On Testing a Data Access Architecture Based on Micro-services

Michał Bodziony[1], Filip Szóstak[2], Konrad Tarnacki[2] and Robert Wrembel[2,*]

[1]IBM Poland Software Lab Kraków, Poland
[2]Poznan University of Technology, Poznań, Poland

Abstract

Architectures that integrate or transport data between systems are more and more frequently built on micro-services, i.e., software components that are independent on each other but communicate over a network to provide some services. They offer advantages like increased flexibility of the architecture and scalability, but the price to pay for it are increased communication costs. In this paper we evaluate the performance of three architectures (varying in complexities) based on micro-services and relate their efficiency (data reading time and throughput) to a base-line architecture (client-server). The paper presents conclusions from a joint project between IBM Software Lab Kraków and Poznan University of Technology.

Keywords

data integration, data movement, data source connector, micro-service, container, Docker, Kubernetes, performance

1. Introduction and motivation

For over six decades, *Data integration* (DI) has been an active research domain [1, 2], with the common objective of providing end-users with a unified view on heterogeneous and typically distributed data. DI research has led to the development of a few standard DI architectures, namely: federated [3], mediated [4], data warehouse [5], lambda [6], data lake [7], data lakehouse [8], polystore [9], data mesh [10], and recently - data spaces [11, 12]. In each of these architectures, an integration layer is an indispensable component that facilitates the transportation of data from source storage systems into an integrated destination system. This layer is implemented through sophisticated software that executes *DI processes*. A *DI process* typically involves a sequence of tasks to ingest data from diverse sources and pre-process them into formats suitable for analytical and machine learning applications. A variety of commercial and open-source DI tools are available on the market [13].

In recent years, architectures based on micro-services are frequently adopted to facilitate the integration and transportation of large data volumes across heterogeneous systems [14, 15, 16, 17, 18]. Such novel architectures present a viable alternative to monolithic approaches, which typically are based on data silos [19, 20].

Micro-services [21] are software components that are independent on each other and communicate over a network to provide some services, like access to distributed and heterogeneous data sources, data pre-processing. This modularity allows for independent development, deployment, management, extensions with new functionalities (e.g., by deploying a new service), and scaling of individual services. Each service can be managed independently, based on its specific needs (e.g., resource allocation policy). By decomposing an integration process into a set of fine-grained services, each service is responsible for a specific integration task (e.g., data ingestion, cleaning, transformation). Thus, micro-services offer alternative and more flexible solutions to many of the limitations of monolithic approaches. Moreover, different services can be built using the most appropriate technology stack (e.g., a programming language,

Published in the Proceedings of the Workshops of the EDBT/ICDT 2026 Joint Conference (March 24-27, 2026), Tampere, Finland
*Corresponding author.
michal.bodziony@pl.ibm.com (Michał Bodziony); filip.szostak@student.put.poznan.pl (F. Szóstak); konrad.tarnacki@student.put.poznan.pl (K. Tarnacki); robert.wrembel@put.poznan.pl (R. Wrembel)
0000-0001-6037-5718 (R. Wrembel)

data storage system) for their specific function. The loosely coupled nature of micro-services makes a system more fault resistant, i.e., if one service fails, it is less likely to bring down the entire integration pipeline. Micro-services communicate with each other over a network, typically using APIs or message brokers.

Data integration by means of micro-services offers a few advantages as compared to standard centralized data integration approaches. First, each micro-service is responsible for making available data from its own DS. Second, it may encapsulate data access and privacy protection rules. Third, a particular service can be scaled up or down (depending on a data volume being processed) independently on other components of the whole integrated system. Thus, micro-services allow to build a modular DI architecture, where each implemented module is a small and independent service that executes a particular data integration tasks.

Micro-services share the same design principle with data mesh and data spaces, which is decentralization. While micro-services changed the technology for building applications by breaking down monoliths, data mesh and data spaces aimed at breaking data silos into domain-specific sources. For this reason, micro-services are particularly attractive to building data mesh and data spaces, where every independent data source is encapsulated in one or more specialized services, each of which offers a particular data set and offers specific data processing means.

From a technological perspective, micro-services cooperate with modern specialized software like *Airbyte* and Apache *Arrow Flight*, which provide means for efficient transportation of large data volumes. The increasing popularity of novel DI architectures and software tools to implement them creates demand for benchmarks, which could evaluate their performance. Organizations require tools that not only assist in selecting the best technologies, but also help identify issues with solutions that have already been implemented.

Unfortunately, the micro-services architecture pays the price of lower performance due to time overhead. The overhead is caused by: (1) increased network communication between services, (2) data format conversions between services (e.g., from the *Airbyte* format to the columnar format of *Arrow Flight*, and (3) time consumed by instantiating containers and PODs (outlined in Section 2).

In this short paper we report our findings on the performance of a simple data mesh architecture, implemented by means of micro-services, as reported in Section 4. The performance evaluation was done within a joint project run by IBM Software Lab Kraków and Poznan University of Technology.

