hence requires the processing engines to hold intermediate

[14] https://spark.apache.org/mllib/.

results of aggregations and calculations as internal state, it is more suitable for simple analytics tasks rather than complex computations that require random access to data or produce large intermediate results.

Similar to reporting and OLAP, streaming is most commonly employed in order to answer analytics questions that are already known in advance and hence typically operates on pre-processed and pre-aggregated data, which allows to speed-up the analyses. Additionally, since streaming needs to process data with a high frequency while the underlying datasets are updated in parallel, a high concurrency can be assumed.

Derived Technical Requirements

Our definition (cf. section "Defining the Lakehouse") imposes several constraints on lakehouses regarding the utilized storage systems and data formats and also demands that lakehouses must be able to serve all three of the previously characterized analytical workloads. This allows to derive several technical requirements that data platforms must fulfill in order to comply with our lakehouse definition and hence enable the construction of architectures such as the one that is illustrated in Fig. 6. In total, we were able to identify eight such technical requirements for lakehouses, which are summarized in Table 6.

All of these requirements were primarily derived from the characteristics and demands of the three types of analytical workloads (cf. section "Analytical Workload Characteristics"), but also from the general expectation on data platforms to enable the management of data for analytical purposes. This covers several aspects such as data ingestion, storage, processing, maintenance and the provisioning of data to analytical applications. Table 6 outlines the degree to which the different workloads and data management aspects influenced the selection of each requirement. In addition, the last column indicates which of the challenges that were described in the section "Challenges of Data Lakes" can be addressed by a data platform that fulfills the corresponding requirement. The individual requirements have been slightly updated and clarified in comparison to our previous work (cf. [8]) and are described in detail below.

R1: Same Storage Type and Data Format

Following up on our lakehouse definition, this requirement demands that all data and all technical metadata is solely stored on a single type of highly scalable storage and that all data (excluding metadata) is stored using the same data format. Examples for storage types include object storages, such as provided by Amazon S3 or Azure Blob Storages[15], but also distributed file system like the HDFS or relational databases. In contrast, Apache Parquet, Apache ORC, CSV and JSON are examples for data formats.

In the context of this and the following requirements, *technical metadata* refers to the metadata that is required from a technical perspective in order to operate the data platform and to serve the three analytical workloads (cf. section "Analytical Workload Characteristics"). For example, metadata that describes the structure of the data (i.e. schema, columns, data types) and is hence required by processing engines in order to perform read and write operations on the data platform, can be considered as technical metadata. In contrast, operational or business metadata, such as descriptions of the stored data from a business perspective or lineage data that serves purely informational purposes, does not fall into this category. Therefore, it is still valid in terms of this requirement to employ supplementary metadata-based data platforms in addition to the lakehouse, as long as the data that is managed on these platforms is not relevant for the technical operability of the lakehouse itself. Examples for such metadata-based data platforms include *data catalogs* [78], which store and integrate metadata from different systems and hence can give an overview of the data that is available within an enterprise, as well as *enterprise data marketplaces* [79], which pursue to simplify and standardize the exchange of data within an enterprise and hence support data democratization.

As discussed in the section "Defining the Lakehouse", these constraints regarding the storage systems and data formats intend to avoid the construction of complex, error-prone and costly enterprise analytics architectures. Accordingly, R1 does not allow to replicate data or technical metadata to different storage types (e.g. from object storage to relational database) or to transform the data to different data formats (e.g. from Apache Parquet to CSV), as these tasks are commonly associated with complex and error-prone conversions and impede the formation of a single source of truth that provides uniform access to the required data. However, the parallel utilization of multiple storage systems of the same type, e.g. several object storages from different cloud providers, is not restricted, as conversions between different storage systems of the same type are usually easier to accomplish and since such hybrid architectures may even be required for ensuring compliance with enterprise policies or for achieving high availability. Furthermore, the data may be replicated and stored in different versions, different levels of granularity and different schemas on the same type of storage, which allows the implementation of data processing pipelines analogously to the ones that can be seen in Fig. 6.

[15] https://azure.microsoft.com/products/storage/blobs/.

While all data that is stored in the lakehouse must leverage the same data format, metadata is allowed to be stored in different formats, as it typically is of a lower volume than the actual data, serves a different purpose and must also be managed and processed differently.

It is worth noting that for the reasons that were discussed in the section "Lakehouse-Based Implementation of the Retail Scenario", the constraint regarding a uniform data format also applies to semi-structured and unstructured data, which can usually be directly embedded into the data formats for structured data.

While R1 is mainly driven by the constraints that are already formulated in the lakehouse definition, it also helps to simplify the data management, as the data cannot is no longer allowed to be spread across different types of storage systems and stored in multiple different formats that complicate all kinds of data-related tasks. Instead, it contributes to the formation of a single source of truth on which the available data can be accessed in a uniform manner. This way, R1 also supports exploratory analyses, such as in the scope of OLAP, data mining and machine learning, since data analysts and data scientists do not need to gather their required data across various sources. With the goal of simplifying enterprise analytics architectures and avoiding a high heterogeneity, R1 particularly addresses the challenge C1 (cf. section "Challenges of Data Lakes").

R2: CRUD for All Types of Data

As it is generally expected from all types of data platforms, lakehouses must be able to store, manage and provision data, such that it can be collected, processed and exploited for analytical purposes. This includes the ability of ingesting and storing new data on the data platform ("C" of CRUD), as well as the possibility to retrieve ("R" of CRUD) this stored data again in order to process and analyze it. While these features are already sufficient for rudimentary data platforms and workloads, it may also be necessary to update (U) or delete (D) existing data in the scope of data management tasks, for example if certain entries have to be obscured or removed due to new data privacy policies within the enterprise. Furthermore, it allows to correct errors in the data, which may have arisen due to incorrectly implemented processing tasks, for example. As described for C3, many current data platforms are based on processing models that do not allow such updates or deletes at the record level and thus not only complicate data management, but also prevent incremental batch processing and thus promote resource-intensive, expensive and slow processing paradigms. To address these issues, R2 requires that a lakehouse must be able to ingest (C), update (U), delete (D), and retrieve (R) data on the record-level. Furthermore, since data mining, machine learning and streaming are not limited to structured data, but may also need to operate on semi-structured and unstructured data, CRUD must be supported for all kinds of data. Besides data management, the possibilities of ingesting, retrieving and modifying can be considered as relevant foundation for all kinds of analytical workloads.

R3: Relational Data Collections

Many prevalent data platforms employ a processing model in which large datasets are broken down to multiple files that are then stored in certain file formats, such as Apache Parquet, CSV or even proprietary ones, on the underlying storage system. For example, in the lakehouse-based implementation, the dataset *Sales* only constitutes a self-contained entity on the logical level and may consist of several physical data files on the underlying storage system that contain the actual sales records. Among other benefits, such as the achievement of immutability for data files and hence a greater robustness for data processing (cf. section "Challenges of Data Lakes"), this also enables the application of partitioning strategies [80] and thus allows more efficient query processing. However, the splitting of datasets into several smaller files must be complemented by abstraction techniques, which allow to present the several data files as a cohesive collection of data to the users. This is necessary because users usually do not want to deal manually with a high number of data files, but instead prefer to interact with the entire dataset as a whole on a logical level. For example, aggregations are generally supposed to be performed on the entire dataset and not just of a small section of it that is represented by one single data file. Processing engines like Apache Spark or Apache Flink typically already provide such data abstractions and hide the underlying complexity and storage strategies from the users. Similarly, R3 requires that lakehouses must provide concepts that allow to compose data to relational data collections on the logical level, such that multiple physical data files on the storage system can jointly represent a cohesive data collection with relational properties [81], including a table-like structure with columns and rows. This way and in conjunction with R2, table-lake semantics for datasets are gained, which are comparable to tables as known from relational databases, albeit with a possibly smaller range of features. This can be achieved in various ways, e.g. by storing and managing technical metadata that tracks information about the available data collections, their structure and the associated data files holding the actual data. The relational abstraction of the datasets is particularly important in order to support reporting and OLAP workloads, as they generally rely on the processing of relational data and require a higher degree of structure than would be provided by other types of abstractions, such as graphs or documents. Besides reporting and OLAP, also other analytical workloads and general data management tasks can benefit from relational data collections, as they simplify the handling and addressing of data sources and sinks, for example by applying naming schemas. Additionally, they provide a basic level of structure that simplifies the generation of data mining and machine learning models from the stored data. As the relational data collections are also supposed to serve as sinks and sources for batch and stream processing (cf. R8), they help to address the challenge C2 by providing a uniform abstraction for batch and stream data.

R4: Query Language

In order to support reporting and OLAP tasks, a lakehouse must at least offer a declarative, structured data query language (DQL) that allows to query the available data collections (cf. R3) in a relational manner. Such a language is necessary, because queries in the scope of OLAP often need to be created in an experimental manner and with high frequency in order to allow data analysts to explore the available data interactively. Although additional language elements, such as those of a data management language (DML) are generally desirable as well, these are not mandatory, since the associated operations could also be issued in other ways, such as via a dedicated API. Besides OLAP, a query language is also helpful for specifying the business facts and aggregations that are supposed to be included and visualized within reports. In the lakehouse-based implementation of the retail scenario (cf. Fig. 6), the query language is especially helpful for the periodic generation of reports from the data collections *Items Agg1* and *Items Agg2*, which contain aggregated data about the sold products. Furthermore, data analysts can use it for querying any of the stored datasets in order to analyze the data via OLAP. While the data warehouse and the data lake of the original implementation (cf. section "Baseline Implementation with Data Warehouse and Data Lake") may offer rather different query capabilities that are based on different engines and dialects and may not allow to join data across both platforms, the lakehouse offers at least one uniform query language that can be used to query all available data. This contributes to the simplification of the enterprise analytics architecture and thus also addresses the challenge C1, at least to some limited extent.

R5: Consistency Guarantees

As discussed for the challenge C4, many data platforms, such as rudimentary data lakes, do not provide inherent capabilities for ensuring the internal consistency of the stored data with respect to its structure and pre-defined constraints. Accordingly, when reading and processing the stored data, it cannot be implicitly assumed that the data is actually available in the expected schema and that all expected attributes are present in the individual data records and possess the expected data types. As a consequence, huge amounts of boilerplate code are necessary in each processing step in order to validate the data and to avoid processing errors and possibly incorrect analyses result. In this context, a lack of consistency guarantees is especially problematic for reporting and OLAP, as many query languages are not suitable for handling erroneous or inconsistent data. Furthermore, aggregations and filtering operations cannot be performed in a meaningful manner if the underlying data does not meet the original expectations. For example, if the retail store in Stuttgart accidentally transmits its sales values in cents instead of euros to the data platform for a short period of time, this error could not be detected automatically without corresponding validation logic in the processing tasks and would lead to incorrect results in the generated reports and OLAP queries. Similarly, the supplier could suddenly change the format of the product identifiers that are submitted to the data platform as part of the product lists, which, if undetected, would lead to inconsistencies and wrong join results in the *Sold Items* and all downstream data collections. With the help of consistency guarantees, which are defined and enforced at the level of the respective data collections, such problems can be detected and avoided, e.g. by rejecting erroneous during write operations or by ignoring the affected records for subsequent read accesses. While reporting and OLAP are particularly susceptible to consistency problems due to their reliance on query languages, the resulting lack of flexibility for error handling and the expectation to provide absolutely trustworthy results, also data mining, machine learning and streaming workloads can benefit from consistency guarantees, since they reduce the required amount of additional validation logic and increase the general robustness of the analyses. Analogously, also data management tasks, such as in data processing pipelines, can be simplified. However, consistency guarantees may not be required and even not be desired for all data collections of the data platform. For example, it may be reasonable to store the data that is provided by the data sources in data collections that accept all data regardless of its validity and do not reject erroneous data. This way, it is possible to track and reconstruct which data was provided by the data sources at which point in time, while the consistency of the data can still be verified and ensures in downstream data collections.

