Data source connectors layer as a service - design patterns

industrial experience report

Michał Bodziony
IBM Poland, Software Lab Kraków
Kraków, Poland
michal.bodziony@pl.ibm.com

Robert Wrembel
Poznan University of Technology
Poznań, Poland
robert.wrembel@put.poznan.pl

ABSTRACT

In data integration architectures connectors to data sources are key components. Traditionally, connectors are organized into a library of connectors. Such an approach has various drawbacks, which we discuss in this paper. In order to mitigate these drawbacks, we propose to implement the *library of connectors as a service* (*LCS*) by means of design patterns. These patterns summarize the experience gained over years, while designing integration architectures in R&D projects run at IBM Poland.

KEYWORDS

data integration, common connectivity layer, design patterns

1 INTRODUCTION

A *common connectivity layer* (CCL) is a software component that allows different systems or applications to communicate with heterogeneous data sources in a unified way. It provides a standard interface for sending and receiving data and it acts as a bridge between different systems that may use different protocols or data formats. One of the well-known examples of a CCL are database connectors (a.k.a. drivers).

One of the most common forms of a CCL is a *library of connectors* (LC). These connectors follow an adapter pattern, which translates a native interface of a data source into a common interface of a CCL. Connectors designed as libraries have some significant limitations, like poor maintainability, limited scalability, and challenging security (cf. Section 3.1). There are many functional and non-functional requirements for a CCL, which are difficult to be met when a CCL is implemented as a LC (these requirements are described in more details in Section 3.2).

In this paper, we propose to build a **LC** used **as a service** (denoted as *LCS*), rather than as an embedded library. To this end we apply design patterns. With these patterns many requirements (cf. Section 3.2) can be more easily and clearly addressed. The proposed patterns summarize the experience gained over years while designing connectivity architectures of products like IBM Cloud Pak for Data.

2 RELATED WORK ON A CONNECTIVITY LAYER

In this section, we outline the fundamental data integration technologies, including: (1) virtual, i.e., federated and mediated, (2) physical, i.e., a data warehouse and a data lake, and (3) a data mesh and a data fabric.

In early 80s, two fundamental virtual data integration architectures were developed, namely *federated* [4] and *mediated* [9]. Both of them share a common feature of storing data only in data sources (DSs), which are typically heterogeneous and distributed. These data are integrated on demand by a software layer located between a user and DSs.

One of the most popular approaches to physical data integration is a data warehouse architecture [8], where data from distributed and heterogeneous sources are ingested and integrated by means of data integration processes, commonly known as ETL processes [1]. Integrated data are then stored in a data repository called a *data warehouse* (DW).

Another approach to a unified access to huge volumes of data is a data lake architecture. A *data lake* (DL) is a large, repository capable of integrating, storing, and processing data of arbitrary complexities and sizes [2, 6]. DLs are typically built on a distributed file system and can store data in arbitrary formats. DLs are often used in conjunction with physical or virtual data warehouses, where a DL is serving as a repository for raw data and a DW is providing a structured view on the data.

In recent years, another technological concept was coined - it is called a *data mesh* (DM). It is a data architecture and data governance approach that promotes the decentralization of data ownership within an organization [5]. It is based on the idea that data is a shared asset that should be owned and governed by teams with domain expertise, rather than being controlled by a central data organization.

A concept strongly related to a data mesh is called a *data fabric*. It is used to describe a data architecture for supporting the flow of data within an organization [7]. By default, the architecture should be flexible, scalable, and resilient, with a goal to make data available for multiple teams and systems across an organization.

3 COMMON CONNECTIVITY LAYER CHALLENGES

An often-used approach to implement a common connectivity layer is the aforementioned library of connectors (LC). This library is a collection of software components that are used to connect to and to interact with various types of data sources. A data source connector is a specific type of software that provides a set of APIs or other programming interfaces that an application can use to manipulate the content of a data source.

A LC typically includes connectors to a variety of different types of data sources, such as relational databases, NoSQL databases, file systems, and cloud storage platforms. Connectors in a LC are typically designed to be used in a standardized way and may include features such as pooling and load balancing, to help improve the performance and scalability of an application.

3.1 Limitations of a library of connectors

In this section we outline some drawbacks of using a LC.

Limited support for specific data sources: a LC may not include connectors for every possible type of a data source that an organization might need to use. In some cases, a data source is supported by one library but is not supported by another. Combining a few different libraries in the same integration software is usually problematic because of different interfaces and different prerequisites.