2. Technologies

In this section we outline modern technologies for building DI architectures based on micro-services. We applied these technologies in the evaluation described in this paper. The technologies include: *Airbyte*, *Arrow*, *Arrow Flight*, *Cloud Pak for Data*, and *Docker*.

Apache *Airbyte* is an alternative tool for data extraction and transportation [22]. The heart of *Airbyte* is its large library of connectors - it offers over 600 pre-built connectors for data source and destination systems. The tool includes also API for building new connectors.

Apache *Arrow* is a platform for software development that provides means for transporting large data volumes between systems. It uses a standardized in-memory columnar format for representing table-like data [23, 24]. The columnar format storage allows efficient data processing by applying Single Instruction Multiple Data [25] and compression [26]. For this reason, it is frequently used in analytical systems (like data warehouses) and architectures for data integration and data transportation. If systems are located on the same node, *Arrow* provides out-of-the-box zero-copy data sharing that allows one system to make available data for usage directly to another system, by means of shared memory.

Data in the *Arrow* columnar format can be transported in bulk in a client-server architecture using Apache *Arrow Flight*, which uses the Google Remote Procedure Call (gRPC) mechanism [27]. To further increase data throughput, *Arrow Flight* groups data into structures called RecordBatches and uses parallel transportation (it was reported that *Arrow Flight* was able to achieve 20 Gb/s throughput per core [28]). Both technologies can be integrated with another component of a DI architecture, namely

IBM *Cloud Pak for Data*.

Cloud Pak for Data [29, 30] is an ecosystem of software services for integrating, storing, managing, preparing for analysis and analyzing data. It is available either as an on-premise software or as a service managed on IBM Cloud. An important feature of this software is its extensibility with the set of data sources that can be accessed by *Cloud Pak for Data*. To this end, a development toolkit called *Connector SDK* is used for creating connectors to new data sources [31]. The *SDK* generates skeleton files for developing connectors in Java and wraps them within an *Arrow Flight* server.

Micro-services are developed with the support of *containerization* that is a lightweight form of virtualization, where an application's executable code, all the needed libraries, and other files are packaged into a unit, called a *container*. Containers represent separate components of a DI architecture, like a database, a data ingestion component, a data wrangling component, an analytical application. Two primary technologies are used for containerization, namely *Docker* and *Kubernetes*. *Docker* [32] is used for building and running individual containers, whereas *Kubernetes* is an orchestration tool that automates the deployment, scaling, and management of containerized applications [33]. To manage containers, *Kubernetes* wraps them into units called *PODs*.

3. Related research

Benchmarking data processing technologies and architectures, like on-line transaction processing (OLTP) databases, on-line analytical systems (OLAP) - mostly data warehouses, and stream processing systems for years has been an active field of research and technological development. These efforts resulted in several benchmarks recognized by industry and research. The core standard includes the family of TPC benchmarks (tpc.org) for evaluating OLTP, OLAP, big data, the Internet of Things, and Artificial Intelligence solutions. There are also methods for evaluating specialized systems, like spatio-temporal data warehouses (e.g., [34, 35]) or graph databases (e.g., [36, 37, 38]).

In the context of the experimental evaluation of data integration/ data transportation architectures based on micro-services few works has been published so far.

[39] presented a benchmark for evaluating big data solutions that apply containerization. Performance metrics included processing time, throughput, resource usage, and latency. The data from *HiBench* (a big data benchmark suite) were used. Paper [40] presented the performance evaluation of *Arrow Flight* on a high-performance computing cluster, parameterized by the degree of parallelism and data volume. The evaluation is based on two metrics, namely: throughput and the total time required to query a dataset. The *NYC Taxi* dataset was used in the tests. [41] presented a system for genomics data processing on computational clusters, based on *Arrow* - for data storage and *Arrow Flight* - for data transportation. The evaluation included: (1) runtime of data shuffling, i.e., transferring data between a cluster nodes, duplicate removal, and overall data pre-processing; (2) cluster scalability w.r.t. the number of nodes; (3) the impact of memory size on performance; and (4) the throughput of *Arrow Flight* w.r.t. the number of *Flight* connections to the cluster. The standard genomic benchmarking set *NA12878* was used in the evaluation.

Despite the aforementioned efforts in testing architectures based on micro-services, to the best of our knowledge there is no evaluation that would clearly answered questions: (1) what is the time overhead for deploying components in the DI architectures, like *Docker* containers and *Kubernetes* PODs, (2) what is the time spent on reading data varying in size, and (3) what is the throughput and its stability w.r.t. varying data size. The outcomes from the project reported in this paper aim at answering these questions.

4. Tested architecture

For the purpose of this project, a simple architecture was built, with the following components used as building blocks: (1) *Airbyte*, (2) *Arrow Flight* - serving as a layer for moving data in a columnar format,

(3) *Docker* and *Kubernetes*, (4) IBM *Cloud Pak for Data*, (5) data sources - a relational database *PostgreSQL*, a non-relational storage *MongoDB*, and *Parquet* files.