Based on the previously discussed aspects, R5 demands that a lakehouse must provide means for checking and enforcing the consistency of data across data collections with respect to the structure and content of data records. For example, this can be achieved by validating new or changed data records against a previously defined schema. In order to avoid hard-to-maintain and error-prone boilerplate code, it is important that the schema must be defined at the level of the data collections itself and that data is then implicitly checked against this schema, rather than requiring the schema to be specified and explicitly checked in the logic of every individual processing step. However, it is up to the implementation of the respective lakehouse to decide whether the consistency should be enforced when data is written to a data collection ("on write") or when the data is queried ("on read").

R6: Isolation and Atomicity

Since a lakehouse is supposed to serve different types of workloads in parallel to possibly multiple consumers and may employ comprehensive data processing pipelines, a comparable high degree of concurrency can be expected for operations that are performed on the individual data collections (cf. R3). Without additional precautions and an explicit assurance of isolation and atomicity for these operations, concurrency anomalies and inconsistencies in the data may occur, as described by the challenge C5. These issues particularly affect the analytical workloads that show a high level of concurrency, such as reporting, OLAP and streaming, as well as all steps of the data processing pipeline that share access to the same data collections. For example, in the lakehouse-based implementation (cf. Fig. 6), the arriving sales data is continuously appended to the data collection *Sold Items*, which is periodically accessed by batch processing tasks in order to generate the aggregations Items Agg1 and Items Agg2 and also by two other batch processing tasks that are responsible for the generation of the *Price Recommendation* and *Campaign Success* machine learning models. Without isolation and atomicity, the batch processing tasks could possibly see intermediate results of the stream data that is ingested into the *Sold Items* data collection, while *Items Agg1* and *Items Agg2* could be left in an inconsistent state if the batch processing tasks fail unexpectedly. With the goal to avoid such issues, to support the proper operation of the analytical workloads and to increase the overall robustness, R6 requires that a lakehouse must ensure isolation and atomicity [10] for all operations that modify or access the data of data collections concurrently. This can be implemented in various ways, such as by applying locking mechanisms or multi-version concurrency control.

R7: Direct Read Access

As discussed in the section "Analytical Workload Characteristics", data mining and machine learning tasks typically require direct access to the data on the storage layer, as data mining and machine learning models cannot be efficiently generated from large volumes of data via general data query languages that are offered by the data platform. Due to R1, it is also not possible to export the data to another platform beforehand, as this would require the operation and maintenance of additional storage systems and data pipelines for exporting the data. Furthermore, this approach would result in the replication of data and possibly also in less up-to-date models, as the data has first to be extracted, transformed and transmitted before it can be exploited for data mining and machine learning. Therefore, in order to avoid these issues, data scientists should be able to directly access the data on the storage system of the data platform with their preferred tools and libraries, such as MLlib, TensorFlow[16] and scikit-learn[17], without needing to export the data first. Since technical metadata may contain important information about the structure of the stored data, e.g. in terms of partitions, the locations of data files and the composition of data collections, and may be necessary in order to ensure atomicity and isolation (cf. R6), also direct access to the technical metadata must be granted. Consequently, R7 demands that lakehouses must provide unmediated and random read access to all stored data and technical metadata and leverage open, standardized file formats, so that the data and technical metadata can be accessed directly on the underlying storage system without needing to export and transform it first. While this requirement facilitates data mining and machine learning, it also opens the lakehouse to other technologies, such as different query engines and batch and stream processing engines, as they also have the opportunity to access and read the data, as long as they are able to interpret the technical metadata and locate and read the data files that constitute the data collections. Hence, R7 also supports the data management with respect to the data consumption aspect. Furthermore, it helps to address C1, as it avoids the export of data to other storage systems.

R8: Unified Batch and Stream Processing

As part of the data management, a lakehouse must support the processing of data, where each processing step includes: (a) the reading of *source data* that is either provided by data sources and supposed to be ingested into the lakehouse or already available on the platform as a data collection, (b) the transformation of this data according to specific application logic by a processing engine and (c) either the provisioning of the *processing results* to analytics applications or their storage on the lakehouse as part of another data collection. Since relational data collections represent the fundamental abstraction for datasets in lakehouses, it is necessary that they can be used as sources and sinks for the individual processing steps. By combining multiple of such steps, complex data processing pipelines according to the pipes-and-filters architectural pattern can be constructed, which materialize their intermediate results on the data platform after each

16 https://www.tensorflow.org.

17 https://scikit-learn.org.

step, so that they can be used as source data by different downstream processing steps. Within the scope of such a pipeline, the raw data as provided by the data sources is gradually processed and enriched by every step, until it has reached sufficient quality and granularity so that it can consumed by analytics applications. For example, in the lakehouse-based implementation of the retail scenario (cf. Fig. 6), the sales data of the two retail stores is first ingested and stored in separate data collections, then merged into a common data collection *Sales*, at one branch enriched by the product data from the supplier, aggregated and finally provided to a dashboard and reporting applications.

The data processing can generally be performed in one of two ways. In batch processing, the source data is bounded and completely available for random access to the processing engine, while in stream processing, the source data is represented by an unbounded and continuously updated stream of data [52], which allows the processing engine only to hold limited amount of the source data (e.g. the ten most recent records or the current maximum value of a certain field) as internal state and otherwise relies on incremental processing, as a complete view on all data is not possible. Although the two paradigms differ strongly in their semantics, current processing engines such as Apache Spark and Apache Flink attempt to unify these approaches by providing similar APIs for both processing types and by converting the user-defined application logic to incremental processing instructions in the case of stream processing. This simplifies the development of processing tasks and avoids the need to maintain two separate code bases with similar application logic. However, due to the differences in the availability of the source data and the resulting implications for the processing semantics, none of the two processing paradigms can generally replace the other for all scenarios. Consequently, data platforms ideally should support both batch and stream processing.

R1, as well as the underlying goal of our lakehouse definition to reduce the complexity of enterprise analytics architectures, both require that the same type of storage system is used for batch and stream processing. However, as discussed for the challenge C2 and the data lake of the baseline implementation (cf. Fig. 3), this is not straightforward to achieve, because many stream processing engines cannot efficiently read small portions of data from and write small portions of data to batch storage with high frequency out-of-the-box. Hence, a lakehouse must take measures that enable the efficient usage of the same type of storage system for batch and stream processing, for example by applying techniques for write compaction and exactly-once-semantics, as well as by implementing additional concepts that allow processing engines to efficiently find the newly available data records [25]. The lakehouse-based implementation of the retail scenario (cf. Fig. 6) shows that operationally expensive architectural patterns, such as the Lambda architecture, can be avoided and the complexity of the architecture further reduced when it is possible to combine batch and stream processing. Here, the processing of the sales data from the two retail stores is primarily performed incrementally via stream processing, which allows the data to be aggregated in near-real time and be made available to the dashboard. At the same time, this incremental path of the pipeline is supplemented by batch processing steps, which allow to calculate complex aggregations and to generate machine learning models from the data in the *Sold Items* data collection. Instead of needing to operate separate batch and speed layers as in the original implementation of the retail scenario, incremental stream processing can now be used as primary processing paradigm and, if required, be complemented by individual batch processing steps. As a result, data can be made available to the analytics applications more quickly and expensive re-computations of the data can be avoided to a large extent. In order to be able to break the boundaries and interleave batch and stream processing this way, the lakehouse must allow to use the same data collections as sinks and sources for batch and stream processing. This demand is supported by R5, which provides additional robustness for the data processing, as well as R6, which guarantees that no concurrency anomalies and inconsistencies can occur when different processing steps need to access the same data collections in parallel.

By taking the previously discussed aspects into account, R8 demands that a lakehouse must: (a) support batch processing, so that the contents of data collections can be read, appended and updated down to the level of data records, (b) support efficient stream processing, so that the contents of data collections can be read, appended and updated down to the level of data records at a high rate and in near-realtime, i.e. multiple operations within few seconds, and (c) allow to combine batch and stream processing by allowing to use the same data collections concurrently as sources and/or sinks for batch and stream processing tasks while ensuring data integrity in accordance with R6.

While R8 ensures that data can be processed and provided to analytics applications in near-realtime and hence is particularly relevant for streaming workloads, the facilitation of data processing as a whole is also an important aspect for enabling data management on the lakehouse, since it allows to prepare the data for the needs of the analytics applications and supports data ingestion and provisioning. By demanding the combination of batch and stream processing, which includes the usage of the same type of storage system for both processing paradigms, as well as the possibility to interleave batch and stream processing tasks as necessary, R8 contributes to the simplification of the analytics architecture and addresses C2. Furthermore, together with R2, it enables the incremental batch and stream processing of data, which can be used for the construction of incremental

Table 7 Overview of lakehouse implementations that are proposed in literature

Source	Int. Pat	Storage	Processing	Frameworks	Metadata storage
[84]	IA	HDFS	Spark	Delta Lake	
[22]	2 T	HDFS, HBase, Druid	Spark, Presto, Hive, Druid	Delta Lake, Hudi*	HBase, Atlas
[85]	IA	HDFS	Spark	Delta Lake	
[86]	2 T	HDFS, Kudu	MapReduce, Spark, Tez, Hive, Impala	Delta Lake*	
[87]	IA	HopsFS	Spark, Hive	Hudi	Hive Metastore
[88]	PA/IA	MinioS3, PostgreSQL	Spark, Trino, PostgreSQL	Iceberg	Hive Metastore

Int. Pat.: Applied Integration Pattern

PA, Parallel Architecture; 2 T, 2-Tier Architecture; IA, Integrated Architecture

stream processing pipelines that can be complemented by batch processing tasks where necessary. For many scenarios, this approach can help to avoid the deployment of expensive batch processing tasks that regularly need to re-compute the results from all available source data, which addresses the challenge C3.

Technology Review and Evaluation

While the previous sections of this paper considered the lakehouse paradigm from a conceptual perspective, this section now applies the results of this discussion to specific technologies in order to assess how lakehouses can be constructed in practice. For this purpose, the section "Lakehouse Implementations and Applications" first reviews literature in terms of different architectural approaches and technologies that are proposed for the implementation of lakehouses in various domains. Subsequently, the section "Minimal Lakehouse Architecture" introduces a minimal lakehouse architecture that serves as a reference for identifying meaningful combinations of these technologies that are of interest for the evaluation. Finally, the section "Technology Evaluation" evaluates these combinations of technologies against our technical lakehouse requirements in order to assess their suitability for the construction of lakehouses.

Lakehouse Implementations and Applications

This section discusses different implementation approaches for lakehouses that are proposed in literature. As the lakehouse paradigm is currently strongly driven by open-source projects, the review primarily focuses on works that present or discuss on-premise implementations based on open-source software for various domains. Although implementations based on cloud services are also discussed in literature (e.g. Kumar and Li [82], Shiyal [20] and L'Esteve [83]), these are often not directly comparable to on-premise implementations due to different usage models and interactions between services from the same cloud provider that lead to new characteristics and features. Therefore, these approaches need to be considered separately and are out of the scope of this paper. Table 7 summarizes several implementations, broken down by the applied integration patterns (*Int. Pat.*, cf. section "Data Warehouses and Data Lakes") and the employed technologies, which are subdivided into technologies for data storage, data processing and possibly additionally leveraged frameworks. The last column of the table indicates whether the usage of a dedicated technology for the management of metadata is mentioned.

Begoli et al. [84] present a series of case studies from the field of biomedical research in which Delta Lake on Apache Spark is employed to address different data management challenges. These challenges arise in particular from the healthcare domain, where data is supposed to be made available for research and at the same time must be protected from unauthorized access. The authors follow the conceptual lakehouse architecture as proposed by Armbrust et al. [5] and extend it for a domain-specific access control system, as well as special data ingestion and processing workflows that allow to catalogue the stored data and to extract relevant metadata.

