Towards a Modular Data Management System Framework

Haralampos Gavriilidis*   Lennart Behme*   Sokratis Papadopoulos*
Stefano Bortoli†   Jorge-Arnulfo Quiané-Ruiz*,⊤   Volker Markl*,⊤

*Technische Universität Berlin   †Huawei German Research Center   ⊤DFKI GmbH

ABSTRACT

Today's data management systems (DMSes) are implemented using monolithic architectures. This is explained by the fact that, historically, the DMS market was dominated by commercial solutions. However, with the advent of open source, lots of alternative DMSes and DMS components have emerged. Yet, they cannot be easily integrated because DMSes are still designed as monoliths. We propose PolyDMS, our vision towards composable DMSes. The core idea of PolyDMS is to split up and wrap individual components with standardized interfaces, such that a Component Orchestrator can flexibly construct DMSes out of them. Users can compose new DMSes by writing small programs in our System Definition Language. PolyDMS allows them to quickly instantiate new DMSes according to their needs while giving them the opportunity to reuse existing components. Our proof-of-concept implementation shows that composing multiple components into a new DMS can yield additional functionality and better performance.

1 INTRODUCTION

Today's data management system (DMS) landscape offers a diverse range of products. Since the dawn of the "one-size-does-not-fit-all" era [27], a number of special-purpose systems emerged on the highly-contested DMS market. While each DMS excels in particular scenarios [1], the system architectures share many similar components. The general architecture of query processors has not changed drastically since the 1980s [16, 19]. Despite this fact, due to the monolithic approaches taken when designing new systems, system architects and engineers reinvent the wheel again and again, as they have to build all components from scratch.

We argue that DMS components should be easily reusable to foster impactful innovation and to make system engineering more efficient. Domain experts should have tools at their disposal that allow them to make special-purpose components composable via pre-defined APIs. Furthermore, composing a new system should not be limited to the few experienced DMS architects in our community.

Unfortunately, the "state of the composability union" in the DMS ecosystem is not strong as of now. We believe that the problem of limited DMS composability is twofold: First, there are no standards that allow system engineers to make their components easily composable. Second, composing existing components into a DMS requires a huge amount of manual effort. Historically, DMSes grew into general-purpose monoliths, with the individual components not being intended for reuse. The widespread use of open source software has dramatically eased the access to existing DMS components but has not removed the hurdle of their limited integrability. Even the few existing standalone components, such as Apache

Calcite [6] and Orca [26], are designed for a rather tight and use case-specific integration. While some DMSes, like Apache Hive [8], evolved by continuously integrating new components into the system, their development history is a testament to the large manual effort and custom solutions required for a good integration. On the other side, recent approaches, such as Presto [25], try to subsume functionality of legacy DMSes and query them seamlessly through connectors. However, it is both impractical and costly to continuously port data and pipeline logic from legacy DMSes to the latest and most advanced systems. Therefore, it is clear that our community has yet to present conceptual approaches combined with practical evidence for a straightforward construction of composable DMSes, rather than relying on yet another new DMS.

In this paper, we present our vision to address the composable DMS research gap and lay out our research agenda for a DMS composition framework, which we call PolyDMS. In our vision, system architects are able to build a new DMS by reusing individual components as building blocks. The idea of PolyDMS comprises three major contributions: (1) First, we define abstract types of DMS components, such as query interfaces or optimizers, and design generic interfaces for each type so that existing libraries can easily turn into a DMS component; (2) To combine the standalone components, we build a Component Orchestrator that interacts with all components, enabling the composition of DMSes; (3) Finally, we design a high-level System Definition Language (SDL) that acts as an interface to the orchestrator and allows users to easily compose a new DMS out of existing components. In a collaboration with Huawei, we have started to materialize and evaluate the potential of our vision and built a proof-of-concept (PoC) OLAP system based on the PolyDMS framework. The promising initial results motivated us to move forward with our research agenda.

In the following, Section 2 reviews previous approaches to building a composable DMS and highlights their limitations; Motivated by current limitations, Section 3 presents our vision for the PolyDMS framework; Section 4 discusses our PoC for a composable DMS; Section 5 concludes with an outlook on future work.