As illustrated in Figure 1, four architectural variants were tested. Variant **A** served as the simplest baseline architecture, where the *Cloud Pak for Data* client established a direct connection to a data source using a native connector. Specifically, *Psycopg2* was used for *PostgreSQL*, *PyMongo* for *MongoDB*, and *Pandas* for *Parquet*.

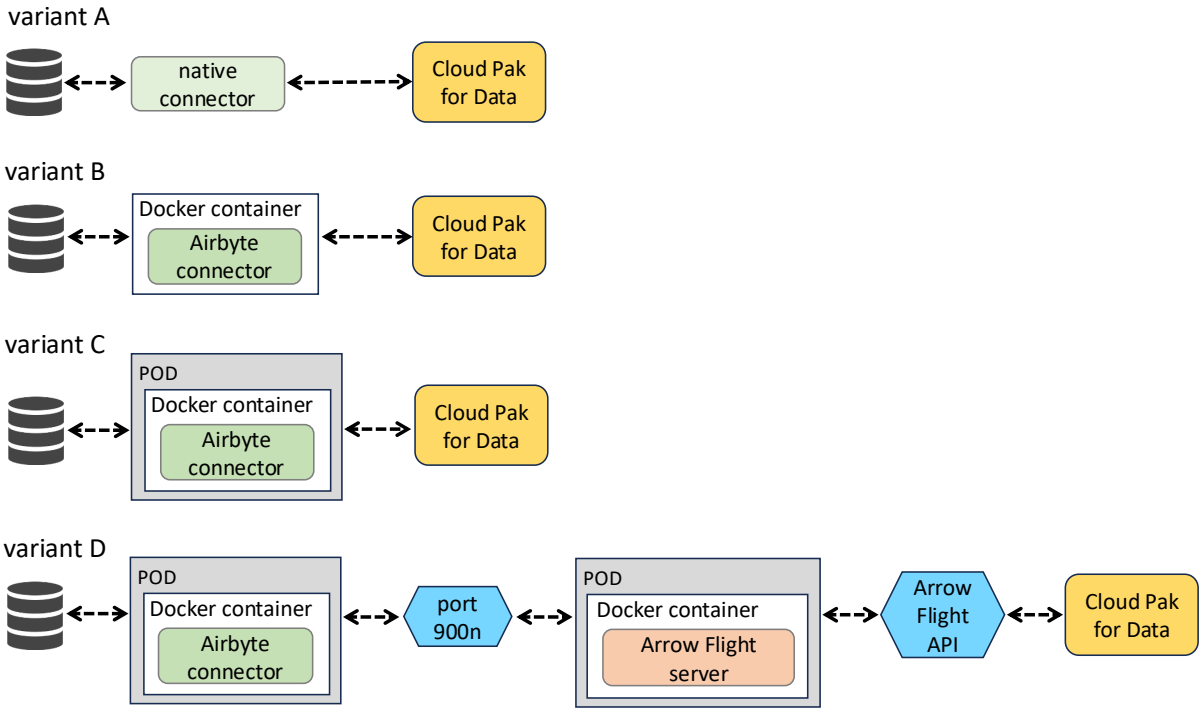

Figure 1: Schematic representations of four variants of the tested architecture

In variant **B**, the *Airbyte* source connector was used to connect to the data source by means of a dedicated connector for each of the three aforementioned data sources. The connectors were instantiated within a *Docker* container. In variant **C**, the connectors were instantiated in a *Kubernetes POD* [42]. Finally, in variant **D**, *Arrow Flight* was used as an intermediate layer between the connector and *Cloud Pak for Data*. The variant represents a typical data transportation architecture for large data volumes.

By extending the complexity of the architectures from **B** to **D** (by adding new components) we were able to measure time overhead introduced by each of these new components.

5. Experimental evaluation

The experiments were run on a workstation with: Intel Core i7-7700HQ 2.8 GHz, 16 GB of DDR4 RAM 2400 MHz, and an SSD disk, on Ubuntu 22.04 LTS.

Each experiment was repeated 10 times. To mitigate the impact of outliers, the minimum and maximum values were discarded, and the average was computed from the remaining 8 measurements. This average is presented in the subsequent charts. The batch size for *Arrow Flight* was equal to 100000 rows, which we found the most efficient for the whole evaluation.

Test data came from the the *TPC-DS* benchmark [43]. Specifically, the data were taken from table *store_sales* (23 attributes of types integer or decimal, average record length = 164 B). The following four data volumes were used in the experiments: (1) 1000 records (0.2 MB), 500000 records (78 MB), 1000000 (156 MB), 2000000 records (313 MB), and 4000000 records (625 MB).

The overall measured time for variant **A** comprised connecting to the data source, reading data, and returning the data as a stream of rows. Variant **B** introduced the additional overhead of container creation, variant **C** added POD creation, and variant **D** further included the creation of two PODs and data translation to the *Arrow Flight* format.

