On Developing Data Connectivity Services for Industrial Applications

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

Data source connectors are core components of any data integration architecture. Typically, they are deployed as libraries of connectors. Such deployment exposes some significant drawbacks, like poor maintainability, limited scalability, limited performance, and challenging security. To mitigate them, we propose to organize connectors as a *library of connectors* that is used *as a service* (*LCaaS*). In this paper, we discuss design patterns of the *LCaaS* that allow to ease connectors maintenance, enhance data access security, and increase performance.

Keywords: data integration, data source connectors, data connectivity service, data fabric

1. Introduction

Data integration (DI) has been researched since the 1960s, gaining its momentum in the 2000s in the context of extract-transform-load processes in data warehouse architectures [13, 18]. This interest keeps growing alongside big data, advanced analytics, and machine learning, e.g., [10, 11]. Various DI architectures have been developed for business applications (see Section 2). Data source (DS) connectors are integral components of any DI architecture, offering standardized interfaces for data exchange between heterogeneous systems. These connectors, available as a *library of connectors* (LC), facilitate access to DS. However, a LC has limitations such as poor maintainability, scalability, performance, and security, as discussed in [2].

Business applications impose functional and non-functional requirements for DS connectors, which are difficult to meet by a LC. The requirements include: (1) *portability*, which refers to the degree to which software components can be migrated between platforms; (2) *security*, w.r.t. encryption, authentication, access control, data integrity, and auditing; (3) *scalability*, which refers to the ability of an integration layer to offer acceptable throughput for heavy workloads; (4) *reliability*, which refers to the ability of an integration layer to provide correct data; (5) *usability*, which reflects how easily an integration layer (connectors) can be deployed and used; (6) *performance*, w.r.t. handling large data volumes and/or a large number of user requests without experiencing a significant decrease in throughput or stability.

From our experience in running business projects on data integration, we conclude that connectors deployed as a LC often do not fulfill the aforementioned requirements. For this reason, we opt for building the LC *as a service* (denoted as *LCaaS*). The *LCaaS* allows to apply multiple design patterns that implement the aforementioned requirements.

This paper includes an extension to our prior work [2], where we addressed design patterns for security, usability, and some aspects of scalability. From the Design Science Research [4] point of view the problem identification has already been covered in [2]. **This paper covers design and instantiation, enhanced by their evaluations**. the assessment of instantiation is mostly focused on performance, as performance is the only aspect of the proposed architecture

that can be potentially negatively affected. Usually, adding extra hops in data processing introduced by the chain of responsibility degrades some aspects of performance (e.g., latency), therefore we focused in this paper on evaluating performance impact mostly.

In Tab. 1 we briefly compare: (1) the standard approach to designing a connectivity layer, i.e., libraries of DS connectors with (2) the approach that was proposed in this paper, i.e., the *Library of Connectors as a Service*. Applying the *LCaaS* approach 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 applied in the architecture of the common connectivity layer of IBM Cloud Pak for Data.

Table 1. Feature comparison of: (1) *connectors as a library* and (2) the *Library of Connectors as a Service* (*LCaaS*)

DS connectors: feature comparison		
Feature	as a library	*LCaaS*
Portability	limited	high
Compatibility	limited	high
Security risks	significant	reduced
Supportable	complex	simplified
Scalability	limited	high

DS connectors: feature comparison		
Feature	as a library	*LCaaS*
Re-usability	limited	high
Maintainability	complex	simplified
Reliability	reduced	high
Usability	complex	simplified
Performance	high	high

Further in this paper we outline data integration architectures (Section 2), discuss drawbacks of organizing connectors as LCs (Section 3, present our approach to managing DS connectors (Sections 4 and 5), and present results of its experimental evaluation (Section 6).

2. Data integration

Over the years the following architectures were used in research and industry in reference to data integration: a federated system, a mediated system, a data warehouse, a data lake, and recently developed - a data fabric. Overviews of these DI architectures can be found in [8, 13, 17].

In 80s, virtual data integration architectures were developed, namely *federated* [3, 6] and *mediated* [16]. Both of them share a common feature of storing data only in (DSs), which are typically heterogeneous and distributed. These data are integrated on demand by a software layer located between a user and the DSs.

An industry-standard *data warehouse* architecture [15] is a database optimized for efficiently storing and retrieving data for analytical queries. Data from distributed and heterogeneous sources are ingested into a DW by means of data integration processes, commonly known as ETL processes [1, 13].