2 EXISTING COMPOSITION APPROACHES

We categorize DMS composition into two main approaches: vertical and horizontal composition. Vertical composition (Section 2.1) is about pipelining multiple components along the query processing flow. In horizontal composition (Section 2.2), on the other side, multiple components of the same type (e.g., execution engines) process queries in parallel. While vertical composition combines separate components into a single query processor, horizontal composition creates a meta-system that spans over multiple query processors. Figure 1 shows a conceptual comparison along the two DMSes Apache Hive and Apache Wayang (incubating). We review prominent examples for each composition approach and close this section with a discussion of the state-of-the-art limitations (Section 2.3).
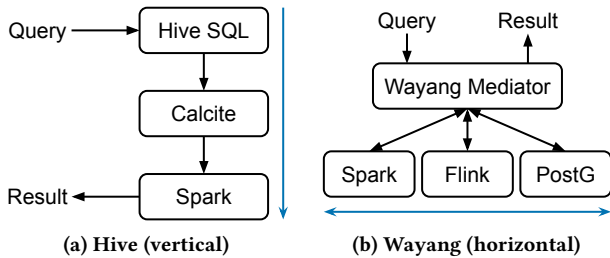
Figure 1: DMS composition approaches (indicated by →).

2.1 Vertical Composition

Current open source and commercial DMS architectures mostly follow a predefined design [19], offering user interface, query optimizer, execution, storage, and metadata components. When composing a DMS vertically, we "glue together" existing frameworks, tools, systems, or platforms. A prominent example for vertical DMS composition is the trend towards separating compute and storage engines, first introduced for cloud environments by the MapReduce paradigm [12]. Among other advantages, this paradigm allows to scale storage and compute individually, and hence was adopted by many commercial DMSes, such as Snowflake [11]. However, soon it became apparent that MapReduce's low-level interface is not the right abstraction for novice data analysts. To offer more intuitive user APIs, research and industry proposed multiple front-ends on top of the MapReduce API, e.g., Pig [23], JAQL [7], or Apache Hive [8]. What all these approaches have in common is that they offer their own higher-level APIs, and compile tasks defined in those APIs to MapReduce programs. Next to convenient user abstractions, higher level APIs also enable advanced query optimization and efficient execution [5, 14], e.g., by leveraging the relational model. When metadata, i.e., relation statistics, are available, systems can employ additional optimizations like join ordering.

Hive is an example for a vertically composed DMS (Figure 1a): It offers its own SQL interface, relies on Calcite for cost-based optimizations, offers either Apache Spark, Apache Tez, or MapReduce as execution engines, and provides its own Hive Metastore that offers metadata to all mentioned components. Even though Hive has evolved out of multiple components, we still consider it to be monolithic, as it tightly couples the aforementioned components into its architecture without a straightforward way to replace them.

2.2 Horizontal Composition

Oftentimes, there is a need to query multiple DMSes at the same time, for example during data integration tasks or to use multiple query processors for a single query [20]. We refer to this type of composition, where multiple components of the same type are used for one query, as horizontal composition. Systems designed for data integration in federated environments (e.g., Tsimmis [10], Garlic [9], Presto [25]) perform query splitting to process different parts of one query on different systems. Building upon the seminal work on federated databases, polystore systems, such as BigDawg [13], Myria [28], Estocada [4], or CloudMdsQL [21], give users more flexibility by offering different query languages and optimizing across system boundaries. Cross-platform systems (which also support the polystore case [20]), such as Wayang [29] (formerly Rheem [2]) and

Musketeer [15], offer sophisticated optimization techniques that allow to place different operations of a query on different systems (e.g., Figure 1b). What all these approaches have in common is a central mediating component that accepts user queries, splits them into different subqueries and orchestrates the execution over multiple DMSes. We argue that those approaches are monolithic as well, because their architectures are similar to traditional DMSes that tightly couple individual components. Hence, they do not allow switching individual components in a plug-and-play manner.

2.3 The Missing Piece

Overall, although one can mix the two composition approaches, we observe that both in vertical and horizontal composition existing systems are tightly coupled. This makes it overall cumbersome to exchange DMS components to, for example, try out novel optimization techniques with a different optimizer. Furthermore, when building a system for a specific use case, system architects either have to re-invent the wheel by implementing all DMS components or put a lot of effort into manually integrating existing components.

We believe that by standardizing the interactions of individual components, one can foster a stronger collaboration between component developers and encourage the development of specialized standalone components. This will allow us to integrate existing components or research prototypes in a much simpler way, and hence enable new DMS designs that do not re-invent the wheel. Thus, a requirement for building composable DMSes is (1) to abstract component functionality in well-defined interfaces, and (2) to provide primitives for component interaction and orchestration for seamlessly integrating all components through their interfaces.