The goal of these experiments was to assess: (1) the time overhead for deploying additional components of the architectures, like *Docker* containers and *Kubernetes* PODs, (2) time spent on reading data varying in size, and (3) throughput and its stability w.r.t. varying data size.

The source codes of all the implemented components are available at GitHub: https://github.com/Cheriit/airbyte-arrow-integration.

5.1. Time overhead for deploying components

To quantify the deployment time of additional architectural components, a query selecting a single record from the smallest dataset was executed from *Cloud Pak for Data*. The overall query execution times, from initiation to result retrieval, for three distinct data sources are presented in Figure 2.

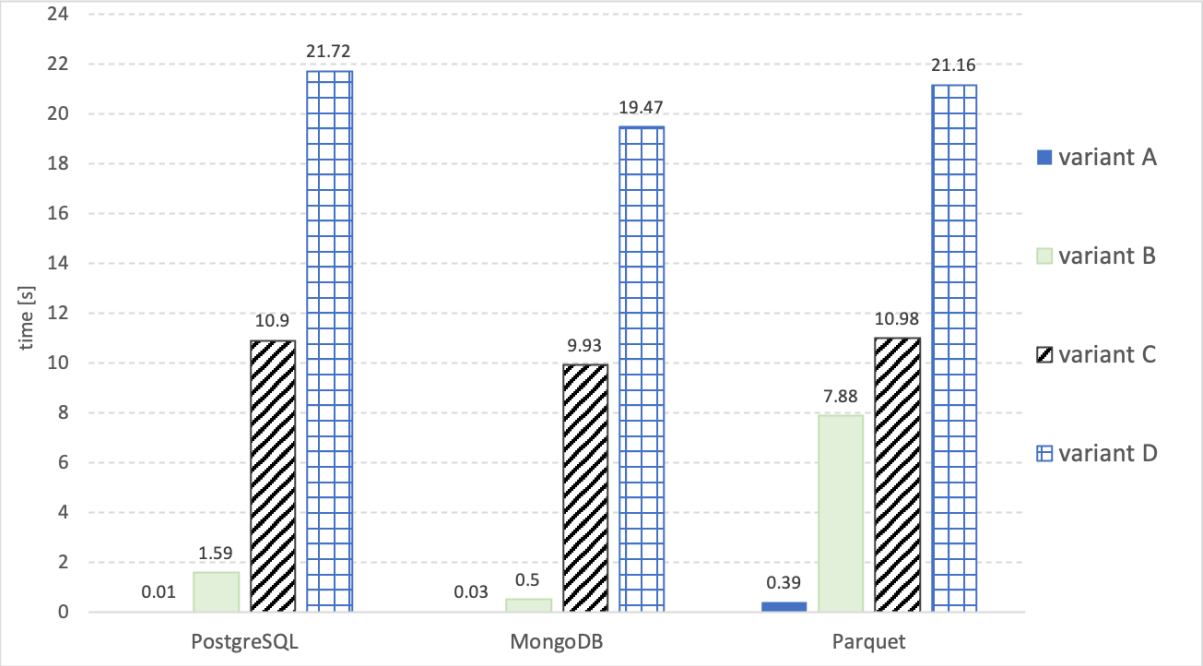

Figure 2: Time overhead for reading 1 record from a data source

The results indicated that the architecture in variant **A** of the tested architecture was the most efficient, likely due to the C implementation (thus efficient) of its native connectors, contrasting with the multi-language implementation of the *Airbyte* connectors. Moreover, variants **B**, **C**, and **D** suffered a performance penalty from the mandatory conversion of ingested data to the JSON format required by *Airbyte*. The total measured time of variant **C** was further increased by POD deployment and connection setup. Finally, the performance of variant **D** was additionally impacted by the deployment of an *Arrow Flight* server in a separate POD and the subsequent data conversion from the row-oriented *Airbyte* format to the columnar *Arrow* format.

5.2. Reading data of varying sizes

In this experiment, four data volumes differing in size were read by means of the four tested architectures. Figure 3 shows the results where *PostgreSQL* served as the data source. Similarly as in the previous experiment, variant **A** is the fastest, and variant **D** is the slowest.