Reduced flexibility: by using a LC, an organization may be limited to the capability that is provided by connectors in the library. This can make more difficult to fully utilize features specific to one or the other data sources.

Dependency on third-party software limitations: a LC is typically developed and maintained by third parties, which means that the organization using the library is dependent on the vendor to fix a problem or to provide updates and new features.

Performance and scalability limitations: connectors in a LC may not be optimized for every possible use case, and may not provide the best performance or scalability for a particular application. In some cases, it would be efficient to combine two or more different implementations of the same connector, in order to achieve the required performance and scalability.

Complexity limitations: if a LC is used by many services/ applications/ processes, the library has to be embedded in many places. It results in a large number of interactions and dependencies that are hard to control. As presented in Fig. 1, each application embedding the LC has a direct dependency on all the services required by the LC. High design complexity can lead to issues such as increased development time, higher maintenance costs, and a higher likelihood of misconfigurations or bugs.

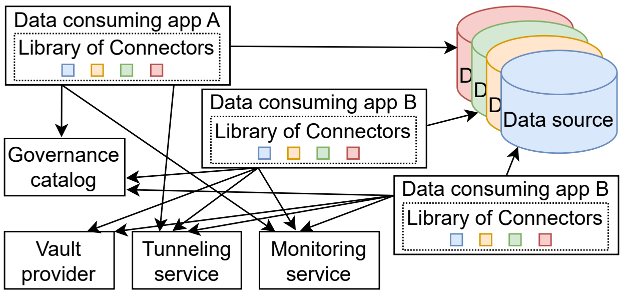

Figure 1: Complexity of integration of data sources with a library of connectors

Dependencies explosion: in a more mature solution, data source connectors depend on many other software components (e.g., data governance catalog, vault providers, tunneling providers, governance policy repositories). In micro-services architectures, usually plenty of services need access to data. Each such service has to embed a LC and therefore has to inherit all the dependencies to other services.

Maintenance costs: data source connectors have to be patched or upgraded regularly (e.g., security patches, new capabilities, licensing). If a LC is used in multiple services, then it has to be maintained in all those services. This often means that services that embed a LC have to be recycled after connectors were upgraded or patched.

Reduced portability: a LC is usually available in a single programming language but in micro-services architectures, services of the same functionality may be written in different programming languages. In order to have all the data source types supported by all the services, one would need data source connectors to be re-implemented for all the languages used by client services.

3.2 Requirements for common connectivity layer

There are several types of requirements to be met by a well-designed CCL. These requirements are especially important when a solution is in line with the principles of micro-services architectures. In this section, we outline these requirements.

Portability. Software components portability is a feature that refers to the degree to which software components can be migrated between platforms. The following aspects of portability can be essential for a connectivity layer: adaptability, replaceability, and installability.

Security. A CCL typically plays a crucial role in ensuring a secure access to data sources that often contain confidential data. The most important aspects of security in a CCL include: encryption, authentication, access control, data integrity, non-repudiation (auditing), and denial of service protection.

Scalability. For a CCL, scalability refers to the ability of the layer to handle increasing amounts of data and/or users without experiencing a decrease in the quality of data access (throughput, delays, and a number of errors). A CCL can be usually scaled in two dimensions: vertical and horizontal.

Reliability. Reliability refers to the ability of a CCL to enable interactions with data consistently and without errors. A CCL that is reliable will be able to handle a wide range of inputs and conditions without failing or producing incorrect output. There are a few factors that can affect the reliability of the layer, namely: robustness, error handling, testing, and documentation.

Usability. Usability is a measure of how easy a system or product can be used. In the context of a CCL, good usability would involve making it easy for users to work with data but also easy for developers to integrate their applications with data sources. There are a few important aspects of the CCL usability, namely: simplicity, consistency, flexibility, and feedback.

Performance. A CCL with good performance is able to handle large data volumes and/or a large number of user requests without experiencing a significant decrease in throughput or stability. There are a few factors that can affect the performance of the layer, including: data volume, the number of users, network latency, and hardware.

4 LIBRARY OF CONNECTORS AS A SERVICE

Many data integration solutions nowadays are designed as micro-services architectures. This motivates to design a CCL as a single service or plurality of services with common interfaces.