Another important approach to a unified access to huge volumes of data is a data lake architecture. A *data lake* (DL) is a large, centralized repository capable of integrating, storing, and processing data of arbitrary complexities and sizes [8, 12]. DLs are typically built on a distributed file system (e.g., Hadoop) and can store data in arbitrary formats. DLs are often used in conjunction with physical data warehouses, where a DL is serving as a repository for raw data and a DW provides a structured view on the data [7, 9].

Recently a *data mesh* gains popularity as a data architecture and data governance approach that promotes the decentralization of data ownership and the establishment of a common language for data across 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. The concept of data mesh is build with the support of multiple engineering technologies, called a *data fabric* [14].

All these architectures are complex w.r.t. software and hardware. All of them use DS connectors deployed as libraries and face problems outlined in Section 3.

3. DS connectors layer

As mentioned before, a typical approach to implementing connections to DSs (a.k.a. a connectivity layer) in any DI architecture is to maintain a library of connectors. This library is a collection of software components that are used to connect to and interact with various types of DSs. A LC typically includes connectors to a variety of different types of DSs. As outlined in Section 1, DI architectures impose some requirements on the connectivity layer. These requirements are especially important for a solution to follow the principles of micro-services architectures (see [2]). Libraries of connectors expose some limitations, which are the following.

Limited support for specific DSs: a LC may not include connectors for every possible type of DS that an organization might need to use. As a consequence, in the same system multiple libraries must be deployed, which increases the complexity of software dependencies.

Reduced flexibility: by using a LC, an organization may be limited to the capability provided by connectors in a given library, which in turn can limit the usage of specific features of a DS.

Dependency on third-party software: a LC is typically developed and maintained by third parties, which means that an organization using a given library is dependent on its developer to fix any issues or to provide updates and new features.

Performance and scalability: 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. As a consequence, a few different implementations (often in different programming languages) of the same connector may be needed, which again increases the complexity of software dependencies.

Complexity: if a LC is used by multiple applications, the library has to be embedded in multiple software modules. This leads to complex dependencies between various software components, which makes understanding and maintaining the whole architecture difficult.

Dependencies explosion: in a more mature solution, DS connectors depend on other software components (e.g., data governance catalog, vault providers, tunneling providers, policies repositories). In architectures based on micro-services, multiple services access data, where every service has to embed a LC and, as a consequence, inherits all the dependencies between other services.

Maintenance costs: since DS connectors have to be patched or upgraded regularly (e.g., security patches, new capabilities, licensing), all services that embed these connectors are impacted.

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 languages. In order to have all the DS types supported by all the services, one would need DS connectors to be re-implemented for all the languages used by client services. In practice, there are no standardized ways of implementing a LC for a plurality of programming languages.

4. Library of Connectors as a Service

Architectures based on micro-services have become an industry standard for developing interoperable systems. The drawbacks outlined in Section 3 became a motivation for developing another approach to organize DS connectors. To this end, we propose an architecture that provides DS connectors as a service - called as the *Library of Connectors as a Service* (*LCaaS*), as shown in Fig. 1.

The *LCaaS* is composed of the *Library of Connectors* (LC) and at least two facilitating components. Connectors from the LC are responsible for mapping native access interfaces of DSs into a common interface. *API Server* makes available the common interface to *Data consuming applications*. *Dispatcher* is responsible for forwarding requests to an appropriate connector. These requests contain data interaction definitions in the format specific to a data

source. This way, applications that connect to data sources directly can be easily migrated to the *LCaaS*.

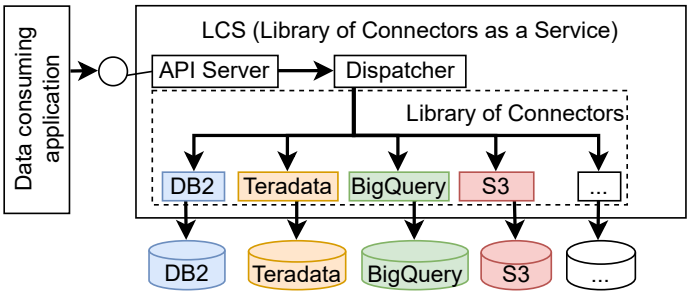

Fig. 1. *LCaaS* - the Library of Connectors as a Service

In architectures based on micro-services, REST APIs are common, but they are not suitable for accessing large volumes of data. The proposed *LCaaS* can be designed to externalize access to data via one of the common protocols and frameworks, suitable for processing large volumes of data. Implementation technologies typically used to build data-intensive architectures include: Apache Arrow Flight, gRPC, WebSocket, REST API, Apache Thrift, Apache Avro.