3 THE POLYDMS FRAMEWORK

We now present PolyDMS, our vision towards addressing the missing piece in composable DMSes. Our overall goal is to enable users to easily create a loose coupling of existing DMS components, such that they do not re-invent the wheel every time when building a new DMS. The key idea of PolyDMS is that existing (and future) DMS components should be independently reusable, i.e., they do not depend on specific other components to work. Therefore, we propose to couple DMS components through a micro-service architecture where each service is accessible through a well-defined API. On a high level, PolyDMS allows users to compose new DMS instances by assembling inter-component pipelines. In the following, we give a brief overview of the PolyDMS framework (Section 3.1), describe how we envision "Components-as-a-Service" (Section 3.2), discuss the orchestration of components (Section 3.3), and outline scenarios that can benefit from PolyDMS (Section 3.4).

3.1 Overview

We show our initial architecture in Figure 2. PolyDMS's design is based on the observation that DMS architectures share components with similar functionality. The colored boxes represent the predefined set of interfaces that system developers can implement to wrap existing or newly-developed components as services. Initially, we define *Query Interface*, *Optimizer*, *Execution Engine*, *Storage Engine*, and *Metadata Store* as abstract DMS component types, as these are the main components involved in OLAP query processing.
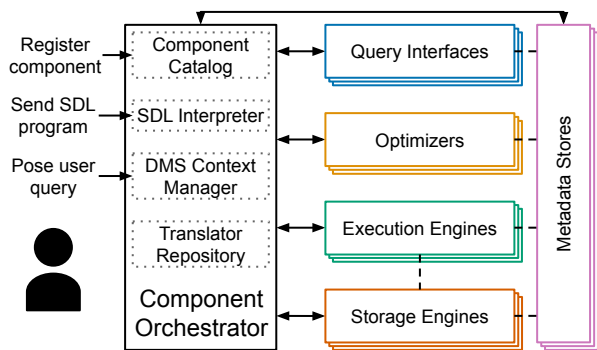
Figure 2: An overview of the PolyDMS framework. Users instantiate a new system by writing a System Definition Language statement. To interact with their system, they send queries to the Component Orchestrator.

Notice that, in practice, one service could handle multiple functions in the query processing flow; there is no dogmatic requirement to separate every abstract component type in every DMS.

From a system perspective, the *Component Orchestrator* is the core of PolyDMS, as it interacts both with users and components. The *Component Catalog* is responsible for maintaining a list of available services. System developers can compose a new DMS by writing a program in our domain-specific language called System Definition Language (SDL). The *SDL Interpreter* parses and validates SDL programs, e.g., by checking the validity of the component flow and the availability of the referenced components. After validating the SDL program, the *DMS Context Manager* instantiates the system and returns the resulting PolyDMS context to the user.

From an end-user's perspective, they can interact with the new DMS as with any other DMS using the previously created context. Upon receiving a query, the Component Orchestrator calls each component and passes the input and output through their interfaces to trigger the query execution. To transform the output of one component into the input of the next, the Component Orchestrator utilizes translators maintained in the *Translator Repository*. These are used to move between representations, e.g., translating a logical query plan from Calcite to a Wayang plan.

3.2 Component Types

Our architecture abstracts DMS components into clearly defined services which would otherwise be intertwined inside a monolithic system. Below, we discuss each component type in more detail.

The Metadata Store is a special component that interacts with all other PolyDMS components. We expect Metadata Stores to act as global catalogs, e.g. providing schema information to the query interface for validation purposes, which can also be leveraged for data discovery and governance purposes. More importantly, the Metadata Store contains data profiling information, such as row counts and attribute value histograms, to assist the optimizer.

The Query Interface accepts user queries and outputs Abstract Syntax Trees (ASTs). The responsibility of this component is to validate user queries with regard to syntax and semantics, and to transform them into an AST, e.g., translating a SQL query into

a logical query plan. Other examples for potential user APIs are Pig [23], SPARQL, and Weld [24], which allows to intermix Python libraries, such as Pandas or Numpy. Although parsing and validation could be done in individual components, we unify these tasks into one component for the sake of simplicity. Note that the query interface also interacts with the Metadata Store to access metadata, e.g., the dataset tables and attributes.