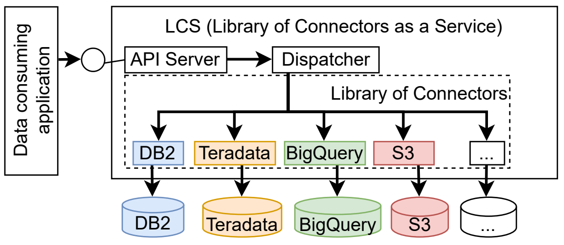

Figure 2: *LCS* - the library of connectors as a service

A *library of connectors service* (*LCS*) is composed of a LC and at least two facilitating components (as depicted in Fig. 2). Connectors from the LC are responsible for adapting native interfaces of data sources into a common interface. An *API Server* externalises this common interface to *data consuming applications*. A *dispatcher* is responsible for instantiating a connector of a given type and forwarding request to this connector.

There may exist multiple services that serve the *LCS* functionality. These services can be implemented to ensure scalability, workloads isolation (security), performance optimization, and workload isolation (monitoring). If multiple services provide connectors capabilities, one could consider integrating them together into a *chain of responsibilities design patterns* combined with a *command design pattern* (c.f., Section 5 for more detailed examples of combining a *LCS*).

5 APPLICABLE DESIGN PATTERNS

In this section we contribute core design patterns that support designing the common connectivity layer as the *library of connectors as a service.*

5.1 Integration with data governance catalog

A *data governance catalog* (DGC) is a centralized repository of metadata about an organization's data assets. It can include metadata about: (1) data sources that are used within an organization, (2) data structures and relationships that are defined, (3) business terms and definitions that are used to describe data, (4) policies and procedures that govern the use of and access to data, (5) data lineage and flow, as well as (6) any metadata or annotations that have been added to data. Yet another metadata often maintained in DGCs are properties, which are necessary to establish a connection to DSs (a.k.a. connection strings, database URLs, or connection assets). When a user wants to access data from any governed data asset, he/she needs to know only an ID of the data asset in a DGC (of course, access to the DGC itself has to be properly authorized).

In the proposed *LCS* architecture, a user who wants access data sends a request to the *LCS*. The request contains only an ID of a data asset in a DGC and an instruction how to interact (e.g., read, write, filter, degree of parallelism) with the asset. A user does not need to provide details like connection strings, table names, paths, schemas, credentials. It significantly increases user experience in accessing data. Since a user does not have to deal with DS credentials, the overall security can be increased as well. It is the *LCS* that retrieves all the information necessary to establish connections (including credentials).

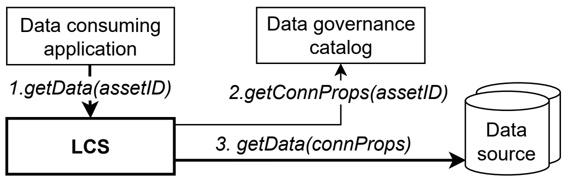

Figure 3: The integration with a *data governance catalog pattern*

As shown in Fig. 3, an application interacts with a DS by sending a request to the *LCS* (cf. step 1). This request identifies a subject data asset by its *assetID*. In step 2, *assetID* is fetched and transformed into connection properties. Then, a specific connector is fed with the connection properties in order to establish a connection to the data source and retrieve data (cf. step 3).

To sum up, there are some advantages of using this approach. First, a system complexity is reduced by limiting a dependency of a consuming application to the *LCS* only. Second, a system security is improved by minimising a set of services with access to data sources credentials (subset of connection properties). Third, a user experience is improved by accessing only assets IDs rather than all assets connection metadata.

5.2 Integration with a vault provider

Vault providers are often used in enterprise environments to store and manage secrets that are used by applications and services. For example, a vault provider might be used to store database passwords, API keys, or encryption keys, which are used by various applications and services within a company.

In some implementations of data governance catalogs, a management of credentials can be delegated to a vault provider. In this case, data governance maintains only a reference to secrets in a vault. This reference needs to be resolved to a secret retrieved from the vault, in order to establish a connection to a data source.

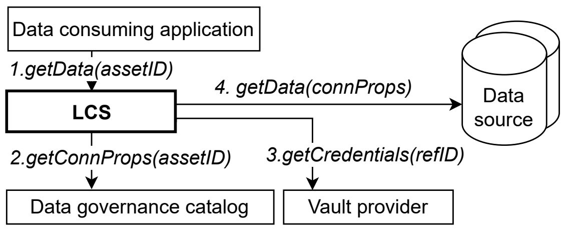

Figure 4: The pattern for integrating with a vault provider

A design pattern proposed in Fig. 4 decouples data consumers from vault providers. It is a *LCS* that retrieves all necessary secrets/credentials from a vault provider and applies them to establish a connection to a DS. In the case of a dynamic security (e.g., one-time passwords) it is the *LCS* that ensures that every new connection is established with fresh credentials retrieved from a vault.