To provide the *LCaaS* functionality, multiple services can be implemented and they may be organized as a chain of design patterns. The patterns for security, usability, and some aspects of scalability were presented in [2], whereas in this paper we discuss patterns for portability, reliability, and performance (see Section 5).

5. Design patterns

5.1. Combined patterns

Patterns elaborated in [2] can be combined in an overall design, which can serve as a **fully-functional connectivity layer or the foundation for the data fabric**. One embodiment of this pattern is proposed in Fig. 2. It contains the following patterns: the integration with data governance catalog, integration with vault provider, tunneling, custom drivers, data locality, policy enforcement, data access monitoring and auditing, and chain of responsibility with bypassing.

In the proposed pattern design, a data access request is handled in the following steps. (1) *Data Consumer* (e.g., an end-user or another service) sends a request to the *Enforcement Proxy* service. (2) The *Enforcement Proxy* service decides if the request can be handled and if any anonymization is required; then eventually forwards the request to the *Monitoring Proxy* service. (3) *Monitoring Proxy* collects statistical information about the request, triggers monitoring of other services, and then forwards the request to *LCaaS*. (4) *LCaaS* retrieves connection properties for the request from *Governance Catalog* and resolves dynamic credentials with *Vault Provider*. If a subject data source is behind a firewall then the *LCaaS* orders a tunnel to be configured with *Tunneling Service*. If the subject data source requires a custom driver then the request is forwarded to *Custom LCaaS*. (5) *Custom LCaaS* decides whether it is more efficient to forward the requests to *Delegate LCaaS*, which is close to the data source. (6) *Delegate LSC* uses connection properties and credentials to establish the actual connection to the data source itself. (7) A response from the data source is sent back. To this end, the data source can use Apache Arrow Flight as its native interface. *LCaaS Delegate* can decide to transform data (e.g., compress), to reduce network traffic. (8) *Monitor Proxy* can collect various metrics on the performance of the environment. (9) Finally, *Enforcement Proxy* can transform data based on its internal policies (e.g., masking, anonymization, encryption).

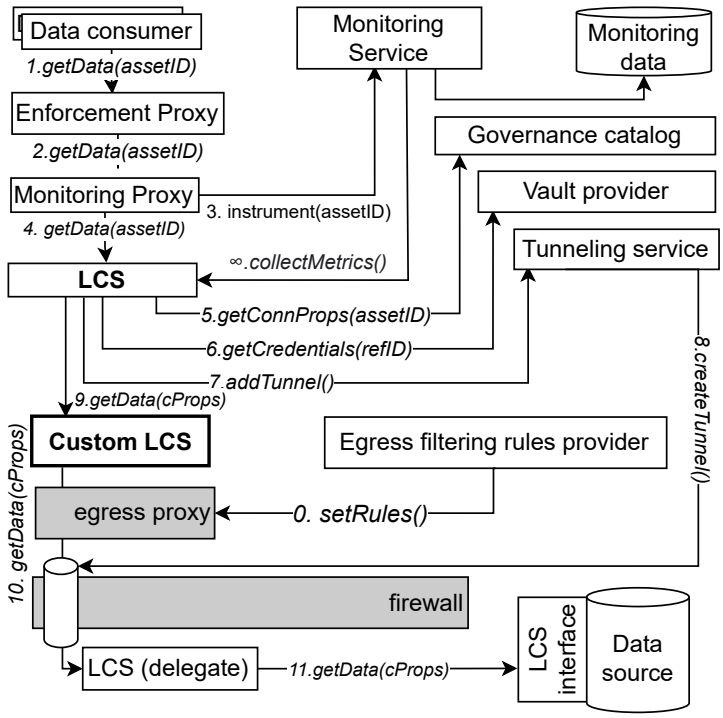

Fig. 2. Multiple patterns combined to offer foundation for the data fabric

5.2. Egress filtering

Egress filtering is a security mechanism that is used to prevent unauthorized or malicious traffic from leaving a network. It is used to enforce network policies and to prevent sensitive data from being exfiltrated or leaked outside a network. It may be used to protect against a variety of security threats, including: malware, spam, phishing, and data leaks. Egress filtering is typically implemented at a network perimeter and it involves examining an outbound traffic to ensure that it complies with predefined security policies. This may include for example checking the destination of the traffic, protocols being used, a content of the traffic. If the traffic does not meet the specified criteria, it may be blocked or redirected.