The Optimizer Interface accepts ASTs and outputs ASTs. The responsibility of this component is to optimize a given plan. For example, one could use a rule-based optimizer for simple query rewrites. If metadata about the queried tables are available, the Optimizer can leverage the Metadata Store to perform cost-based rewrites like join ordering. Alternatively, system engineers could also wrap existing optimizers, such as Apache Calcite [6] or Orca [26]. Note that query optimization is not limited to optimizations found in relational optimizers, but can also be of different nature. For instance, one can use a cross-platform optimizer, such as Wayang, to decide on an efficient operator placement strategy. Also note that one can utilize multiple Optimizers in a row, e.g., to first apply rule-based and then cost-based optimizations or to first perform logical and then cross-platform optimizations. In this case, the Component Orchestrator requires a Translator for transforming between two different optimizer representations.

The Execution Engine Interface accepts query plans and returns the query results. Execution engines offer low-level APIs, to which we can map ASTs using an appropriate Translator. For example, after optimizing a query plan logically through an Optimizer, we use a Translator to create an Apache Spark job. This way, it is possible to use different optimizers for one execution engine, e.g., by using Calcite- and Orca-to-Spark Translators, one can use an alternative to Spark's own Catalyst optimizer [5].

The Storage Engine Interface is not the main focus of our initial work so far as there already has been seminal work on separating compute from storage [12]. Essentially, one must provide means to access the data that a particular storage engine holds. Examples for storage engines include file systems (e.g., HDFS) or cloud storage buckets, but also existing DMSes.

3.3 Component Orchestration

The Component Orchestrator is an integral part of PolyDMS, as it maintains active components, controls the information flow, and steers the interaction between components.

The Component Catalog maintains a list of all active components an orchestrator manages. After implementing the respective component type interface and starting a service instance, users register components at the orchestrator. This includes important meta data, such as the access point and the input/output representation that the component expects. In general, the Component Orchestrator can manage more components than required for a single DMS so that users can flexibly compose different DMSes out of the set of available components.

The SDL Interpreter parses SDL programs submitted by system developers. SDL is a small domain-specific language inspired by workflow definition languages, such as Apache AirFlow [18], and

provides primitives for defining component interactions. Most importantly, SDL offers a `PolyDMSContext` object and a `translate()` function. The `PolyDMSContext` contains references to the individual system components, what input they receive, and how their output is processed. `translate()` is a generic method, which the orchestrator replaces with a specific Translator from the Translator Repository after analyzing the metadata of the involved components. The SDL Interpreter automatically checks if an SDL program is valid by inspecting the output-input connections of all components and validating the query processing workflow. For example, if the output of an Execution Engine is used as the input of an Optimizer or there is no fitting Translator in the Translator Repository, the interpreter throws an error. Program 1 gives a concrete SDL example showcasing how users can use a `PolyDMSContext` and how `translate()` connects components with different input/output.

The DMS Context Manager instantiates new systems and manages all existing system contexts. Upon submitting a valid SDL program, users receive a `PolyDMSContext`, which offers methods for executing queries and running administrative tasks, such as checking the system status or shutting down the system. How users interact with the newly created system depends on the chosen Query Interface component. A SQL interface, for example, simply accepts strings that adhere to the respective SQL dialect while a Pandas interface accepts input that is understood by the underlying Python interpreter. Going forward, we envision that the Context Manager and `PolyDMSContexts` could be wrapped by a web-based GUI to ease the system interaction for non-expert users.

The Translator Repository maintains all user-defined translations between different input/output representations. Translators act as "glue code" between distinct representations, e.g., a query plan in Apache Calcite and Wayang. A Translator implementation consists of a mapping between the internal representations of the two involved components. The concept of mappings between different representations has already been explored in various previous works. For example, Hagedorn et al. [17] map Pandas operators to SQL statements, while Emma [3] and Wayang [2] map their own operators to underlying system operators (e.g., Spark, Flink). Beyond mappings, we are investigating the use of static code analysis for the translation of imperative languages. This could be useful to translate queries written in popular data science libraries, such as Pandas, to a representation understood by relational algebra-based optimizers and execution engines.

3.4 Application Scenarios

Having laid out the details of our vision for composable DMSes, we now present two scenarios based on our real-world observations in which PolyDMS can assist DMS architects.