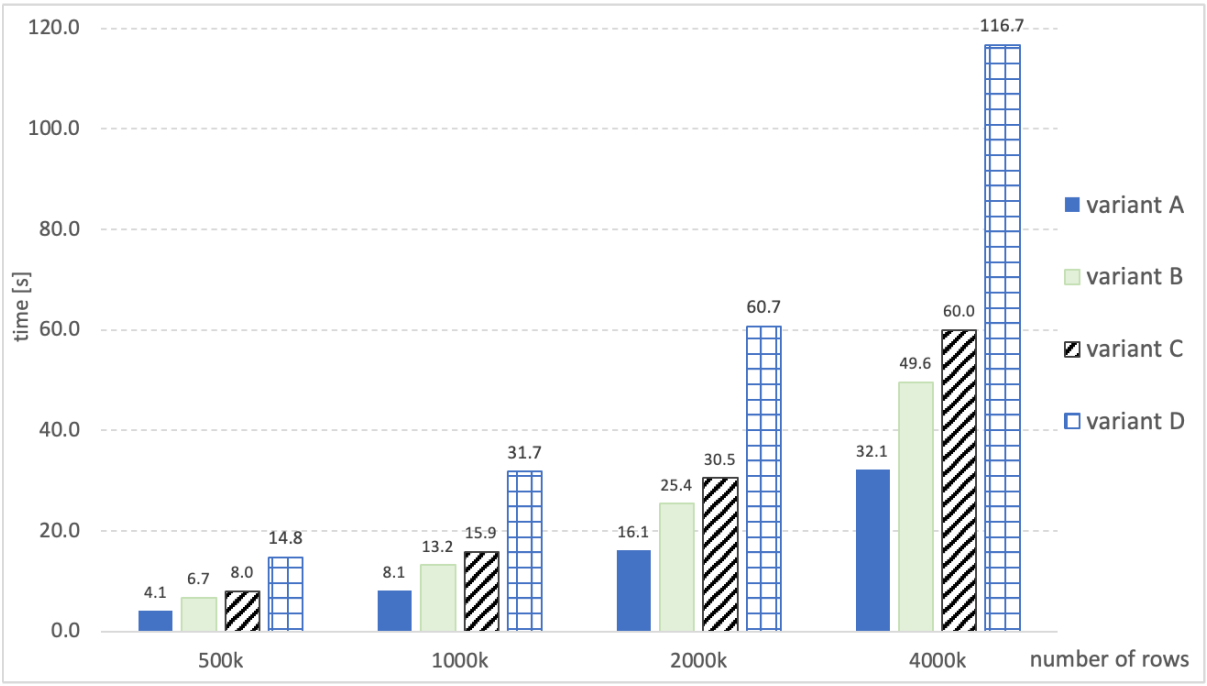

Figure 3: Overall data transfer time from *PostgreSQL*

Figure 4 shows the results where *MongoDB* served as the data source. Again, variant **A** is the fastest, and variant **D** is the slowest. Notice, that similar characteristics was observed for *Parquet* serving as a data source.

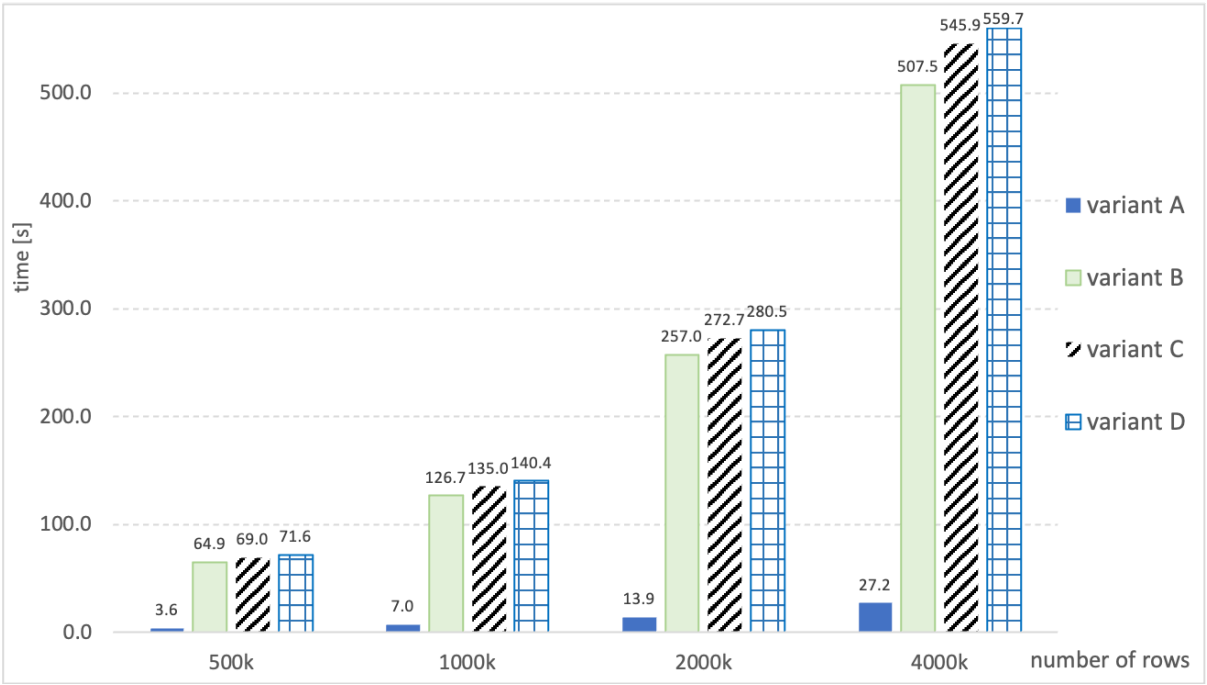

Figure 4: Overall data transfer time from *MongoDB*

5.3. Throughput

This experiment aimed at: (1) measuring the throughput offered by the four tested architectures and (2) verifying whether the throughput remains stable when a transmitted data volume increases (in case of memory leaks and malfunctioning communications between the components such a behavior could be observed).