In the proposed architecture, egress filtering has to be configured only for a single service, i.e., the *LCaaS* (see Fig. 3). All the other services, which are clients of the *LCaaS*, can have their output traffic disabled. This way, the management of egress filtering is centralized.

More challenging use cases are exposed by multi-tenant architectures, where a single instance of a software application serves multiple tenants. In such an architecture, each tenant has its own dedicated pool of resources and data and is isolated from other tenants. This allows multiple entities to share the same hardware and software, while still maintaining the security and privacy of their own data. Multi-tenancy is commonly used in cloud computing and software-as-a-service (SaaS) environments.

Multi-tenancy has several benefits, including among others: reduced costs, increased scalability, and better resource utilization. However, it can also pose some challenges, such as the need to carefully manage security and privacy, and to eliminate the impact of one tenant on the performance of others. Multi-tenancy management becomes even more complex when tenants have different requirements for egress filtering.

The pattern proposed in this paper aims at ensuring that each tenant has its own instance of the *LCaaS*, while all the clients of the *LCaaS* can be still multi-tenants. This way, every tenant can have its own configuration of egress filtering and this configuration can be enforced at a network layer.

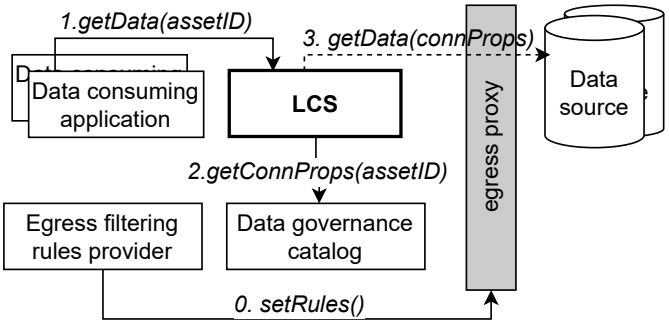

Fig. 3. The pattern for isolation and egress filtering for a single-tenant

This pattern offers the following advantages: (1) improved functionality by enabling low-level egress filtering configuration for each tenant, (2) improved performance by serving a plurality of tenants by the same instance of the *LCaaS*, in case all of them have the same egress filtering requirements, and (3) improved security by reducing the surface of potential security attacks, since access to external DSs is made from one service only.

5.3. Connector-less DSs

DSs connectors typically introduce some performance overhead (e.g., initiating a connection, transforming data, binding interfaces, translating commands). This can be particularly noticeable when executing a large number of data access requests.

If connectors are made available as the *LCaaS*, then they can be provisioned at a DS or a DS itself can implement the same interfaces as the *LCaaS* (see Fig. 4). This way, a request sent to the *LCaaS* can be routed directly to a given DS and data can be returned directly to the requesting client without any transformation. Although this sounds like an optimal approach, it is hard to assume that such level of unification is possible for all the existing DSs. Nevertheless, the pattern should be taken into consideration for DSs that can be enhanced with the interface.

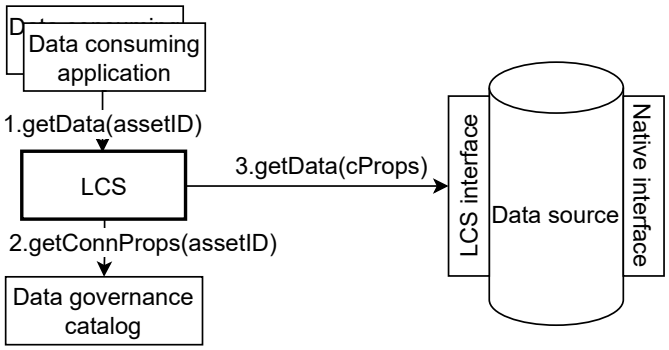

Fig. 4. The pattern for connections without a DS connector - a DS implements the same interfaces as the *LCaaS*

This pattern offers the following advantages: (1) improved maintainability, since the complexity of a software stack is significantly reduced and (2) improved performance achieved by reducing: adaptation of interfaces, network traffic, and data transformations.

6. Evaluation of design patterns

Most of the proposed patterns are suitable for architectures based on micro-services, where it is common to deploy micro-services as separate processes. The pattern designs assume that responsibilities are typically distributed across multiple independent services, each of which offering a specific functionality. Extra-process and inter-process communication introduce some overhead and resource consumption.

In this section, we present the results of the performance evaluation of the proposed patterns, focusing on throughput, CPU usage, and memory usage for common types of operations, in different types of tenancies.