Applying the discussed pattern allows to: (1) reduce complexity by limiting dependency only on a vault provider; (2) increase security by minimizing a set of services that request access to the vault provider; (3) enhance functionality, since dynamic security is handled by a *LCS*; (4) increase portability by a simple replacement of a vault provider.

5.3 Tunneling

In a networking context, tunneling refers to the practice of encapsulating one network protocol inside another protocol. Tunneling can be used to connect to a DS in situations where it is not possible to connect directly to the DS. This might be because the DS is behind a firewall that blocks incoming connections. A client can then use this connection to send commands to the DS as if it were directly connected to the DS.

With the proposed design pattern (as shown in Fig. 5), the *LCS* ensures a proper configuration of tunneling, necessary to establishing a secure connection to a DS specified in a request. Metadata necessary to configure such a tunnel can be maintained in a DGC and referred to by data assets. The other responsibility of the *LCS* can be: purging unused tunnels, refreshing credentials or certificates used by existing tunnels.

The proposed design pattern offers the following advantages: (1) reduced complexity by limiting the number of services directly depending on a tunneling service; (2) improved security by minimizing a set of services with an access to a tunneling service and maintaining a single path of tunneling management; (3)

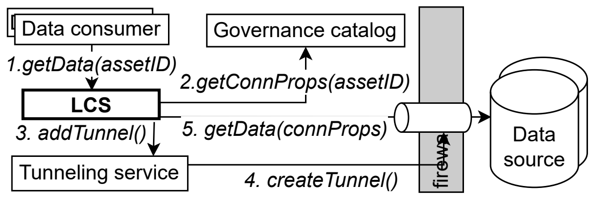

Figure 5: The integration with the tunneling services pattern

improved performance by maintaining tunnels pools and tunnels re-using for a plurality of connections.

5.4 Custom drivers

There are cases where a user needs to plug in some specific database drivers, e.g., custom JDBC drivers. However, there are some risks (like compatibility, security, performance) that have to be properly addressed when creating an extensible system that allows users to add a custom code.

The pattern shown in Fig. 6 proposes that a custom driver is wrapped with a service (further called *custom LCS*). The custom LCS is chained with the LCS. If a request is targeted to a data source supported by the custom LCS, then the LCS delegates the request to the custom LCS. The custom service uses the same interface and data format as the LCS.

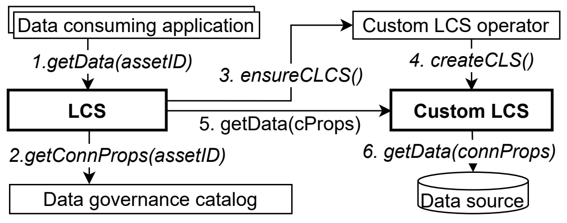

Figure 6: The pattern for extensibility with custom drivers

The pattern offers two main advantages, namely: (1) improved security by a maximal isolation of custom code; (2) improved performance and security by ensuring that only pre-defined quota of resources (e.g., computing units, memory, network) can be consumed by the custom LCS.

5.5 Data locality

Data locality refers to the concept of storing data in a close proximity to where they are used. This can improve performance of applications because it reduces time and resources required to access the data. There are a few different types of data locality, including: (1) spatial locality, typically in distributed systems, when data are accessed from nearby memory locations, (2) temporal locality, when data are accessed multiple times in a short period of time, and (3) cache locality, when data are stored in cache memory.

A promising technique used to achieve data locality is the so-called *push down optimisation* [3]. Push down consists in moving the most selective tasks (typically queries) towards DSs.

In some implementations, a DS connector can offer processing that reduces the footprint of returned data. It is obviously better to have such processing closer to a DS. If connectors are wrapped with a service, then it is easier to provision them collocated with a DS (cf. Fig. 7).