Xiao et al. [22] also attempt to transfer the lakehouse idea to the healthcare domain, for which they built a data platform that is supposed to enable the management and integration of large amounts of heterogeneous medical data. This platform has to deal with structured, semi-structured and unstructured data that originates from different sources, such as various medical devices, which all produce data in different formats. The design of the data platform can be classified as a variant of the 2-tier architecture, where new data is first ingested into an instance of the HDFS. For analyses, two different modules exist: The data warehouse module allows to transfer structured data from the HDFS to HBase[18], where it can be processed and queried with the help of Hive. In contrast, the data lake module leverages Apache Spark with Delta Lake in order to enable the analysis of the unstructured

[18] https://hbase.apache.org.

data that is stored on the HDFS, while Apache Druid can be used for storing and querying structured data. In addition, data analysts can utilize Presto to query the data across both modules. For performance reasons, the authors plan to replace Delta Lake with Apache Hudi[19] in the future.

Ren et al. [85] also propose a data platform for the management of heterogenous medical data, which they refer to as “multi-source heterogeneous data lake platform”. The authors state that traditional data warehouses are not capable of dealing with the increasing amounts of data and that conventional data lakes did not prove sufficient, e.g. due to a lack of reliability and real-time analytics capabilities. Therefore, they chose a more modern architectural approach, which corresponds to an integrated architecture and hence shares characteristics with the lakehouse paradigm. Their data platform leverages the HDFS as underlying storage system and uses Apache Spark with Delta Lake for processing and analyzing the data. In their Function Layer, they wrap the Spark API in order to enhance it for data fusion and additional query capabilities and implement a REST API on top of it that can be used for visualizing and analyzing the data.

Park et al. [86] present the implementation of a data platform for managing and processing huge amounts of heterogenous maritime data about vessels, which in particular focuses on making the data available to AI applications. The platform builds on top of the Hadoop ecosystem and consists of a data lake that uses the HDFS as underlying storage system. Next to the data lake, a data warehouse is constructed, which utilizes the distributed storage engine Apache Kudu[20]. As Apache Kudu is not tied to the HDFS and uses a separate storage instead, it can be assumed that the data cannot be directly shared between the data lake and the data warehouse and hence needs to be transferred between both platforms. Therefore, we conclude that the proposed architecture again resembles an instance of the 2-tier architecture. For serving analytical workloads that require the processing of SQL queries on the data warehouse and the data lake, Apache Impala[21] is leveraged. In addition, an analytics module exists, which allows to process and analyze the data using Apache Spark or Apache Tez. On top of this architecture, different frameworks for data mining and machine learning can be employed as part of an AI layer, which enables the implementation of analytics applications for different use cases, such as the detection of abnormal ship behavior. The authors state that they plan to integrate Delta Lake into their architecture in the future.

Ormenisan et al. [87] pursue to enable the construction of reliable and reproducible machine learning pipelines by achieving implicit provenance and further management features. For this purpose, they constructed a feature store that uses the file system HopsFS[22] as underlying storage for data in the Apache Parquet format and leverages Apache Spark with Apache Hudi for reading and writing the data. This results in a data platform with an integrated architecture that can exploit the time travel and data versioning capabilities of Hudi in order to preserve and reproduce features that were used during different training runs of machine learning models.

The data platform presented by Tovarňák et al. [88] aims at enabling the storage, management and analysis of network telemetry data for use cases such as performance monitoring and fault detection. The authors state that traditional data warehouses did not prove sufficient for this purpose due to their complexity and high costs, while conventional data lakes impose other challenges, e.g. in terms of metadata management and a lack of transactional access. In order to overcome these issues, the authors attempt to adapt the lakehouse paradigm as described by Armbrust et al. [5] for their network telemetry scenario. The resulting data platform uses Minio S3[23] as storage system and leverages Apache Spark with Apache Iceberg for ingesting and processing the data in the Apache ORC file format, where a Hive Metastore serves as Iceberg catalog. For small amounts of data that must queried with low latency, the data platform utilizes a relational PostgreSQL[24] database, into which the data is ingested by Apache Spark. By integrating Trino, the platform aims to enable efficient query processing across this database and the data that is managed by Iceberg. When neglecting the PostgreSQL database, which is only supposed to store small amounts of data for probably operational use cases, the data platform complies with an integrated architecture.

As discussed in the section “Concepts and Definitions for Lakehouses”, multiple different understanding of the lakehouse paradigm exist in literature. Based on the previously discussed work, it can be concluded that when applied to real-world scenarios, these different perspectives also result in a broad spectrum of implementations that utilize different architectural approaches and technologies and hence possess different characteristics. The previously identified approaches and technologies are explored in more detail in the following sections in order to evaluate whether they are also suitable for the construction of lakehouses that comply with of our definition and our technical requirements.

19 https://hudi.apache.org.

20 https://kudu.apache.org.

21 https://impala.apache.org.

22 https://github.com/hopshadoop/hops.

23 https://min.io.

24 https://www.postgresql.org.

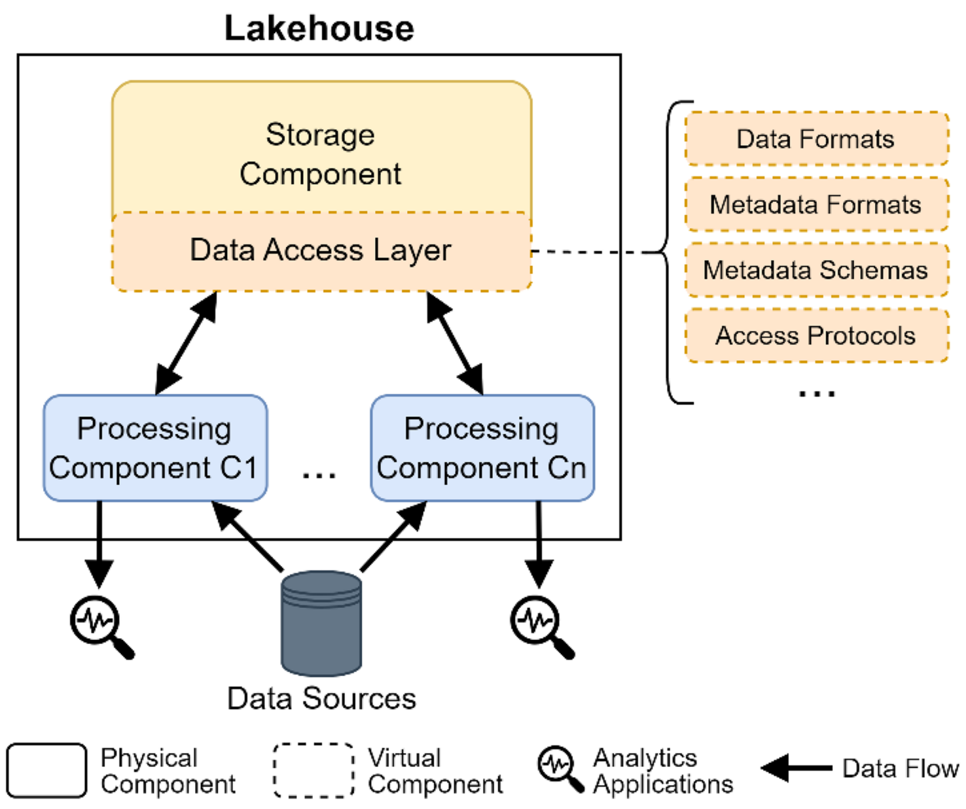

Fig. 7 Minimal lakehouse architecture that serves as a reference for identifying meaningful combinations of technologies within our evaluation

Minimal Lakehouse Architecture

The technologies that we identified in the previous section differ strongly in their purpose, their characteristics, and their dependencies and are therefore not directly comparable. In addition, some of the technologies, e.g. PostgreSQL, can already represent a self-contained data platform with storage and query capabilities themselves, while others, e.g. Apache Spark, provide only a subset of the required features and hence can only constitute a data platform in conjunction with other technologies. In addition, not all kinds of technologies are compatible with each other and some technologies may exhibit emergent characteristics and additional features when interacting with specific other technologies. For example, although Apache Spark does not provide atomicity and isolation natively (cf. section "Challenges of Data Lakes"), data can be read and written with ACID guarantees when Apache Spark is used in conjunction with a framework such as Delta Lake or Apache Iceberg. For these reasons, it is not sufficient to consider these technologies individually in order to evaluate their suitability for the construction of lakehouses; instead, they must be assessed in conjunction with other technologies with which they may be able to interact. However, on the other side, the applicability of the evaluation results is reduced when too many technologies are considered at once. In order to address these issues and enable a systematic evaluation of different technologies, we first introduce a minimal architecture for lakehouses in this section. This architecture possesses only the minimum necessary components and serves as a reference in order to identify meaningful combinations of technologies for the evaluation.

Since the purpose of every data platform is to enable the storage and management of data for analytical purposes, it must fundamentally consist of at least two basic types of components: *storage components* that can persistently store the actual data and *processing components* that allow to read data from the storage components, process data and/or write data to the storage components. For example, an instance of the HDFS usually serves as a storage component within data platforms, as it allows to store the data, while Apache Spark represents a processing component that can read, process and write data, but typically does not provide any capabilities for storing data persistently itself. Depending on their role in the corresponding data platform, many technologies will fall into one of these two categories. However, other technologies, such as PostgreSQL, can in principle be used for both storing and querying data and hence be classified as both types of components. Each data platform consists of at least one storage and one processing component; however, depending on the technologies, multiple storage and processing components may be necessary, for example if each processing component is functionally limited to either reading or writing data. In addition to storage and processing, a data

Table 8 Evaluation results for different combinations of technologies

Technology combinations				Technical requirements								LH
T	Storage	Processing	Frameworks	1	2	3	4	5	6	7	8	
1	F/O	Spark		✓		✓	✓			✓		
2	F/O, MS	Spark				✓	✓	✓				
3	F/O	Spark	Delta Lake	✓	✓	✓	✓	✓	✓	✓	✓	✓
4	F/O	Spark	Hudi	✓	✓	✓	✓	✓	✓	✓	✓	✓
5	F/O	Spark	Iceberg	✓	✓	✓	✓	✓	✓	✓	✓	✓
6	F/O, MS	Spark	LakeSoul		✓	✓	✓	✓	✓		✓	
7	F/O, MS	Spark	Iceberg, Nessie		✓	✓	✓	✓	✓		✓	
8	F/O, MS	Hive-managed tables			✓	✓	✓	✓	✓			
9	F/O, MS	Hive-external tables				✓	✓	✓				
10	Kudu, MS	Impala			✓	✓	✓	✓				
11	F/O, MS	Impala	Iceberg			✓	✓	✓	✓			
12	F/O, MS	Dremio				✓	✓					
13	F/O, MS	Dremio	Delta Lake			✓	✓					
14	F/O	Dremio	Iceberg	✓	✓	✓	✓	✓	✓	✓		
15	F/O, MS	Trino			✓	✓	✓	✓	✓			
16	F/O, MS	Trino	Delta Lake		✓	✓	✓	✓	✓			
17	F/O, MS	Trino	Hudi			✓	✓					
18	F/O, MS	Trino	Iceberg		✓	✓	✓	✓	✓			
19	Snowflake-internal tables				✓	✓	✓	✓	✓			
20	O, MS	Snowflake-external tables				✓	✓					
21	O, MS	Snowflake-Iceberg tables	Iceberg		✓	✓	✓	✓	✓			

The final column indicates whether the corresponding combination does already represent a lakehouse

F/O: File System/Object Storage; MS: Separate Metadata Storage

✓: Requirement Fulfilled

platform may also comprise other types of components, such as components for orchestrating or monitoring the data flow. For a minimal data platform and the scope of our evaluation, however, these are dispensable.