6.1. Experimental environment

Hardware - the experiments were run on a cluster of three physical nodes. Its structure and hardware parameters are outlined in Fig. 5.

Software - all the experiments were conducted with a high-performance transport layer Apache Arrow Flight (a cross-language development platform for in-memory data, designed to efficiently transfer large datasets between systems and across different programming languages). Both, the data server and proxy server, depicted in Fig. 5 were implemented in Python, based on the Apache Arrow Flight framework. At this stage of experimentation, Jupiter notebook on the client side was used to implement a plurality of test scenarios and their automatic re-execution for statistically significant results. In production, the applications playing the role of clients can be implemented with many other technologies e.g., Flaskk/Diengo.

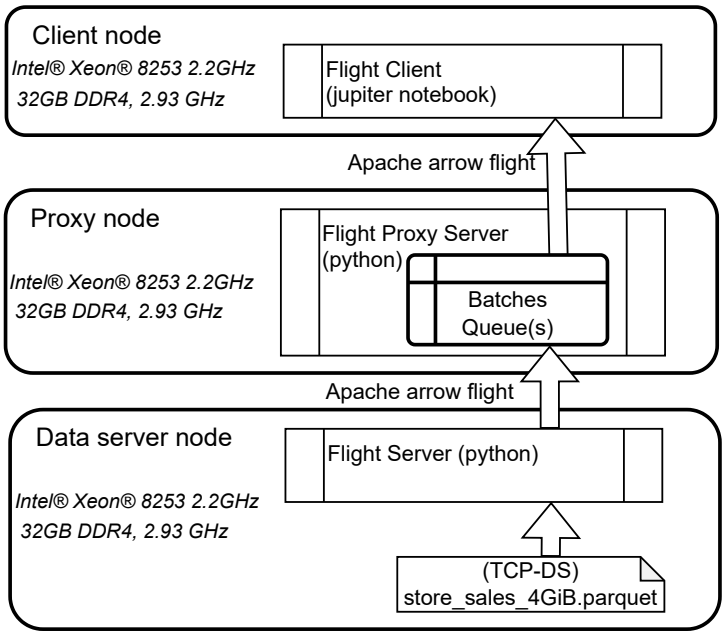

Fig. 5. Delegation pattern overhead - experiment setup

Dataset - all the experiments were processing data from the TPC-DS benchmark (see tpc.org). In the experiments, data from fact table *store_sales* were used and transformed into the Parquet file format (i.e., a columnar storage file format optimized for big data processing and analytics, designed to efficiently store and query large amounts of structured and semi-structured data).

6.2. Costs of passing requests

Most of the proposed design patterns assume that a data access request is *delegated* from one service (process) to another. For example, in the case of the combined solution depicted in Fig. 2, a request is passed across a chain of 10 processes (assuming that the tunnel is handled by two processes). Delegation is applied in the following patterns (see Section 5.1 and [2]): (1) custom drivers, (2) conflicting drivers co-existence, (3) egress filtering (for multi-tenants, see Section 5.2), (4) data locality, (5) policy enforcement, (6) data access monitoring and auditing, and (7) chain of responsibility. The **goal of this evaluation** is to understand what is the performance consequence of adding an intermediary service for handling requests (delegation).

This experiment assessed the impact of delegation on data unload throughput. Datasets of different volumes (from 1GB to 32GB) were unloaded in two scenarios: (1) directly from the Flight Server and (2) indirectly via the Flight Proxy Server, see Fig. 5. The proxy server was fully transparent, as it just bypassed requests and data to and from the data server node. The average throughput of unloads, with and without a proxy, was measured and its results are shown in Fig. 6. As we can observe from the chart, the measured overhead of delegation on throughput is very small and can be further reduced by buffering in the proxy server. In addition to the throughput, resource consumption was measured. The proxy server consumes around 80% of a single core to maintain the throughput levels in the scenario from Fig. 6.

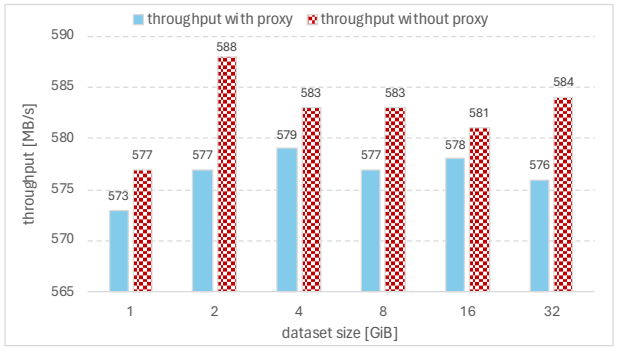

Fig. 6. Overhead of unloading throughput caused by delegation

The experiments show also that the proxy CPU consumption percentage is the same for tested data volumes and elapse times of unloads are proportional to data volumes, as visualized in Fig. 7.