The obtained results for *PostgreSQL* are shown in Figure 5. Variant **A** offers the highest throughput and variant **D** - the lowest. Moreover, the throughput remains stable for the tested data volumes.

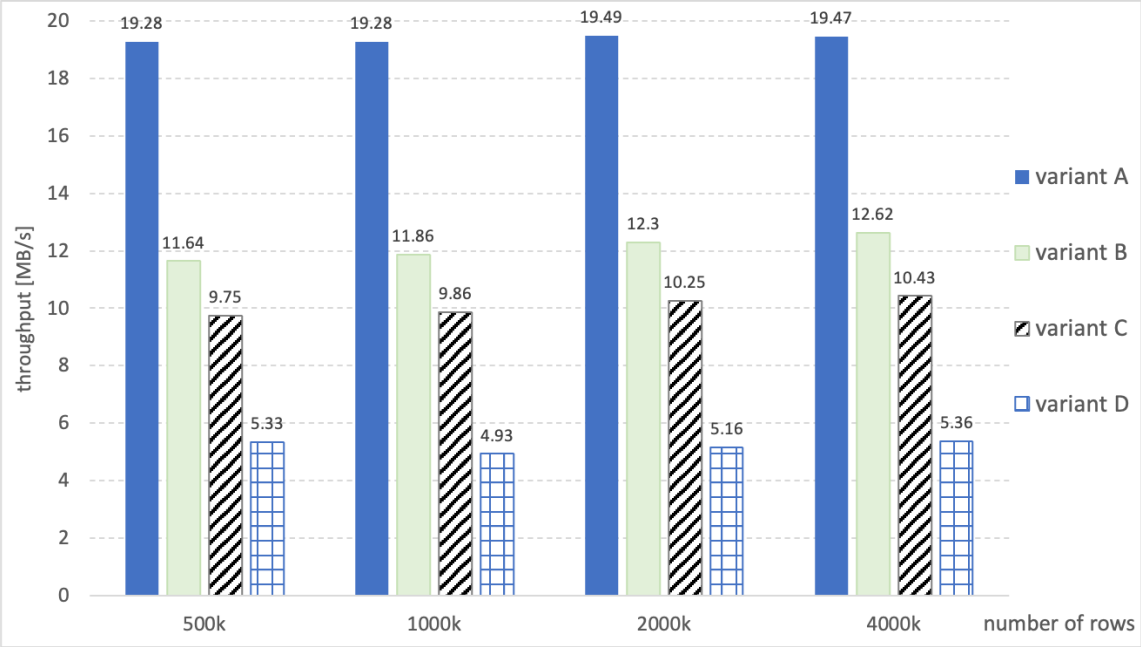

Figure 5: Throughput for a parameterized data volume for a data source being *PostgreSQL*

Figure 6 illustrates the performance results using *MongoDB* as the data source. Consistent with earlier observations, variant **A** achieves the highest throughput across all tested data volumes, whereas variant **D** remains the least performant. This performance profile was mirrored when accessing data stored in the *Parquet* format.

6. Summary and conclusions

A data integration architecture built with the support of micro-services has a few advantages (see Section 1) but its main drawback is additional time overhead added by multiple components of the architecture, like *Docker* containers and *Kubernetes* PODs, establishing connections between the components, and data format conversions. The experiments showed that native connectors (in the client-server architecture) offered the best performance. The more complex the micro-services architecture is the less efficient it is. The findings from this project have been included into the IBM Knowledge Base.

The project allowed us to identify paths for future works, which include: (1) efficient compression techniques for data in the *Arrow* format, (2) caching techniques for data returned by deployment commands of the components (e.g., *discovery*), (3) techniques for re-usability of PODs.

Declaration on Generative AI

The author(s) have not employed any Generative AI tools.