Our definition for lakehouses imposes a constraint on the data platform architecture by stating that only one type of storage system may be utilized. As a result, only one storage component has to be considered for our minimal lakehouse architecture that is illustrated in Fig. 7. In addition, the requirements that we derived from this definition (cf. section "Derived Technical Requirements") impose different technical constraints on the involved combinations of components. As shown in Table 7, most of the identified technologies can be classified as either storage or processing components or both, depending on their role in the respective architecture. This includes metadata stores, which can be considered as storage components, since their primary purpose is to store and provide metadata and they usually do not need to process complex queries.

However, the column *frameworks* of Table 7 contains technologies that do not embody independently executable, physical components. Instead, the listed frameworks rather represent a collection of different concepts that specify how the processing components should interact with the storage component in order to achieve further desirable properties, such as atomicity and isolation for read and write operations. While implementations of these concepts are often already available for certain processing components in the form of libraries, they are in principle technology-agnostic and hence generally not limited to a fixed set of technologies. In order to be able to explicitly reflect the impact of these frameworks in our lakehouse architecture despite their lack of a physical representation, we introduce a new component called *data access layer*.

As indicated in Fig. 7, this component is solely virtual and reflects concepts to which all the utilized processing components explicitly or implicitly adhere when interacting with the storage component. These concepts may include agreements on: (a) the *data formats* that are used for the data that is supposed to be stored on the storage component, (b) the *metadata formats* that are used for the technical metadata that is supposed to be stored on the storage component, (c) the *data schemas* that are used for modeling the technical metadata and (d) *access protocols* that specify how the processing components may read and write data and metadata on the storage component, possibly including the reading and/or writing of additional technical metadata. For example, the framework Delta Lake demands that the processing components use Apache Parquet as data format, JSON as format for the metadata and specifies as part of its Delta Transaction Log Protocol[25] how metadata has to be structured and how data accesses have to be performed. For Apache Hudi and Apache Iceberg, similar specifications exist. It is important to note that all processing components share the same data access layer, as otherwise no compatibility in terms of formats, schemas and protocols may be achieved. Furthermore, the data access layer is considered to be always present, even if no data lake framework or comparable technology is leveraged, as it just represents the concepts on which all involved processing components implicitly or explicitly agree.

Technology Evaluation

This section evaluates numerous combinations of technologies regarding their suitability for the construction of lakehouses. By considering combinations of technologies instead of individual technologies, possibly existing interactions that lead to emergent features and characteristics can be covered as well. The assessed technologies were selected based on their perceived relevance in the ongoing lakehouse debate and are hence either commonly discussed in the context of the lakehouse paradigm (cf. e.g. [25, 26]) or were identified as part of the literature discussion that we conducted on existing lakehouse implementations in the section "Lakehouse Implementations and Applications". All of these technologies are opensource, except for Snowflake,[26] which we added as a representative for cloud-based data warehouses in order to enable a wider range of comparisons. In total, we examined 21 different combinations of technologies, which are summarized along with the evaluation results in Table 8. Here, each row corresponds to one of the investigated combinations, whose technologies are detailed in the columns two to four. The following columns indicate which of the technical requirements R1-R8 are met by the respective combination, while the last column indicates whether the combination already constitutes a lakehouse, which is only the case if all eight requirements are met. The combinations listed in this table were selected with the help of the minimal lakehouse architecture as follows: Since the storage component of our architecture is only challenged by the two requirements R1 and R7, we decided to focus on the processing components and the data access layer and to select a compatible technology for the storage component only after and in dependence to the other components. Since most technology combinations behave similarly for different instances of file systems or object storages, these storage systems are summarized under the wildcards *F* and *O* in Table 8. Similarly, the wildcard *MS* indicates that the technology combination relies on a separate storage system for managing all or parts of the technical metadata. Examples for such metadata storages include the Hive Metastore[27] (HM) and compatible implementations.

We found that the evaluation will offer the greatest practical applicability when only one single processing component is considered per combination. As a consequence, not each of the considered combinations must necessarily represent a full-fledged data platform, as some of the processing components may be limited to either reading or writing data and data platforms generally have to support both. However, through this approach, our evaluation can provide insights on the individual contributions that a processing component, possibly supported by a framework for the data access layer, can make for the construction of lakehouses. Based on our evaluation results, it can then be easily inferred if and how different technologies can complement each other to form a lakehouse that complies with our definition. For example, Dremio with Iceberg tables on an object storage, which is included in Table 8 as technology combination T14, does not already constitute a lakehouse, as R8 is unfulfilled due to missing support for stream processing. However, when Apache Spark with Apache Iceberg (T5) is utilized as second processing component in addition to Dremio, a lakehouse can be created that provides features of both Apache Spark and Dremio.

Data Platforms with Apache Spark

Apache Spark is a popular distributed batch processing engine which is leveraged in almost all implementations that we discussed in the section "Lakehouse Implementations and Applications". With the help of its Structured Streaming module,[28] Spark also supports stream processing via micro-batching, where continuous data streams are processed in the form of small batch processing tasks with a programming model that is similar to that of batch processing [63]. Among many other sources and sinks, Spark can also read datasets from and write datasets to file systems and compatible object storages in open file formats, including CSV, Apache Parquet and Apache ORC, without needing to replicate data or outsourcing metadata. Hence, R1 and R7 are satisfied for technology combination T1. However, Spark considers datasets as immutable and does not support random updates or deletes of individual records (cf. C3) and therefore does

[25] https://github.com/delta-io/delta/blob/master/PROTOCOL.md.

[26] https://www.snowflake.com/.

[27] https://cwiki.apache.org/confluence/display/Hive/AdminManual+Metastore+Administration.

[28] https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html.

not fulfill R2. While Spark offers a relational abstraction for datasets and SQL as query language within its Spark SQL module[29] and hence meets R3 and R4, it does not provide sufficient means for implicitly enforcing the consistency of data collections (cf. C4) without a HM. When a HM is integrated, as represented by technology combination T2, persistent tables with a pre-defined schema can be created [64], hence allowing schema enforcement and fulfilling R5. However, the backend of the HM is typically a relational database, which represents another type of storage system besides the one that is leveraged for the actual data and thus violates R1. In addition, the HM does not provide unmediated access to the technical metadata, which is why R7 is not satisfied. Regardless of whether an instance of the HM is used, Spark does not guarantee isolation and atomicity (cf. C5), which are required for R6. Similarly, stream processing on file systems and object storages is not sufficiently supported by Spark due to a lack of optimizations (cf. C2). Therefore, it can be concluded that Apache Spark with a file system or object storage does not constitute a lakehouse.

In recent years, several open-source frameworks have emerged that pursue to implement features and characteristics of data warehouses on top of data lakes. For this purpose, they offer various formats, schemas, protocols and other specifications by which the data access layer can be strengthened and various weaknesses of processing engines be addressed. These frameworks typically already provide libraries and extensions for common processing engines that then control how data is read from and written to file systems or object storages, allowing them to enforce their specifications. The most popular representatives of these frameworks are Delta Lake (cf. T3), Apache Hudi (cf. T4) and Apache Iceberg (cf. T5), but more recently, also LakeSoul[30] (cf. T6) arose as a possible alternative. All four frameworks manage technical metadata for the stored data, including information about partitions, the structure of data collections and the data files that belong to them. By exploiting this metadata, the frameworks increase the relational abstraction of datasets and, for example, enable Apache Spark to randomly append, update and delete individual records of datasets, as well as to persistently store schemas of data collections and enforce it on new or changed data. This way, R2 and R5 can be satisfied. Even more important, the frameworks maintain a log as part of their technical metadata that tracks changes to the data and indicates which data files comprise the current and past versions of each data collection. With the help of this log, the frameworks enable typical data warehouses features, such as time travel, and implement data access protocols that ensure atomicity and at least snapshot isolation for read and write operations via multi-version concurrency control [89, 90], satisfying R6. Furthermore, they provide various improvements for stream processing on top of file systems or object storages, such as write compaction techniques or different types of data collections that are either optimized for writing or reading data. In conjunction with Apache Spark, these improvements enable unified batch and stream processing, where the same data collections can be concurrently used as sources and/or sinks, as demanded by R8. Despite applying different approaches for ensuring atomicity and isolation, Delta Lake, Apache Hudi and also Apache Iceberg, at least when using the Hadoop-based implementation of the Iceberg Catalog,[31] can all store their required technical metadata in open formats and directly accessible on the same storage system as the actual data and do not rely on other storage components. In contrast, LakeSoul inherits its atomicity and isolation guarantees from a PostgreSQL database [91], which represents a mandatory dependency for managing the technical metadata. For this reason, the technology combination T6 does not satisfy R1 and R7.

Although the frameworks appear to be very similar from a high-level perspective, their underlying design principles differ strongly. Jain et al. [90] analyze and compare Delta Lake, Apache Hudi and Apache Iceberg in depth and also propose and apply a benchmark for evaluating the performance of such data platforms. Similarly, Camacho-Rodríguez et al. [92] also examine characteristics of these three frameworks and propose another benchmark for their empirical evaluation of data lake frameworks. It builds on the same decision support benchmark as the one proposed by Jain et al., but considers more diverse workloads, which also allow to evaluate aspects such as longevity and time travel capabilities. Beside these empirically supported evaluations, Weller [93] and Belov and Nikulchev [94] provide purely conceptual comparisons.

Project Nessie[32] can be considered as an extension for Apache Iceberg, as it provides a custom implementation of the Iceberg Catalog and enhances the framework for semantics as known from Git.[33] With Nessie, data analysts or processing tasks can create a new branch based on the current state of the stored data, modify data on this branch, commit their changes atomically and finally merge all their commits on the new branch into the main branch at once. By applying this approach, cross-table transactions that span several data collections become feasible. Furthermore, it allows multiple subtasks of a processing job to commit their changes individually on a working branch and then merge this working

[29] https://spark.apache.org/docs/latest/sql-programming-guide.html.

[30] https://github.com/lakesoul-io/LakeSoul.

[31] https://iceberg.apache.org/javadoc/latest/org/apache/iceberg/hadoop/HadoopCatalog.html.

[32] https://projectnessie.org/.

[33] https://git-scm.com.

branch into the main branch as soon as all tasks have successfully completed, leading to similar semantics as known from distributed transactions [95]. The core of Nessie constitutes a dedicated server that offers a REST API and requires a separate database as backend for storing technical metadata, such as RocksDB[34] [96]. Due to these dependencies, R1 and R7 are not fulfilled by technology combination T7.