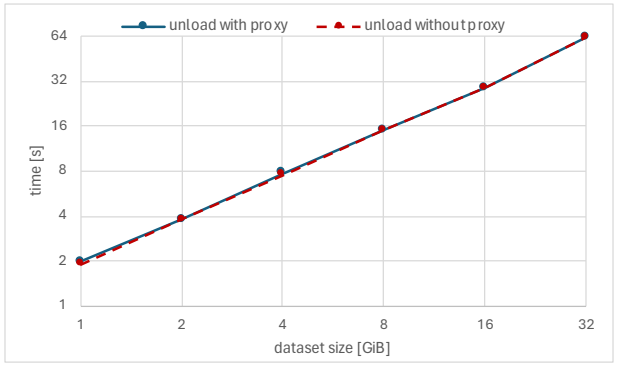

Fig. 7. Data unload elapse time overhead caused by delegation

Dispatching requests is a slightly different pattern than a simple delegation in a chain of

responsibilities. The dispatcher service decides which is the next service to send the request to. Dispatching takes place in the following patterns: custom drivers, conflicting drivers co-existence, and egress filtering. From the performance point of view, the dispatcher service introduced the same performance overhead as the proxy server, discussed above.

6.3. Memory usage

Data in the Apache Arrow format are organized in batches. Each batch is a list of columns and each column is an array of values. The arrays have an equal length, which is equal to the number of records in a batch. The size of batches (expressed in a number of records) has an impact on the performance both of the Data Server and the Proxy Server. The experiments on the 4GB TCP-DS dataset revealed that the smallest impact on the CPU consumption introduced by the proxy server is when a batch size included 65536 records (see Fig. 8).

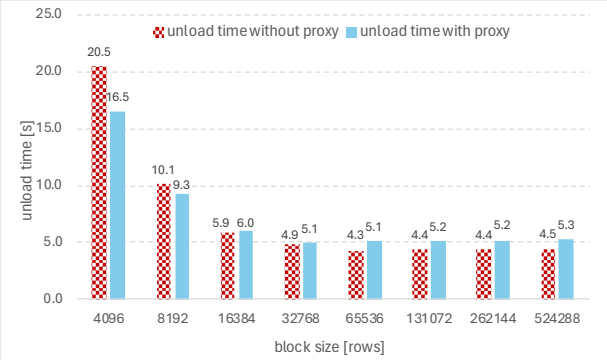

Fig. 8. Proxy CPU consumption vs. a batch size

Each record of the subject table requires 96 bytes of memory. With optimal batch size (65k records) the size of the batch is 6.2MiB. A simple proxy requires two queues for batches - one input queue and one output queue. In addition to batches in queues, one batch is needed to process data between an input and output. Those 5 batches consume 31MiB of memory. The flight server process itself requires 29.7MiB of memory. The overall consumption of memory introduced by the proxy server is approximately equal to 50.7MiB.

6.4. Scanning

A proxy server that only passes requests and data would be useless, therefore we run experiments to get insight into costs of the core operations on data. One of them is the scan of a full dataset. Scanning can take place in the following patterns: policy enforcement as well as data access monitoring and auditing.

The experiments were conducted on unloading different datasets of different volumes, but this time the Proxy Server scanned all the data to search for the maximum value in each column (the worst scenario). The elapsed time overhead and CPU usage were measured.

Elapsed time overhead introduced by scanning proxy is shown in Fig. 9. It is slightly bigger as compared to the results from Fig. 7, but it is still two orders of magnitude smaller than the unloading time. The proxy server consumes slightly more than one core to maintain the throughput while scanning all records and all columns.