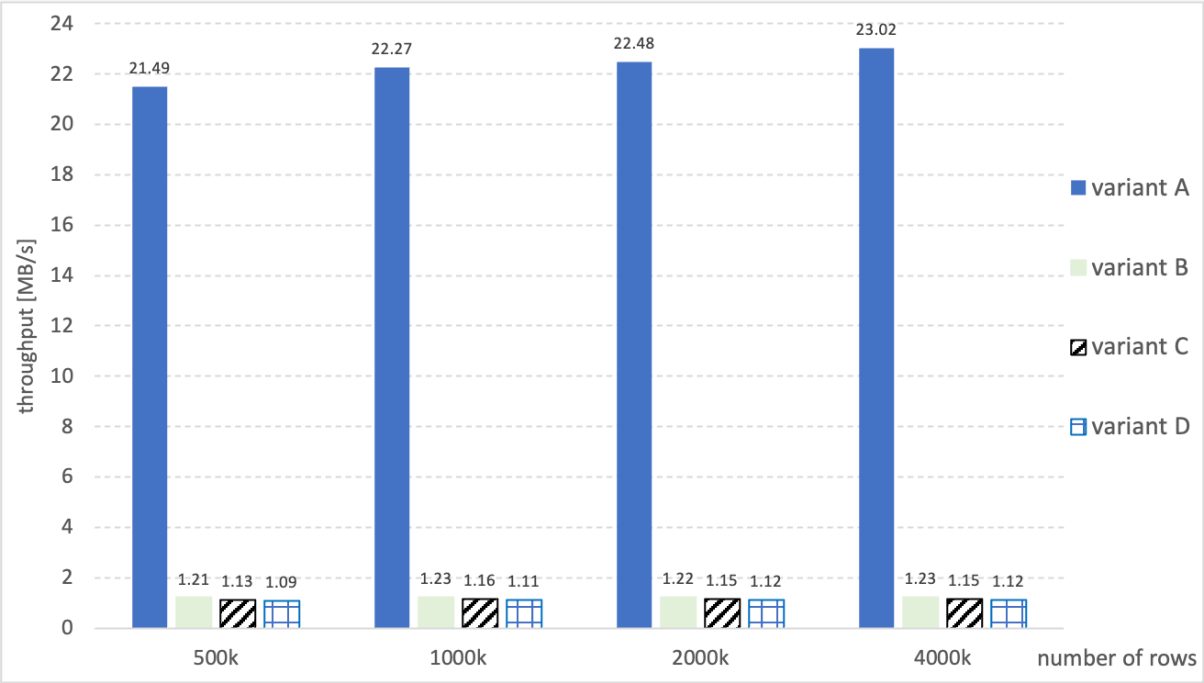

Figure 6: Overall data transfer time from *MongoDB*

References

[1] T. Timakum, S. Lee, H. Hu, I. Song, M. Song, DOLAP: A 25 year journey through research trends and performance (invited talk), in: Int. Workshop on Design, Optimization, Languages and Analytical Processing of Big Data (DOLAP), volume 3653, CEUR-WS.org, 2024.

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

[3] A. Elmagarmid, M. Rusinkiewicz, A. Sheth, Management of Heterogeneous and Autonomous Database Systems, Morgan Kaufmann Publishers, ISBN 1-55860-216-X, 1999.

[4] G. Wiederhold, Mediators in the architecture of future information systems, Computer 25 (1992) 38–49.

[5] A. A. Vaisman, E. Zimányi, Data Warehouse Systems - Design and Implementation, Second Edition, Data-Centric Systems and Applications, Springer, 2022.

[6] A. Gillet, É. Leclercq, N. Cullot, Lambda+, the renewal of the lambda architecture: Category theory to the rescue, in: Int. Conf. Advanced Information Systems Engineering (CAiSE), LNCS 12751, Springer, 2021.

[7] R. Hai, C. Koutras, C. Quix, M. Jarke, Data lakes: A survey of functions and systems, IEEE Trans. Knowl. Data Eng. 35 (2023).

[8] S. A. Errami, H. Hajji, K. A. E. Kadi, H. Badir, Spatial big data architecture: From data warehouses and data lakes to the lakehouse, Journal of Parallel and Distributed Computing 176 (2023).

[9] R. Tan, R. Chirkova, V. Gadepally, T. G. Mattson, Enabling query processing across heterogeneous data models: A survey, in: IEEE Int. Conf. on Big Data, 2017.

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

[11] C. P. Ayala, B. Bilalli, C. Gómez, J. Mazón, O. Romero, Challenges to enforce data quality in data spaces, in: Int. Workshop on Design, Optimization, Languages and Analytical Processing of Big Data (DOLAP) @EDBT/ICDT, volume 3931, CEUR-WS.org, 2025.

[12] A. Morejón, A. Berenguer, L. de Espona, D. Tomás, J. Mazón, Exploring content-based catalogs for enhanced discovery services in data spaces, in: Int. Workshop on Design, Optimization, Languages