Data Platforms with Apache Hive

Similar to the previously discussed frameworks, also Apache Hive pursues to provide data warehousing capabilities for data lakes. However, in contrast to these frameworks, Hive does not act as a library or extension for other processing engines, but rather represents a self-contained data warehouse that is built on top of the Hadoop ecosystem. Hive supports the HDFS and several object storages as underlying storage systems for the data, as well as a variety of open data formats, including Apache Parquet and Apache ORC. By using a command-line interface, users can register tables that abstract from the stored data files and represent them as relational tables, which satisfies R3. The technical metadata about these tables, including information about their columns and partitions, are not managed on the same storage as the actual data. Instead, Hive uses an instance of the Hive Metastore for this purpose, which relies on a relational backend. Consequently, Hive does not satisfy R1 and R7, as the technical metadata resides on another type of storage and is not directly accessible. In Hive, two types of tables exist: *Managed tables* are exclusively managed by Hive itself, meaning that the data and the structure of tables can only be modified using Hive commands, including all kinds of DDL and DML operations. Although the data continues to be stored in open formats on the storage system, where it can be read and in principle also be written by external processes outside of Hive, such modifications can lead to inconsistencies and undefined behavior, as the metadata managed by Hive is not automatically updated. In contrast, with *external tables*, the data is not assumed to be owned by Hive and external modifications are permitted. However, some DML operations, such as updates and deletes of individual records, are limited to managed tables and hence only managed tables fulfill R2. Via the command-line interface, users can issue queries against data in both types of tables via HiveQL, a SQL-like query language that satisfies R4. Hive then translates the queries into processing jobs for distributed query engines like MapReduce[35] or Apache Spark and initiates their execution. When registering tables, Hive requires the specification of a schema, which is persisted in the HM and enforced "on-read" when querying the data, hence fulfilling R5. Optionally, Hive can provide atomicity and snapshot isolation [89], which satisfy R6, but only for managed tables that use Apache ORC as data format [97]. Although Hive offers a Streaming API[36] for managed tables with enabled ACID guarantees, it only allows to ingest streaming data from stream processing engines and cannot be used to treat Hive tables as sources of data streams. Furthermore, Hive does not provide any stream processing capabilities itself. For this reason, R8 is neither fulfilled for external nor for internal Hive tables.

Data Platforms with Apache Impala

Apache Impala [98] is a distributed SQL engine that can be used to query data from different data sources of the Hadoop ecosystem, including Apache Kudu and Apache HBase. Furthermore, it can query data through a data access layer established by Apache Iceberg on an instance of the HDFS or a cloud object storage. In the scope of this evaluation, Impala is considered in combination with Kudu (cf. T10) and Iceberg (cf. T11), for which it already fulfills R4. Apache Kudu describes itself as a "distributed storage engine" for analytics and represents an open-source storage system that also integrates with the Hadoop ecosystem. Hence, it can be utilized by various technologies that are associated with Hadoop, including MapReduce, Apache Spark and Impala. Despite this integration, Kudu manages and stores its data independently and does not run on an instance of the HDFS. Kudu only constitutes a storage component that provides basic interfaces for efficiently storing the data and does not neither include query capabilities, nor means for processing the data. For this purpose, Kudu must be used in combination with compatible processing components, such as Impala. Internally, Kudu leverages a column-oriented data format with compression for persisting the data. This format is inspired by Apache Parquet, but includes various optimizations and is not directly accessible from outside of Kudu [99], which violates R7. Similar to relational databases, Kudu allows to store data in tables with a pre-defined schema that supports primary keys and different data types, including binary large objects. Due to this relational abstraction and the enforcement of schemas "on-write", R3 and R5 are satisfied. However, in contrast to typical relational databases, Kudu has not adapted support for multi-row transactions yet and therefore does not provide ACID guarantees [99], which conflicts with R6. When used with Kudu, Impala also supports DML statements for appending, updating and deleting individual rows [100], which fulfills R2. However,

[34] https://rocksdb.org.

[35] https://hadoop.apache.org/docs/r1.2.1/mapred_tutorial.html.

[36] https://cwiki.apache.org/confluence/display/Hive/Streaming+Data+Ingest+V2.

while Kudu can optionally be integrated with the Hive Metastore [99], Impala necessarily relies on an instance of it for accessing table definitions. As the HM most likely uses a relational database as backend that represents another type of storage system, R1 is not fulfilled. Furthermore, Kudu und Impala do not provide stream processing capabilities out of the box and hence cannot meet R8. When using Impala with Apache Iceberg on an instance of the HDFS or a cloud object storage instead of Apache Kudu (cf. T11), Impala interacts with a so-called Iceberg catalog, which is a central software component of Iceberg that holds a pointer to the metadata of each table and serves as an entry point for performing data operations, such as queries. By exploiting the guarantees of the underlying storage system, Iceberg catalogs are also responsible for ensuring ACID properties for these operations. Although Impala supports different types of Iceberg catalogs, including the Hadoop-based implementation, it still relies on the HM. Nevertheless, in comparison to T10, the evaluation results for T11 change with respect to two requirements: On the one hand, Impala does not support row-level update and delete operations for Iceberg tables [101] and hence violates R2 in this technology combination. On the other hand, R6 can now be fulfilled, since Iceberg can ensure atomicity and isolation for read and write operations in the scope of the data access layer.

Data Platforms with Dremio or Trino

Dremio[37] is entitled as a "open lakehouse" that brings self-service analytics and data warehouse functionality to data lakes. A similar tool is Presto[38] [102], which describes itself as a distributed SQL engine for data analytics and "the open lakehouse" and allows to query large datasets that are possibly distributed over several different data sources. Trino[39] is a popular fork of Presto which evolved separately, but still shares a lot of features and characteristics with the original project. Both tools are commonly mentioned in the context of lakehouses and appear to play an important role in several prototypical lakehouse implementations (cf. [5, 88, 103]). In this evaluation, we focus on Trino and the open-source version of Dremio as representatives for this type of technology. Despite their different focus, Dremio and Trino share several similarities from a high-level perspective: Both technologies represent processing components that are built around SQL-based query engines and allow to query, join and analyze data across different storage components, such as databases of different kind, distributed file systems or object storages. This way, the data can reside on various storage systems in various formats does not need to be extracted, transformed and loaded into a separate data warehouses for analyses. Both Dremio and Trino can access and query data that is stored in an instance of the HDFS or in cloud object storages, which is reflected by the technology combinations T12 and T15, respectively. Among other formats, Dremio accepts data in Apache Parquet, JSON and Excel [104], while Trino's Hive Connector[40] supports Apache Parquet, Apache ORC, Apache Avro and further formats [105]. Both technologies allow to abstract and query the stored data files as tables, which satisfies R3 and R4. Dremio can either derive and locally cache the technical metadata about the available datasets and their structure itself, or retrieve it from a connected HM or AWS Glue[41] deployment [106]. However, in both cases, the metadata is stored on another type of storage system as the actual data and does not allow direct, unmediated access, which violates R1 and R7. Trino always relies on an external HM or a compatible implementation for querying data that resides on the HDFS or object storages [105] and hence also cannot meet the two requirements in this technology combination. As Dremio natively does not provide DML support for the stored data [107], it can neither be used to modify the data, nor to provide consistency guarantees or atomicity and isolation for read and write operations. Hence, it does not fulfill R2, R5 and R6 in this technology combination. In contrast, Trino allows to append, update and delete individual records, but only for datasets in the Apache ORC format that represent tables with enabled ACID guarantees in the HM [105]. For this data, Trino can satisfy the requirements R2, R5 and R6. Finally, as Dremio and Trino constitute distributed SQL query engines that are designed for interactive data analyses, they do not provide stream processing capabilities and thus cannot fulfill R8.

Similar to Impala, Trino and Dremio can also be used to query data that resides on an instance of the HDFS or a cloud object storage through a data access layer that is established by one of the previously discussed data lake frameworks. While Dremio supports Delta Lake (cf. T13) and Apache Iceberg (cf. T14), Trino offers dedicated connectors for Delta Lake[42] (cf. T16), Apache Hudi[43] (cf. T17) and Apache Iceberg[44] (cf. T18). When used in combination with Delta Lake, Dremio needs to regularly retrieve the technical metadata from the storage component and cache it in an internal metadata store in order to be able to query the latest version of the data. In addition, DML operations are still not supported. Therefore, the evaluation results of

[37] https://www.dremio.com.

[38] https://prestodb.io.

[39] https://trino.io.

[40] https://trino.io/docs/current/connector/hive.html.

[41] https://aws.amazon.com/de/glue/.

[42] https://trino.io/docs/current/connector/delta-lake.html.

[43] https://trino.io/docs/current/connector/hudi.html.

[44] https://trino.io/docs/current/connector/iceberg.html.

T13 do not change in comparison to T12. However, when Apache Iceberg is leveraged, Dremio can interact with an Iceberg catalog [108], such as the Hadoop-based catalog implementation that can reside on the same storage system as the actual data. Hence, no technically relevant metadata must be stored on another type of storage system and all data, as well as the metadata, remain available in open formats and can be directly accessed on the storage component. Therefore, R1 and R7 can be considered fulfilled. Furthermore, Dremio supports DML operations for Iceberg tables, including appends, updates and deletes of individual records [109], which meets R2. By exploiting the guarantees that Iceberg adds to the data access layer, the technology combination T14 can also provide consistency guarantees, as well as atomicity and isolation, which are required by R5 and R6. In contrast to Dremio, Trino supports DML operations when used with Delta Lake, including append, update and delete operations on the record level [110]. In addition, Delta Lake can enrich these operations for consistency guarantees, as well as atomicity and isolation. Therefore, technology combination T16 fulfills the requirements R2, R5 and R6. However, in order to integrate with Delta Lake, Trino requires access to a metadata store [110], such as the HM, which again prevents the fulfillment of R1 and R7. This also applies to technology combination T17, in which Trino utilizes Apache Hudi for the data access layer and relies on a separate HM. However, in contrast to Delta Lake, Trino does not support DML operations for Hudi [111] and hence T17 cannot fulfill R2, R5 and R6. When using Trino in combination with Apache Iceberg, as suggested by T18, Trino requires access to either an instance of the HM, a compatible metadata store implementation like a AWS Glue deployment, or an Iceberg catalog [112]. However, instead of the Hadoop-based implementation of the Iceberg catalog, currently only catalogs based on JDBC or REST are supported [112], which impedes the storage of the technical metadata along with the actual data. Although it may be possible to create custom catalogs based on either JDBC or REST that leverage the same type of storage as it is used for the actual data, this would likely require additional development efforts and hence we currently consider R1 and R7 as not fulfilled. Nevertheless, similar to T14, also T18 provides sufficient DML operations so that R2, R5 and R6 can be satisfied due to the data access layer.

Data Platforms with Snowflake

As discussed in the section "Concepts and Definitions for Lakehouses", some authors argue that modern, cloud-based data warehouses already represent lakehouses, as they increasingly attempt to adopt typical characteristics of data lakes. An example of such a data warehouse is Snowflake, which provides several features that go beyond those of traditional data warehouses, including its cloud-native alignment, the separation between compute and storage resources and additional management and query capabilities for semi-structured data [11]. Snowflake supports three types of tables [113], which are evaluated separately. Data in *internal tables* (cf. T19) is managed exclusively by Snowflake. It is typically ingested via ETL pipelines and stored in the cloud using Snowflake's proprietary, performance-optimized data format, where it cannot be directly accessed by users or external processes, as all details are hidden behind the cloud offering. Consequently, R1 and R7 are not fulfilled. The internal tables are comparable to tables of typical relational databases and hence enforce pre-defined schemas, can be queried using SQL and allow to append, update and delete individual rows, which satisfies the requirements R2, R3, R4 and R5. Furthermore, all operations are executed with ACID guarantees, which also fulfills R6. In additional, Snowflake provides various data warehousing capabilities and features for internal tables, including time travelling, stored procedures, data governance policies and many more. Snowflake also offers libraries that provide dataframe-like syntax for programming languages like Python, where the performed operations are converted to SQL and subsequently executed on the data warehouse. However, while batch processing is supported by Snowflake in accordance with R8, stream processing is mostly limited to the ingestion of streaming data.

In contrast to internal tables, the data of *external tables* (cf. T20) is neither managed, nor stored by Snowflake, as it resides as files in open formats on third-party cloud object storages. Among others, Snowflake supports CSV, JSON and Apache Parquet as data formats for external tables. Snowflake abstracts the stored data files as tables and allows to query them similar to internal tables, which satisfies R3 and R4. However, external tables are read-only cannot be written or modified by Snowflake [114], which is a missing prerequisite for R2, R5 and R6. Optionally, a Hive Metastore can be integrated, which allows Snowflake to synchronize its internally managed metadata about the external tables with the metadata store [115]. Without the HM, Snowflake needs to regularly update its internally managed metadata about the external tables, which can be triggered either manually or automatically based on notifications from the cloud provider. In both cases, metadata needs to be stored on a different type of storage system than the actual data and is not directly accessible, which violates R1 and R7.