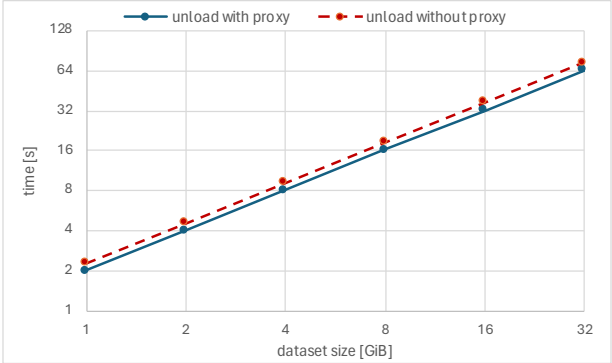

Fig. 9. Unloading combined with scanning: elapsed time and CPU usage

Table 2. The CPU usage of Apache Arrow Flight server for the multi-tenancy design pattern

server startup time [s]	first request time [s]	subsequent request time [s]
6.5	3.2	2.6

6.5. Costs of Single-, Multi-, and Mixed-tenancy

Single tenancy and multi-tenancy are two different architectural approaches for designing software systems that serve multiple users or tenants.

In the *single-tenancy* architecture, each user or tenant has a dedicated instance of the software application running on the server. Each instance operates independently and is isolated from other instances. Data, configuration, and resources are dedicated to each tenant. This approach provides maximum isolation and security between tenants but may require more resources and maintenance effort. A single tenancy may be required for the following patterns: egress filtering, monitoring proxy, and custom drivers.

In the *multi-tenancy* architecture, a single instance of a software application serves multiple users or tenants. Resources, including a database, are shared among multiple tenants. Each tenant data and configuration are logically separated within the shared infrastructure. This approach allows efficient resource utilization but requires proper isolation and security measures.

In certain situations, a *mixed-tenancy* (a combination of single- and multi-tenancy) can be efficient; for example, when the single-tenancy is to ensure custom egress filtering for each user. In this scenario, some subgroups of users may have the same requirements about egress filtering. For example, there may exist a group that allows no egress traffic, whereas another group may allow any egress traffic. Each group with the same egress filtering requirements can be served in the multi-tenant mode.

Multi-tenancy requires spawning servers dedicated to each account. It implies more resource consumption. In our evaluation, it was measured how much it costs for LCaaS implemented with Apache Arrow Flight. The experiments showed that maintaining an idle Apache Arrow Flight server for each tenant costs as much as 30MiB of memory. Spawning the server on-demand introduces some latency and CPU cycles spent on startup and warm-up of the server. The CPU times depicted in Tab. 2 can help to calculate the cost of spawning the server on demand. With both CPU and memory consumption, one can decide if it is better to maintain up-and-running service for a long time or spawn them on demand.

7. By-passing in the chain of responsibility

This technique is applicable to the following patterns: the custom drivers, conflicting drivers co-existence, egress filtering, data locality, policy enforcement, data access monitoring, and auditing, as well as the chain of responsibility.

Retrieving data via the Apache Arrow Flight protocol is split into the following two stages: (1) *get_flight_info* and (2) *do_get*, which provides information regarding a single specific data stream. Streams usually refer to datasets, like a table in a RDBMS or a file in a HDFS or a stream of events. *do_get* is in charge of fetching the exposed data streams and sending them to a client sequentially or in parallel.

In the case of the chain of responsibility pattern, the *get_flight_info* request from a client is sent across the chain of servers. Each server implements the same interface but serves a different purpose. In some cases, selected servers can be skipped. By-passing some servers can be applied either to both data access methods or only to *do_get*.

The decision about by-passing the next server S_{i+1} in a chain can be made by the proceeding server S_i and then both stages can be skipped by S_{i+1} (i.e., messages are sent directly to server S_{i+2}). The other situation is that during the processing of get_flight_info, the decision is made that *do_get* should be skipped by this node. It is important to stress that most of the elapsed time and resource consumption is related to *do_get*, as data are transferred in the second stage.

Our experiments showed that in the case of unloading of 4GiB of the TCP-DS dataset, the CPU time consumed by the *get_flight_info* is only 0.055% of the overall elapsed time. The average CPU times measured experimentally in the environment described in Section 6.1 are as follows: *get_flight_info: 0.000627s (constant)*, *do_get: 1.131731s (4GiB)*. The time necessary to handle the preparation request (*get_flight_info*) is negligible compared to data unloading time (*do_get*). Whenever possible, the *do_get* request should by-pass these nodes that do not have to process transferred data.

8. Summary

The existing implementations of a connectivity layer have significant limitations. By identifying these limitations (see Section 3) and by specifying a comprehensive set of requirements for a connectivity layer taking advantage of novel technologies (see, Section 1) we were able to work out a set of design patterns for the most common use cases (see Section 5).