and Analytical Processing of Big Data (DOLAP) @EDBT/ICDT, volume 3931, CEUR-WS.org, 2025.
[13] Gartner data integration tools reviews and ratings, 2025. URL: https://www.gartner.com/reviews/market/data-integration-tools, accessed Jan, 2026.
[14] D. Chia, Scaling data pipelines on kubernetes, 2020. URL: https://airbyte.com/blog/scaling-data-pipelines-kubernetes, accessed Jan, 2026.
[15] Cloud Native Computing Foundation annual survey, 2022. URL: https://www.cncf.io/reports/cncf-annual-survey-2022/, accessed Jan, 2026.
[16] S. Singh, C. H. Muntean, S. Gupta, Resilient microservices: an investigation into istio effectiveness in kubernetes, Cluster Computing 29 (2026).
[17] S. Zeng, H. Zhang, Z. Yan, CASLO: joint scaling and deployment for microservices leveraging context-aware SLO assignment, Journal of Network and Computer Applications 247 (2026).
[18] A. C. M. Saucedo, G. Rodríguez, F. G. Rocha, R. P. dos Santos, Migration of monolithic systems to microservices: A systematic mapping study, Information & Software Technology 177 (2025).
[19] O. Bruns, L. Söhn, T. Tietz, J. J. Steller, E. Posthumus, T. Schrade, H. Sack, Gotta catch'em all: From data silos to a knowledge graph, in: Satellite Events @ Semantic Web (ESWC), volume 15344 of *Lecture Notes in Computer Science*, Springer, 2024.
[20] M. Sienkiewicz, From data silos to data mesh: A case study in financial data architecture, in: Int. Conf. on Database and Expert Systems Applications (DEXA), volume 16046 of *LNCS*, Springer, 2025.
[21] IBM Developer. Build software with fine-grained, loosely coupled services, 2025. URL: https://developer.ibm.com/depmodels/microservices/, accessed Jan, 2026.
[22] Airbyte. Hundreds of connectors out-of-the-box, 2025. URL: https://airbyte.com/connectors, accessed Jan, 2026.
[23] W. McKinney, Introducing apache arrow flight: A framework for fast data transport, 2019. URL: https://arrow.apache.org/blog/2019/10/13/introducing-arrow-flight/, accessed Jan, 2026.
[24] What is Apache Arrow? Capabilities & benefits, 2018. URL: https://www.dremio.com/resources/guides/apache-arrow/, dremio; accessed Jan, 2026.
[25] J. Hildebrandt, J. Pietrzyk, A. Krause, D. Habich, W. Lehner, Partition-based SIMD processing and its application to columnar database systems, Datenbank-Spektrum 23 (2023). URL: https://doi.org/10.1007/s13222-022-00431-0.
[26] M. Kuschewski, D. Sauerwein, A. Alhomssi, V. Leis, Btrblocks: Efficient columnar compression for data lakes, ACM on Management of Data 1 (2023).
[27] gRPC documentation, 2021. URL: https://grpc.io/docs/, accessed Jan, 2026.
[28] It's time to replace ODBC & JDBC, 2019. URL: https://www.dremio.com/blog/is-time-to-replace-odbc-jdbc/, dremio; accessed Jan, 2026.
[29] IBM Documentation. Overview of Cloud Pak for Data, 2025. URL: https://www.ibm.com/docs/en/cloud-paks/cp-data/5.3.x?topic=overview-cloud-pak-data, accessed Jan, 2026.
[30] IBM Cloud Pak for Data, ???? URL: https://www.ibm.com/products/cloud-pak-for-data.
[31] IBM Cloud Pak for Data - Connector SDK, 2025. URL: https://github.com/IBM/cp4d-connector-sdk, accessed Jan, 2026.
[32] A safer container ecosystem, for everyone, 2026. URL: https://www.docker.com/, accessed Jan, 2026.
[33] Production-grade container orchestration, 2026. URL: https://kubernetes.io/, accessed Jan, 2026.
[34] C. Düntgen, T. Behr, R. H. Güting, Berlinmod: a benchmark for moving object databases, VLDB Journal 18 (2009).
[35] V. Pandey, A. Kipf, T. Neumann, A. Kemper, How good are modern spatial analytics systems?, VLDB Endowment 11 (2018).
[36] Graph benchmark. Comparing and understanding graph databases, ????. URL: https://graphbenchmark.com/, accessed Jan, 2026.
[37] Open graph benchmark, ????. URL: https://ogb.stanford.edu/, accessed Jan, 2026.
[38] Graph Data Council, ????. URL: https://ldbcouncil.org/, accessed Jan, 2026.
[39] S. Ceesay, A. Barker, B. Varghese, Plug and play bench: Simplifying big data benchmarking using

containers, in: Int. Conf. on Big Data (BigData), IEEE, 2017.
[40] T. Ahmad, Benchmarking Apache Arrow Flight - a wire-speed protocol for data transfer, querying and microservices, in: Benchmarking in the Data Center: Expanding to the Cloud (BID@PPOPP), ACM, 2022.
[41] T. Ahmad, C. Ma, Z. Al-Ars, H. P. Hofstee, Communication-efficient cluster scalable genomics data processing using Apache Arrow Flight, in: Int. Symp. on Parallel and Distributed Computing (ISPDC), IEEE, 2022.
[42] Kubernetes documentation. PODs, 2025. URL: https://kubernetes.io/docs/concepts/workloads/pods/, accessed Jan, 2026.
[43] TPC Benchmark DS - standard specification 3.2.0, 2021. URL: https://www.tpc.org/TPC_Documents_Current_Versions/pdf/TPC-DS_v3.2.0.pdf, accessed Jan, 2026.