The third type of tables in Snowflake are *Iceberg tables* (cf. T21). They are similar to external tables, with the difference that Apache Iceberg is utilized to establish a data access layer that governs the access to the data that resides on the third-party cloud object storage. For this purpose, Snowflake manages an Iceberg catalog itself [116], which allows to treat Iceberg tables more similar to internal tables, including full DML support and ACID guarantees for data operations [113]. This way, the requirements R2, R5 and R6 can be satisfied by technology combination T21. However, as the Iceberg catalog resides within Snowflake and not on the same storage system as the actual data, its metadata is not directly accessible and R1 and R7 are not fulfilled. Nevertheless, Iceberg tables allow to keep data in open formats on external cloud object storages and to share it between Snowflake and other processing components, which avoids the data being locked-in. In addition, Snowflake offers features that go beyond those natively supported by Apache Iceberg, such as multi-table transactions [113].

Summary of the Evaluation

From the 21 assessed technology combinations, only three fulfilled all eight requirements and hence can be considered as lakehouses that comply with our definition. These combinations use Apache Spark as processing component and either an instance of the HDFS or an object storage as storage component. In addition, they utilize one of the three data lake frameworks Delta Lake, Apache Hudi or Apache Iceberg, which act as a library for Spark and strengthen the data access layer by abstracting the stored data as relational data collections, allowing DML operations on the level of records, providing consistency guarantees, ensuring atomicity and isolation for data operations and supporting the unification of batch and stream processing. It is worth noting that both SQL query engines like Impala, Dremio and Trino, as well as data warehouses like Snowflake are currently increasingly adding support for these data lake frameworks. While the integration of Apache Iceberg appears to be the most progressed, possibly due to its flexible catalog-based approach that supports different systems for the management of technical metadata, the support for Delta Lake and Apache Hudi still appears to be rather sparse. For example, DML operations are often not supported when one of these two frameworks are used and the processing components must regularly retrieve and cache metadata, as opposed to processing the metadata on demand. Nevertheless, based on these developments, a trend towards an open architecture for data platforms can be observed, in which the data is stored in open formats on a single type of storage system that provides direct access. Analogously to the lakehouse architecture of Fig. 7, different types of processing components, which are selected depending on the requirements of the analytics scenario, can then process and query the data. This includes batch and stream processing engines, such as Apache Spark and Apache Flink, but also SQL query engines such as Dremio and Trino or even data warehouses like Snowflake that support to query data on external, third-party storage systems. In addition to these components, a data lake framework is selected, which then acts as extension for the involved processing components and ensures that the consistency of the data is preserved despite concurrent operations. Furthermore, it can provide additional features to the processing engines, such as time travel capabilities. The resulting architecture provides high flexibility, as the data is not locked-in and various tools that are specialized for different types of analyses can be leveraged in parallel.

Discussion

In the previous sections, a comprehensive overview of concepts and technologies related to the lakehouse paradigm was provided. The main contributions included the identification of typical challenges that conventional data platforms and especially data lakes are commonly facing, the outlining of lakehouses as distinct architectural approach for data platforms, the proposal of a new, sharper definition for lakehouses, the derivation of corresponding technical requirements, as well as the evaluation of different technologies regarding their suitability for the construction of lakehouses. This section first discusses several threats to validity, which may limit the generalizability and applicability of the aforementioned results. Subsequently, it points to open research gaps that may be addressed as part of future works in this field.

Threats to Validity

For the results of this work, we identified several threats of validity, which are described and discussed in detail below.

Completeness and Relevance of the Identified Challenges

In the scope of the sections "Overview of Data Platforms" and "Challenges of Data Lakes", several challenges were identified and discussed that commonly arise during the development and operation of conventional data platforms. Thereby, the focus was particularly on data lakes, for which a total of five technically-focused challenges were described. These challenges are derived from empirical observations by the authors and descriptions from literature, including blog articles and technical reports from technology vendors. The selection of the challenges is based on their perceived frequency of occurrence and relevance for typical enterprise analytics architectures. However, since every domain and application scenario shows its own requirements and restrictions, these challenges can neither be considered complete nor equally relevant in every context. For example, the challenge of combining batch and stream processing (cf. C2) can be neglected in scenarios in which the data only needs to be processed as batches due to relaxed timing requirements. Similarly, aspects like access control [117], privacy [118], data preparation [119] and metadata management [26] may pose important challenges in specific application scenarios.

Diverging Motivations for the Development of Lakehouses

For the development of our lakehouse definition and the subsequent derivation of technical requirements, we assumed the simplification of enterprise analytics architectures as the fundamental, overarching goal of lakehouses and also aligned the definition and requirements accordingly. This goal is either explicitly or implicitly reflected in the majority of works in literature and also represented in the term “lakehouse” itself. However, due to the wide ambiguity surrounding the term and its associated concepts (cf. section “Concepts and Definitions for Lakehouses”), also other perspectives may arise that see different motivations for the lakehouse paradigm. Accordingly, the concepts and discussions presented in this paper are only applicable under the assumption that the simplification of enterprise analytics architectures constitutes the primary goal of lakehouses.

Diverging Understandings for Analytical Workloads

The definition for lakehouses as proposed in this paper specifies that lakehouses must be able to serve the typical analytical workloads of data warehouses and data lakes in order to be able to replace them in many cases and thus to allow the simplification of enterprise analytics architectures. These workloads are characterized in the section “Definition and Requirements for Lakehouses” and subsequently used to derive eight technical requirements that a lakehouse must meet in order to comply with the definition. Although the characterization of the mentioned analytical workloads is largely based on literature, some simplifying assumptions had to be made, as the workloads are described differently in various sources and also other terms with partially overlapping meanings are introduced, such as *business intelligence*. Consequently, the assumptions made in this paper do not necessarily reflect the workloads of every application scenario. Since the characteristics of the workloads were also used to derive the technical requirements in the section “Definition and Requirements for Lakehouses”, it is therefore possible that these requirements do not cover all aspects that may be necessary for lakehouses to replace data warehouses and data lakes in practice.

Limitations of the Technology Evaluation

In order to enable a structured evaluation of different technologies and their interactions with respect to their suitability for the construction of lakehouses, the section “Technology Review and Evaluation” introduced a minimal lakehouse architecture which served as a reference for the identification of meaningful technology combinations. However, this approach also led to the consideration of a rather idealistic lakehouse architecture with only few involved technologies, which is probably rarely encountered in practice. Instead of individual ones, different processing engines and even different frameworks may be used in parallel. Furthermore, also other types of technologies are generally relevant in practical implementations, such as tools for the orchestration of processing jobs and the management of metadata, which were not considered in the context of this architecture. The selection of the evaluated technologies also poses limitations, as these were chosen based on their perceived relevance and popularity, whereby cloud services were mostly excluded from consideration throughout the paper. Accordingly, our evaluation focused on a subset of several popular technologies, which has already allowed us to derive interesting and helpful insights. However, especially in view of the rapid innovations and developments in this field, an extended evaluation may be necessary in the scope of future work in order to increase the practical applicability of the results.

Future Research Directions

The area of analytical data platforms represents a broad research field that is currently subject to fast developments and rapid innovation. Lakehouses constitute a newer subfield, in which still many open questions exist that should be addressed in the course of future research. On the one hand, there is a need for research at a technological level, which aims at the optimization of lakehouse technologies and thus pursues to adapt and improve features like complex queries, advanced index structures, data versioning capabilities, interoperability and access control for large data volumes on lakehouses. This way, the associated technologies can reach a higher level of maturity.

On the other hand, also research at a conceptual level is necessary. As discussed for the threats to validity, there is currently still a lack of long-term empirical information on the motivational factors for enterprises to utilize a lakehouse, important architectural decisions and the extent to which the associated technologies and concepts have proven themselves in practice. Accordingly, it remains an open question under which circumstances lakehouses pose viable alternatives to conventional data platforms. In order to address this gap, future works could collect empirical experiences and analyze them in order to derive guidelines and recommendations for enterprises.

In addition, research in terms of lakehouse architectures is required, as it is currently unclear how such data platforms should be designed and which architectural decisions and trade-offs have to be made during their development. Consequently, there is uncertainty about which existing and proven concepts from data warehouses and data lakes can also be applied to lakehouses. Among others, this particularly affects architectural aspects such as data organization, data modelling, metadata management and access control. For example, it is not entirely clear whether unstructured data like images, videos and documents should generally also be managed in a lakehouse.

Another open research gap lies in the question of how modern data platforms like lakehouses can be enhanced for self-service analytics capabilities. Here, the goal is to empower data analysts and data scientists so that they can find, access, prepare and exploit their required data on the data platform on their own. In addition to challenges such as the consideration of privacy aspects and the implementation of access control mechanisms, this requires the development of concepts that give users the opportunity to prepare data and store pre-processed data on the data platform, for example by applying suitable aggregation and filter operations. As of today, users generally either need to operate their own infrastructure or rely on data engineers for this purpose.

Conclusions

This paper provided a comprehensive overview of concepts, definitions, implementation approaches and technologies related to the lakehouse paradigm. In addition, a conceptual foundation was established that outlines lakehouses as a distinct architectural approach. This includes the identification of typical challenges that conventional data platforms are facing, the introduction of a new, sharper definition for lakehouses that incorporates their potential benefits for enterprises, as well as the derivation of technical requirements that lakehouses should fulfill in order to comply with the definition. The considerations behind these concepts were explained and demonstrated with the help of a representative analytics scenario, for which a baseline implementation based on a data warehouse and data lake and an implementation based on a lakehouse were compared. Finally, the paper introduced a minimal lakehouse architecture that served as framework for the identification of meaningful and relevant combinations of different popular technologies and evaluated them regarding their suitability for the construction of lakehouses by applying the technical requirements. As a result of this evaluation, it turned out that especially frameworks for data lakes, such as Delta Lake, Apache Hudi and Apache Iceberg, can currently be considered as enablers for lakehouses, as they can enhance several processing components for features and characteristics that allow them to access the data in a uniform and consistency-preserving manner. Consequently, this leads to the emergence of an architecture where the data is stored centrally and in a directly accessible manner, so that it can be exploited by different types of consuming applications in parallel. This way, the typical analytical workloads of data warehouses and data lakes can be served from one single data platform.

Our work provides a conceptual foundation for fostering further discussions and research regarding the lakehouse paradigm. In future work, we plan to investigate the costs and benefits of the different types of data platforms in more detail and to address some of the open research gaps that were discussed in the section "Discussion".

Author Contributions JS conducted most of the research, developed the example scenario, performed the technology evaluation and wrote the final manuscript. CG and AL significantly enriched the work with their scientific expertise and their experiences from industrial practice, which are particularly reflected in the design of the example scenario and the conceptual results of this paper. CG, AL and HS were engaged in crucial discussions regarding the contents of the paper and strongly improved its quality by incorporating their knowledge and views on all covered aspects. CG, HS and BM supervised the work and provided highly valuable feedback that strengthened the focus, structure and the reasoning of the work. HS also edited the manuscript.

Funding Open Access funding enabled and organized by Projekt DEAL. No funding was received for this research.

Data Availability In the scope of this research, no datasets were analyzed or generated.

Declarations

Conflict of interest On behalf of all authors, the corresponding author states that there is no conflict of interest.

Informed Consent This research does neither include a study, nor involve participants and hence no informed consent is required.

Involvement of Human Participants and/or Animals This research does neither include a study, nor involve human participants or animals.