Usually, adding extra hops, introduced by the chain of responsibility, degrades some aspects of performance (e.g., latency) but: (1) this degradation can be reduced by the approach proposed in Section 7, (2) some patterns can significantly improve performance (e.g., tunneling, data locality, connector-less data sources, monitoring and tuning). Notice that for analytical workloads, a throughput is more important than latency.

In this paper, we experimentally proved that the throughput reduction was negligible. By introducing a record batch buffering in intermediary nodes, the throughput can be even improved in some cases. Other aspects of performance can be improved by the introduction of workload management in the connector service itself. This topic is much broader than the scope of this paper. Data access prioritization, scheduling, throttling, dynamic scaling, and other optimization means can be developed for the proposed architecture. This way, we believe that the proposed *LCaaS* architecture opens new research paths in data integration architectures.

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.

References

[1] Berkani, N., Bellatreche, L., and Guittet, L.: ETL Processes in the Era of Variety. In: *Transactions on Large-Scale Data- and Knowledge-Centered Systems* 39 (2018).

[2] Bodziony, M. and Wrembel, R.: Data Source Connectors Layer as a Service - Design Patterns. In: *Int. Workshop on Design, Optimization, Languages and Analytical Processing of Big Data (DOLAP)*. Vol. 3369. CEUR-WS.org, 2023.

[3] Bouguettaya, A., Benatallah, B., and Elmargamid, A.: Interconnecting Heterogeneous Information Systems. Kluwer Academic Publishers, 1998.

[4] Brocke, J. vom, Hevner, A., and Maedche, A.: Design Science Research. Cases. Springer, 2020.

[5] Dehghani, Z.: Data Mesh: Delivering Data-Driven Value at Scale. O'Reilly, 2022.

[6] Elmagarmid, A., Rusinkiewicz, M., and Sheth, A.: Management of Heterogeneous and Autonomous Database Systems. Morgan Kaufmann Publishers, 1999.

[7] Errami, S. A., Hajji, H., Kadi, K. A. E., and Badir, H.: Spatial big data architecture: From Data Warehouses and Data Lakes to the LakeHouse. In: *Journal of Parallel and Distributed Computing* 176 (2023).

[8] Hai, R., Koutras, C., Quix, C., and Jarke, M.: Data Lakes: A Survey of Functions and Systems. In: *IEEE Trans. Knowl. Data Eng.* 35.12 (2023).

[9] Jemmali, R., Abdelhédi, F., and Zurfluh, G.: DLToDW: Transferring Relational and NoSQL Databases from a Data Lake. In: *SN Computer Science* 3.5 (2022).

[10] Jovanovic, P., Nadal, S., Romero, O., Abelló, A., and Bilalli, B.: Quarry: A User-centered Big Data Integration Platform. In: *Information Systems Frontiers* 23.1 (2021).

[11] Klumpp, M., Severin, B., Lechte, H., Menck, J. H. D., Keil, M., Straub, S. M., Ruiner, C., Milke, V., Hagemann, V., and Hesenius, M.: Driving Big Data - Integration and Synchronization of Data Sources for Artificial Intelligence Applications with the Example of Truck Driver Work Stress and Strain Analysis. In: *Int. Conf. on Information Systems (ICIS)*. 2022.

[12] Nargesian, F., Zhu, E., Miller, R. J., Pu, K. Q., and Arocena, P. C.: Data Lake Management: Challenges and Opportunities. In: *VLDB Endowment* 12.12 (2019).

[13] Simitsis, A., Skiadopoulos, S., and Vassiliadis, P.: The History, Present, and Future of ETL Technology. In: *Int. Workshop on Design, Optimization, Languages and Analytical Processing of Big Data (DOLAP)*. Vol. 3369. CEUR-WS.org, 2023.

[14] Strengholt, P.: Data Management at Scale: Modern Data Architecture with Data Mesh and Data Fabric. O'Reilly, 2023.

[15] Vaisman, A. A. and Zimányi, E.: Data Warehouse Systems - Design and Implementation. Data-Centric Systems and Applications, 2nd ed. Springer, 2022.

[16] Wiederhold, G.: Mediators in the Architecture of Future Information Systems. In: *Computer* 25.3 (1992).

[17] Wrembel, R.: Data Integration, Cleaning, and Deduplication: Research Versus Industrial Projects. In: *Int. Conf. Information Integration and Web Intelligence (iiWAS)*. Vol. 13635. LNCS. Springer, 2022.

[18] Wrembel, R., Abelló, A., and Song, I.: DOLAP data warehouse research over two decades: Trends and challenges. In: *Information Systems* 85 (2019).