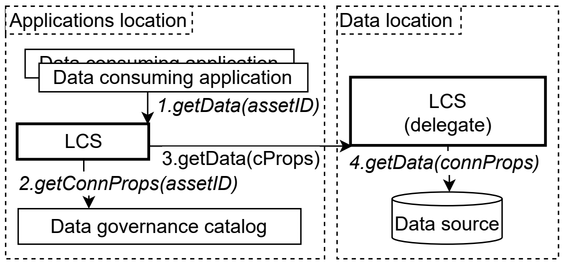

Figure 7: The pattern for data and computing co-location

The presented pattern offers the following advantages: (1) improved performance by co-locating a DS connector with a data source; (2) improved performance by reducing a volume of data to be transferred between a DS and a client application; (3) improved security by enabling push-down of masking, encryption, or other security related techniques.

5.6 Policy enforcement

In the context of data governance, *policy enforcement* refers to the process of ensuring that data policies and rules are obeyed within an organization. Data governance policies typically cover a wide range of topics, including data quality, security, privacy, retention, and access.

There are several different ways that a policy enforcement can be implemented as part of data governance. For example, an organization may use automated tools for monitoring data access and usage as well as for enforcing policies around data retention and data privacy. Mechanisms for centralized or federated policy enforcement can be part of a data fabric architecture.

Examples of policies, which can be enforced on data access are: denying some data for certain roles, masking data and data anonymisation, encryption, pre-processing, and cleaning.

In the proposed architecture, a LCS is a central component, enhanced with capabilities of policies enforcement. The design pattern proposed here adds a proxy service in front of the LCS. The proxy is responsible for loading policy rules evaluation and enforcing the policies on data transferred to/from a DS.

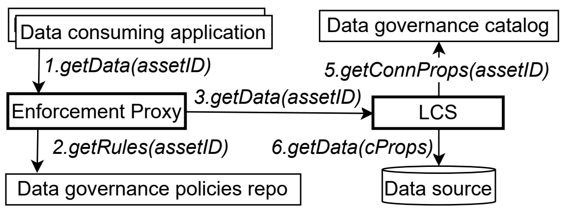

Figure 8: The pattern for federated policies enforcement

Advantages of the pattern for federated policies enforcement include: (1) improved reliability by centralizing enforcement of global and federated policies; (2) improved maintainability by reducing dependencies between a repository of policies and components of a system where the policies are enforced.

5.7 Data access monitoring and auditing

Data access monitoring and auditing refers to the process of tracking and recording access to data in a system, e.g., who and when is accessing data, what data are accessed, and what commands are executed. There are different ways that data access monitoring and auditing can be implemented, including the use of logging, auditing, and monitoring tools, as well as the implementation of access controls and user authentication systems.

Monitoring and auditing is very challenging if data source connectors are implemented as libraries, since a monitoring capability has to be part of a library. Every host of such a library needs to be configured for monitoring (e.g., a granularity of monitoring or access to a repository that stores monitoring data).

In the proposed LCS architecture, monitoring can be again implemented as a proxy service in front of the LCS. The proxy sends notifications to a *monitoring service* that describes the context of requests handled by the LCS. The *monitoring service* combines information from the proxy with metrics collected in continuous way (represented by symbol ∞ in Fig. 9).

Yet another interesting pattern is to have a dedicated LCS for every connection. This way, every data processing executed by a connector are encapsulated by its own process/ container and can be monitored as a black box. Detailed resource utilization can be measured for data processing. The results of such monitoring can be used afterwards for modeling profiles of data processing or identifying common patterns of data access.

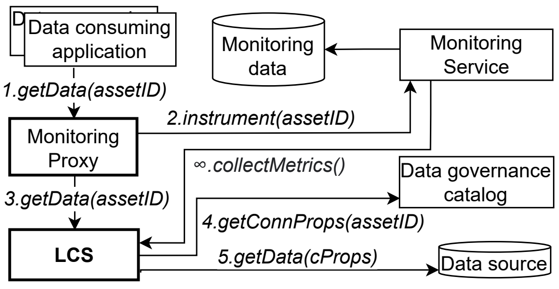

Figure 9: The centralized monitoring and auditing pattern

The presented pattern offers the following advantages: (1) improved maintainability, since there is only one component in a system where monitoring is configured; (2) improved security by having a reliable auditing of data access events; (3) enhanced performance monitoring by having a very detailed resource consumption measuring.

5.8 Chain of responsibility with bypassing

A design pattern called a *chain of responsibility* is used to process requests or handle events in a decoupled way. In this pattern, a request is passed through a chain of services, each of which has the opportunity to process the request or pass it on to the next component in the chain.