References

1. Lasi H, Fettke P, Kemper H-G, Feld T, Hoffmann M. Industry 4.0. Bus Inf Syst Eng. 2014;6:239–42.
2. Gröger C. Industrial analytics—an overview. IT Inf Technol. 2022;64:55–65.
3. Inmon WH. Building the data warehouse. New York: Wiley; 2005.

4. Giebler C, Gröger C, Hoos E, Schwarz H, Mitschang B. Leveraging the data lake: current state and challenges; 2019.
5. Armbrust M, Ghodsi A, Xin R, Zaharia M. Lakehouse: a new generation of open platforms that unify data warehousing and advanced analytics. In: Proceedings of CIDR, vol. 8; 2021.
6. Nambiar A, Mundra D. An overview of data warehouse and data lake in modern enterprise data management. BDCC. 2022;6:132. https://doi.org/10.3390/bdcc6040132.
7. Rosenbaum A, Edjlali R, Ronthal A. Hype cycle for data management 2023. Online; 2023.
8. Schneider J, Gröger C, Lutsch A, Schwarz H, Mitschang B. Assessing the lakehouse: analysis, requirements and definition proceedings of the 25th international conference on enterprise information systems, vol 25 (2023)
9. Kimball R, Ross M. The data warehouse toolkit. The definitive guide to dimensional modeling. New York: Wiley; 2013.
10. Haerder T, Reuter A. Principles of transaction-oriented database recovery. ACM Comput Surv. 1983;15:287–317. https://doi.org/10.1145/289.291.
11. Dageville B, Cruanes T, Zukowski M, Antonov V, Avanes A, Bock J, Claybaugh J, Engovatov D, Hentschel M, Huang J, et al. The snowflake elastic data warehouse. In: Özcan F, Koutrika G, Madden S, editors. Proceedings of the 2016 international conference on management of data. New York: ACM; 2016. p. 215–26. https://doi.org/10.1145/2882903.2903741.
12. Baars H, Kemper H-G. Business intelligence and analytics. Wiesbaden: Springer; 2021.
13. Bose R. Advanced analytics: opportunities and challenges. Ind Manag Data Syst. 2009;109:155–72. https://doi.org/10.1108/02635570910930073.
14. Dixon J. James Dixon's blog. Pentaho, Hadoop, and Data Lakes (2010). https://jamesdixon.wordpress.com/2010/10/14/pentaho-hadoop-and-data-lakes/. 23.01.2024.
15. Giebler C, Gröger C, Hoos E, Schwarz H, Mitschang B. A zone reference model for enterprise-grade data lake management. In: 2020 IEEE 24th international enterprise distributed object computing conference (EDOC). IEEE; 2020. p. 57–66. https://doi.org/10.1109/EDOC49727.2020.00017.
16. Sawadogo P, Darmont J. On data lake architectures and metadata management. J Intell Inf Syst. 2021;56:97–120. https://doi.org/10.1007/s10844-020-00608-7.
17. Gröger C. There is no AI without data. Commun ACM. 2021;64:98–108.
18. Oreščanin D, Hlupić T. Data lakehouse—a novel step in analytics architecture. In: 2021 44th international convention on information, communication and electronic technology (MIPRO); 2021. p. 1242–6.
19. Jain P, Kraft P, Power C, Das T, Stoica I, Zaharia M. Analyzing and comparing lakehouse storage systems. In: Proceedings of the CIDR; 2023.
20. Shiyal B. Beginning azure synapse analytics. Transition from data warehouse to data lakehouse. New York: Apress; 2021.
21. Hlupić T, Oreščanin D, Ruzak D, Baranovic M. An overview of current data lake architecture models. In: 2022 45th jubilee international convention on information, communication and electronic technology (MIPRO). IEEE; 2022. p. 1082–7. https://doi.org/10.23919/MIPRO55190.2022.9803717.
22. Xiao Q, Zheng W, Mao C, Hou W, Lan H, Han D, Duan Y, Ren P, Sheng M. MHDML: construction of a medical lakehouse for multi-source heterogeneous data. In: Traina A, Wang H, Zhang Y, Siuly S, Zhou R, Chen L, editors. Health information science. Lecture notes in computer science, vol. 13705. Cham: Springer; 2022. p. 127–35. https://doi.org/10.1007/978-3-031-20627-6_12.
23. Alonso PJG. SETA, a suite-independent agile analytical framework; 2016.
24. Hansen J. Selling the data lakehouse; 2021. https://medium.com/snowflake/selling-the-data-lakehouse-a9f25f67c906. 23.01.2024.
25. Armbrust M, Das T, Sun L, Yavuz B, Zhu S, Murthy M, Torres J, van Hovell H, Ionescu A, Łuszczak A, et al. Delta lake. Proc VLDB Endow. 2020;13:3411–24. https://doi.org/10.14778/3415478.3415560.
26. Harby AA, Zulkernine F. From data warehouse to lakehouse: a comparative review. In: 2022 IEEE international conference on big data (big data). IEEE; 2022. p. 389–95. https://doi.org/10.1109/BigData55660.2022.10020719.
27. Azeroual O, Schöpfel J, Ivanovic D, Nikiforova A. Combining data lake and data wrangling for ensuring data quality in CRIS. Proc Comput Sci. 2022;211:3–16. https://doi.org/10.1016/j.procs.2022.10.171.
28. Eckerson W. All hail, the data lakehouse! (if built on a modern data warehouse); 2020. https://www.eckerson.com/articles/all-hail-the-data-lakehouse-if-built-on-a-modern-data-warehouse. 23.01.2024.
29. Inmon WH, Levins M, Srivastava R. Building the data lakehouse. Basking Ridge: Technics Publications; 2021.
30. Raina V, Krishnamurthy S. Building an effective data science practice. A framework to bootstrap and manage a successful data science practice. Berkeley: Apress L. P; 2022.
31. Feinberg Donald, Russom P, Showell N. Hype cycle for data management 2022. Online (2022).
32. Oracle Corporation: What is a Data Lakehouse? (2023). https://www.oracle.com/big-data/what-is-data-lakehouse/. 23.01.2024
33. Liu G, Pang Z, Zeng J, Hong H, Sun Y, Su M, Ma N. IoT lakehouse: a new data management paradigm for AIoT. In: Zhang S, Hu B, Zhang L-J, editors. Big data—big data 2023. Lecture notes in computer science, vol. 14203. Cham: Springer; 2023. p. 34–47. https://doi.org/10.1007/978-3-031-44725-9_3.
34. Zhang Y, Peng B, Du Y, Su J. GeoLake: bringing geospatial support to lakehouses. IEEE Access. 2023;11:143037–49. https://doi.org/10.1109/ACCESS.2023.3343953.
35. Ait Errami S, Hajji H, Ait El Kadi K, Badir H. Spatial big data architecture: from data warehouses and data lakes to the lakehouse. J Parall Distrib Comput. 2023;176:70–9. https://doi.org/10.1016/j.jpdc.2023.02.007.
36. Vox C, Broneske D, Piewek J, Feigel J, Saake G. Investigating lakehouse-backbones for vehicle sensor data. In: Strauss C, Amagasa T, Kotsis G, Tjoa AM, Khalil I, editors. Database and Expert systems applications. Lecture notes in computer science, vol. 14146. Cham: Springer; 2023. p. 243–58. https://doi.org/10.1007/978-3-031-39847-6_17.
37. Basker E, editor. Handbook on the economics of retailing and distribution. Cheltenham: Edward Elgar Publishing; 2016.
38. Krafft M, Mantrala MK, editors. Retailing in the 21st century. Berlin: Springer; 2010. https://doi.org/10.1007/978-3-540-72003-4.
39. Bhatia SC. Retail management. New Delhi: Atlantic Publ. & Distrib; 2008.
40. Bradlow ET, Gangwar M, Kopalle P, Voleti S. The role of big data and predictive analytics in retailing. J Retail. 2017;93:79–95. https://doi.org/10.1016/j.jretai.2016.12.004.
41. Aktas E, Meng Y. An exploration of big data practices in retail sector. Logistics. 2017;1:12. https://doi.org/10.3390/logistics1020012.
42. Lekhwar S, Yadav S, Singh A. Big data analytics in retail. In: Satapathy SC, Joshi A, editors. Information and communication technology for intelligent systems. Smart innovation, systems and technologies, vol. 107. Singapore: Springer; 2019. p. 469–77. https://doi.org/10.1007/978-981-13-1747-7_45.
43. Fisher M, Raman A. Using data and big data in retailing. Prod Oper Manag. 2018;27:1665–9. https://doi.org/10.1111/poms.12846.