The patterns presented in the previous sections can be combined together, which produces longer chains (e.g., Fig. 10). Having such a longer chain of responsibilities brings all the benefits of each individual pattern, but it may negatively affect performance. A performance penalty is caused by adding extra hops in data access request handling, but the penalty can be reduced by the approach proposed below.

In many cases, access to data is done in two phases, namely: (1) data preparation (e.g., preparing statements in JDBC or getting flight info in Apache Arrow Flight) and (2) data access itself. Let us consider a preparation phase (steps 1-4 in Fig. 10) followed by a data access phase (step 5). Although the preparation phase runs through the whole chain, the data access phase can bypass some nodes. For example, in Fig. 10 monitoring, enforcing, and the main LCS are bypassed in step 5, which directly leads (connects) to the LCS co-located with the data source.

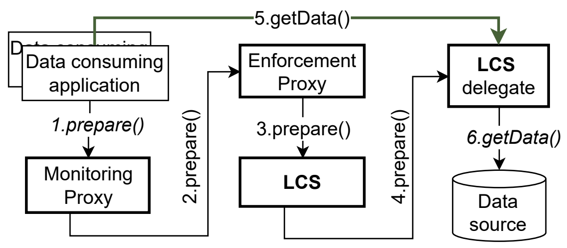

Figure 10: An exemplary chain of responsibilities

6 SUMMARY

The existing implementations of a CCL have significant limitations. By identifying these limitations (cf. Section 3.1) and by specifying a comprehensive set of requirements for good and modern CCL (cf. Section 3.2) we were able to work out a *set of design patterns* for common use cases (cf. Section 5).

Designing data source connectors as a service brings all the benefits of a service-oriented architecture, including loose-coupling, modularity, versioning, availability, security, and scalability. Most of the proposed patterns and ideas have already been deployed in the architecture of the common connectivity layer of IBM Cloud Pak for Data.

Acknowledgements. The work of Michał Bodziony is related to his employment at IBM Polska Sp z o.o. Additionally, his work is supported by the Applied Doctorate grant no. DWD/4/24/2020 from the Polish Ministry of Education and Science. The work of Robert Wrembel is supported by IBM Shared University Reward 2019.

REFERENCES

[1] Syed Muhammad Fawad Ali and Robert Wrembel. 2017. From conceptual design to performance optimization of ETL workflows: current state of research and open problems. *The VLDB Journal* 26, 6 (2017), 777–801.
[2] Antonia Azzini, Sylvio Barbon Jr., Valerio Bellandi, Tiziana Catarci, Paolo Ceravolo, Philippe Cudré-Mauroux, Samira Maghool, Jaroslav Pokorný, Monica Scannapieco, Florence Sèdes, Gabriel Marques Tavares, and Robert Wrembel. 2021. Advances in Data Management in the Big Data Era. In *Advancing Research in Information and Communication Technology - IFIP's Exciting First 60+ Years, Views from the Technical Committees and Working Groups (IFIP AICT)*, Vol. 600. Springer, 99–126.
[3] Michal Bodziony, Rafal Morawski, and Robert Wrembel. 2022. Evaluating pushdown on NoSQL data sources: experiments and analysis paper. In *Int. Workshop on Big Data in Emergent Distributed Environments (BiDEDE), in conjunction with the ACM SIGMOD/PODS Conference.* ACM, 4:1–4:6.
[4] A. Bouguettaya, B. Benatallah, and A. Elmargamid. 1998. *Interconnecting Heterogeneous Information Systems.* Kluwer Academic Publishers, ISBN 0792382161.
[5] Zhamak Dehghani. 2022. *Data Mesh: Delivering Data-Driven Value at Scale.* O'Reilly, ISBN 1492092398.
[6] Fatemeh Nargesian, Erkang Zhu, Renée J. Miller, Ken Q. Pu, and Patricia C. Arocena. 2019. Data Lake Management: Challenges and Opportunities. *VLDB Endowment* 12, 12 (2019), 1986–1989.
[7] Piethein Strengholt. 2023. *Data Management at Scale: Modern Data Architecture with Data Mesh and Data Fabric.* O'Reilly, ISBN 1098138864.
[8] Alejandro A. Vaisman and Esteban Zimányi. 2022. *Data Warehouse Systems - Design and Implementation.* Springer, ISBN 3662651661.
[9] Gio Wiederhold. 1992. Mediators in the Architecture of Future Information Systems. *Computer* 25, 3 (1992), 38–49.