44. Kart L, Linden A, Schulte WR. Extend your portfolio of analytics capabilities. Gartner research note G00254653. Gartner Group, Stamford, CT; 2013.
45. Raorane A, Kulkarni RV. Data mining techniques: a source for consumer behavior analysis. arXiv; 2011.
46. Bounsaythip C, Rinta-Runsala E. Overview of data mining for customer behavior modeling. VTT Inf Technol Res Rep Vers. 2001;1:1–53.
47. Pantano E, Giglio S, Dennis C. Making sense of consumers' tweets. IJRDM. 2019;47:915–27. https://doi.org/10.1108/IJRDM-07-2018-0127.
48. Rambocas M, Pacheco BG. Online sentiment analysis in marketing research: a review. JRIM. 2018;12:146–63. https://doi.org/10.1108/JRIM-05-2017-0030.
49. Langen H, Huber M. How causal machine learning can leverage marketing strategies: assessing and improving the performance of a coupon campaign. PLoS ONE. 2023;18:e0278937. https://doi.org/10.1371/journal.pone.0278937.
50. Mehrotra P, Pang L, Gopalswamy K, Thangali A, Winters T, Gupte K, Kulkarni D, Potnuru S, Shastry S, Vuyyuri H. Price investment using prescriptive analytics and optimization in retail. In: Gupta R, Liu Y, Shah M, Rajan S, Tang J, Prakash BA, editors. Proceedings of the 26th ACM SIGKDD international conference on knowledge discovery & data mining. New York: ACM; 2020. p. 3136–44. https://doi.org/10.1145/3394486.3403365.
51. Ito S, Fujimaki R. Optimization beyond prediction. In: Matwin S, Yu S, Farooq F, editors, Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining. New York: ACM; 2017. p. 1833–41. https://doi.org/10.1145/3097983.3098188.
52. Akidau T, Chernyak S, Lax R. Streaming systems. The what, where, when, and how of large-scale data processing. O'Reilly, Beijing, Boston, Farnham, Sebastopol, Tokyo; 2018.
53. Hai R, Koutras C, Quix C, Jarke M. Data lakes: a survey of functions and systems. IEEE Trans Knowl Data Eng. 2023. https://doi.org/10.1109/TKDE.2023.3270101.
54. Bauer A, Günzel H (eds.) Data-warehouse-Systeme. Architektur, Entwicklung, Anwendung, vol. . dpunkt.verlag, Heidelberg; 2013.
55. The Data Warehouse Lifecycle Toolkit. Expert methods for designing, developing, and deploying data warehouses. New York: Wiley; 1998.
56. Gray J, Bosworth A, Lyaman A, Pirahesh H. Data cube: a relational aggregation operator generalizing GROUP-BY, CROSS-TAB, and SUB-TOTALS. In: Proceedings of the twelfth international conference on data engineering. IEEE Comput. Soc. Press; 1996. p. 152–159. https://doi.org/10.1109/ICDE.1996.492099.
57. Marz N, Warren J. Big data. Principles and best practices of scalable real-time data systems. Manning, Shelter Island, NY; 2015.
58. Gillet A, Leclercq É, Cullot N. Lambda+, the renewal of the lambda architecture: category theory to the rescue. In: La Rosa M, Sadiq S, Teniente E, editors. Advanced information systems engineering. Lecture notes in computer science, vol. 12751. Cham: Springer; 2021. p. 381–96. https://doi.org/10.1007/978-3-030-79382-1_23.
59. Kreps J. Questioning the lambda architecture. The lambda architecture has its merits, but alternatives are worth exploring. Radar/Data (2014). 23.01.2024.
60. Vinoyang: Incremental Processing on the Data Lake; 2020. https://hudi.apache.org/blog/2020/08/18/hudi-incremental-processing-on-data-lakes/. 23.01.2024.
61. Apache Spark Documentation: SQL Syntax. DML Statements; 2023. https://spark.apache.org/docs/latest/sql-ref-syntax.html#dml-statements. 23.01.2024.
62. Apache Flink Documentation: Table API & SQL. SQL: UPDATE Statements; 2023. https://nightlies.apache.org/flink/flink-docs-release-1.17/docs/dev/table/sql/update/. 23.01.2024.
63. Apache Spark Documentation: Structured Streaming Programming Guide. Output Modes; 2023. https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html. 23.01.2024.
64. Apache Spark Documentation: Spark SQL, DataFrames and Datasets Guide. Data Sources (2023). https://spark.apache.org/docs/latest/sql-programming-guide.html. 23.01.2024.
65. Apache Spark Documentation: Generic Load/Save Functions. Save Modes; 2023. https://spark.apache.org/docs/latest/sql-data-sources-load-save-functions.html#save-modes. 23.01.2024.
66. Apache Hadoop Documentation: Introduction. Object Stores vs. Filesystems; 2023. https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-common/filesystem/introduction.html#Core_Expectations_of_a_Hadoop_Compatible_FileSystem. 23.01.2024.
67. Apache Hadoop Documentation: Hadoop-AWS module: Integration with Amazon Web Services. Warnings. Warning #1: Directories are mimicked; 2023; https://hadoop.apache.org/docs/stable/hadoop-aws/tools/hadoop-aws/index.html. 23.01.2024.
68. Sandre, Scott and Lee, Denny and Kryński, Mariusz: Multi-cluster writes to Delta Lake Storage in S3; 2022. https://delta.io/blog/2022-05-18-multi-cluster-writes-to-delta-lake-storage-in-s3/. 23.01.2024.
69. Apache Parquet Documentation: File Format. Types; 2022. https://parquet.apache.org/docs/file-format/types/. 23.01.2024.
70. Apache ORC Documentation: Types; 2023. https://orc.apache.org/docs/types.html. 23.01.2024.
71. Vaisman A, Zimányi E. Data warehouse systems: design and implementation. London: Springer; 2022.
72. Zheng JG. Data visualization in business intelligence. In: Munoz JM, editor. Global business intelligence. Routledge studies in international business and the world economy. New York: Routledge; 2017. p. 67–82.
73. Pendse N, Creeth R. The OLAP report. Business Intelligence; 1995.
74. Han J, Pei J, Tong H. Data mining. Concepts and techniques. Amsterdam: Morgan Kaufmann Publishers; 2022.
75. Zhou Z-H. Machine learning. London: Springer; 2021.
76. Gröger C, Schwarz H, Mitschang B. The manufacturing knowledge repository. In: Proceedings of the international conference on enterprise information systems (ICEIS); 2014. p. 39–51.
77. Kejariwal A, Kulkarni S, Ramasamy K. Real time analytics: algorithms and systems. arXiv; 2017.
78. Zaidi E, de Simoni G, Edjlali R, Duncan AD. Data catalogs are the new black in data management and analytics; 2017. https://www.gartner.com/en/documents/3837968. 23.01.2024.
79. Eichler R, Gröger C, Hoos E, Stach C, Schwarz H, Mitschang B. Introducing the enterprise data marketplace: a platform for democratizing company data. J Big Data. 2023. https://doi.org/10.1186/s40537-023-00843-z.
80. Singh T, Gupta S, Satakshi, Kumar M. Performance analysis and deployment of partitioning strategies in Apache Spark. Proc Comput Sci. 2023. https://doi.org/10.1016/j.procs.2023.01.041.
81. Codd EF. The relational model for database management. Version 2. Reading: Addison-Wesley; 1990.
82. Kumar D, Li S. Separating storage and compute with the databricks lakehouse platform. In: 2022 IEEE 9th international conference on data science and advanced analytics (DSAA). IEEE; 2022. p. 1–2. https://doi.org/10.1109/DSAA54385.2022.10032386.
83. L'Esteve R. The Azure data lakehouse toolkit. Berkeley: Apress; 2022.

84. Begoli E, Goethert I, Knight K. A lakehouse architecture for the management and analysis of heterogeneous data for biomedical research and mega-biobanks. In: 2021 IEEE international conference on big data (big data). IEEE; 2021. p. 4643–51. https://doi.org/10.1109/BigData52589.2021.9671534.
85. Ren P, Li S, Hou W, Zheng W, Li Z, Cui Q, Chang W, Li X, Zeng C, Sheng M, et al. MHDP: an efficient data lake platform for medical multi-source heterogeneous data. In: Xing C, Fu X, Zhang Y, Zhang G, Borjigin C, editors., et al., Web information systems and applications. Lecture notes in computer science, vol. 12999. Cham: Springer; 2021. p. 727–38. https://doi.org/10.1007/978-3-030-87571-8_63.
86. Park S, Yang C-S, Kim J. Design of vessel data lakehouse with big data and AI analysis technology for vessel monitoring system. Electronics. 2023;12:1943. https://doi.org/10.3390/electronics12081943.
87. Ormenisan AA, Meister M, Buso F, Andersson R, Haridi S, Dowling J. Time travel and provenance for machine learning pipelines. In: 2020 USENIX conference on operational machine learning (OpML 20). USENIX Association; 2020.
88. Tovarnak D, Racek M, Velan P. Cloud native data platform for network telemetry and analytics. In: 2021 17th international conference on network and service management (CNSM). IEEE; 2021. p. 394–6. https://doi.org/10.23919/CNSM52442.2021.9615568.
89. Weikum G, Vossen G. Transactional information systems, theory, algorithms, and the practice of concurrency control and recovery. London: Elsevier; 2001.
90. Jain P, Kraft P, Power C, Das T, Stoica I, Zaharia M. Analyzing and comparing lakehouse storage systems. In: Proceedings of the 13th annual conference on innovative data systems research; 2023.
91. LakeSoul Documentation: LakeSoul Introduction; 2023. https://lakesoul-io.github.io/docs/intro. 23.01.2024.
92. Camacho-Rodríguez J, Agrawal A, Gruenheid A, Gosalia A, Petculescu C, Aguilar-Saborit J, Floratou A, Curino C, Ramakrishnan R. LST-bench: benchmarking log-structured tables in the cloud. arXiv; 2023.
93. Weller K. Apache Hudi vs. Delta Lake vs. Apache Iceberg. Lakehouse feature comparison; 2023. https://www.onehouse.ai/blog/apache-hudi-vs-delta-lake-vs-apache-iceberg-lakehouse-feature-comparison. 23.01.2024.
94. Belov V, Nikulchev E. Analysis of big data storage tools for data lakes based on apache hadoop platform. IJACSA. 2021. https://doi.org/10.14569/IJACSA.2021.0120864.
95. Project Nessie Documentation: Features; 2023. https://projectnessie.org/features/. 23.01.2024.
96. Project Nessie Documentation: Architecture; 2023. https://projectnessie.org/develop/. 23.01.2024.
97. Apache Hive Documentation: Hive Transactions; 2023. https://cwiki.apache.org/confluence/display/Hive/Hive+Transactions. 23.01.2024.
98. Marcel K, Alexander B, Victor B, Taras B, Casey C, Alan C, Justin E, Martin G, Daniel H, Matthew J, et al. Impala: a modern, open-source SQL engine for hadoop. In: Conference on innovative data systems research; 2015.
99. Apache Kudu Documentation: Frequently Asked Questions; 2023. https://kudu.apache.org/faq.html. 23.01.2024.
100. Apache Impala Documentation: SQL Reference; 2023. https://impala.apache.org/docs/build/asf-site-html/topics/impala_langref.html. 23.01.2024.
101. Apache Impala Documentation: Iceberg Tables; 2023. https://impala.apache.org/docs/build/html/topics/impala_iceberg.html. 23.01.2024.
102. Sethi R, Traverso M, Sundstrom D, Phillips D, Xie W, Sun Y, Yegitbasi N, Jin H, Hwang E, Shingte N, et al. Presto: SQL on everything. In: 2019 IEEE 35th international conference on data engineering (ICDE). IEEE; 2019. p. 1802–13. https://doi.org/10.1109/ICDE.2019.00196.
103. Chen F, Yan Z, Gu L. Towards low-latency big data infrastructure at Sangfor. In: Chen J, He D, Lu R, editors. Emerging information security and applications. Communications in computer and information science, vol. 1641. Cham: Springer; 2022. p. 37–54. https://doi.org/10.1007/978-3-031-23098-1_3.
104. Dremio Documentation: Querying Your Data. Querying Files and Directories; 2023. https://docs.dremio.com/current/sonar/query-manage/querying-data/files-and-directories/. 23.01.2024.
105. Trino Documentation: Connectors. Hive Connector; 2023. https://trino.io/docs/current/connector/hive.html. 23.01.2024.
106. Dremio Documentation: Connecting to Your Data. Object Storage; 2023. https://docs.dremio.com/software/data-sources/object-storage/. 23.01.2024.
107. Dremio Documentation: SQL Reference. SQL Commands Reference; 2023. https://docs.dremio.com/current/reference/sql/commands/. 23.01.2024.
108. Dremio Documentation: Data Formats. Apache Iceberg; 2023. https://docs.dremio.com/software/data-formats/apache-iceberg/. 23.01.2024.
109. Dremio Documentation: SQL Commands. SQL Commands for Apache Iceberg Tables; 2023. https://docs.dremio.com/current/reference/sql/commands/apache-iceberg-tables/. 23.01.2024.
110. Trino Documentation: Connectors. Delta Lake Connector; 2023. https://trino.io/docs/current/connector/delta-lake.html. 23.01.2024.
111. Trino Documentation: Connectors. Hudi Connector; 2023. https://trino.io/docs/current/connector/hudi.html. 23.01.2024.
112. Trino Documentation: Connectors. Iceberg Connector; 2023. https://trino.io/docs/current/connector/iceberg.html. 23.01.2024.
113. Malone J. Iceberg tables: powering open standards with snowflake innovations; 2022. https://www.snowflake.com/blog/iceberg-tables-powering-open-standards-with-snowflake-innovations. 23.01.2024.
114. Snowflake Documentation: Databases, Tables, & Views. External Tables; 2023. https://docs.snowflake.com/en/user-guide/tables-external-intro. 23.01.2024.
115. Snowflake Documentation: Integrating Apache Hive Metastores with Snowflake; 2023. https://docs.snowflake.com/en/user-guide/tables-external-hive. 23.01.2024.
116. Ortloff, Ron and Herbert, Steve: Unifying Iceberg Tables on Snowflake; 2023. https://www.snowflake.com/blog/unifying-iceberg-tables. 23.01.2024.
117. Chen Z, Shao H, Li Y, Lu H, Jin J. Policy-based access control system for delta lake. In: 2022 10th international conference on advanced cloud and big data (CBD). IEEE; 2022. p. 60–65. https://doi.org/10.1109/CBD58033.2022.00020.
118. Ma C, Hu X. A data analysis privacy regulation compliance scheme for lakehouse. In: Proceedings of the 2023 2nd international conference on algorithms, data mining, and information technology. New York: ACM; 2023. p. 1–5. https://doi.org/10.1145/3625403.3625405.
119. Zouari F, Ghedira-Guegan C, Boukadi K, Kabachi N. A semantic and service-based approach for adaptive mutli-structured data curation in data lakehouses. World Wide Web. 2023;26:4001–23. https://doi.org/10.1007/s11280-023-01218-3.