Diverse User Requirements. DMSes, especially in OLAP use cases, are often used by different user groups (e.g., departments). Given the varying user backgrounds, it is reasonable to assume that there are different system language (i.e., query interface) preferences among the user base. While SQL is the most common standard to interact with DMSes, object-oriented Python libraries like Pandas or functional languages have also been shown to work as DMS query interfaces [17]. Another example is when OLTP-specialized

systems are also required to efficiently serve OLAP-queries, as shown by replacing MySQL's optimizer with Orca [22]. PolyDMS makes it easy to change system components like query interfaces based on user requirements. For example, by plugging a Pandas-to-SQL Translator [17] into the system, one can support multiple user front-ends in an existing relational optimization pipeline.

Optimization Synergies. As query optimizers take a query plan as input and produce another plan as output, we can chain them to potentially improve the optimization results. Some existing systems, such as Calcite, already offer multi-stage optimization as an internal feature [6]. While in the case of Calcite, this is limited to the repeated application of Calcite's rule- and cost-based optimization, PolyDMS enables users to combine any optimizer with each other. As available optimizers often excel in different situations, we argue that combining different optimizers can yield better query runtime performance, e.g., by combining a logical optimizer with a cross-platform optimizer (Section 4). Moreover, PolyDMS also simplifies the efforts of researchers when benchmarking new optimizers and analyzing the impact of new optimization techniques.

4 PROOF-OF-CONCEPT

We evaluated our vision for PolyDMS in a collaboration with Huawei by composing a PoC OLAP DMS based on PolyDMS. Huawei is a large telecommunication and cloud services provider working with a broad range of customers. PolyDMS enables system architects to flexibly adapt to varying use case requirements and compose a dedicated DMS for each scenario with little overhead. Combined with the straight-forward integration and reuse of custom components, we argue that PolyDMS is a compelling framework for the use of composable DMSes in industry. We describe the details of our PoC system, including the initial requirements and how the different components interact, in Section 4.1. Section 4.2 then presents a preliminary performance evaluation of our PoC, which shows that composing multiple components into a new DMS can yield additional functionality *and* better performance.

4.1 Composition

We challenged PolyDMS with a real-world industry use case that existing DMSes cannot handle efficiently. Consequently, we composed a new DMS to solve this use case. As part of its service offering, Huawei is running a wide range of data analytics workloads on premise and in the cloud. In practice, solution architects often face fixed legacy DMS infrastructures that require a cross-platform system for utilizing multiple heterogeneous DMSes in one workload. Existing cross-platform systems [2, 15], however, do not support SQL interfaces or important logical optimizations like join ordering. Nevertheless, end users still request convenient interface via a language they know (i.e., SQL) and good query performance. Thus, the goal of our PoC was to compose a new cross-platform DMS with SQL support and an advanced query optimizer.

Figure 3 summarizes the architecture of the PoC system, while Program 1 shows how we envision an equivalent SDL statement. We use Apache Wayang (incubating) [29, 2] as a state-of-the-art cross-platform system together with Apache Calcite [6] for query parsing and optimization and our custom-developed metadata store
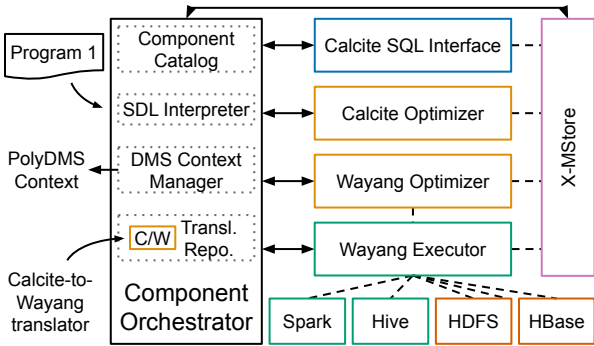
Figure 3: Overview of the PoC system architecture.

```python
def cross_platform_dms(input):
    ctx = PolyDMSContext()
    ctx.md = XMStore()
    ctx.qi = CalciteSqlInterface(input, ctx.md)
    ctx.log_opt = CalciteOpt(ctx.qi, ctx.md)
    ctx.cp_opt = WayangOpt(translate(ctx.log_opt), ctx.md)
    ctx.exec = WayangExec(ctx.cp_opt, ctx.md)

    return ctx.initialize()
```

Program 1: Prototypical SDL statement for the PoC system. Bold blue indicates dedicated SDL primitives.

X-MStore. Below, we discuss our choice of components as well as their interaction.

Query Interface. We use Calcite's SQL parser to implement the Query Interface service. The interface accepts SQL queries from the `PolyDMSContext` after the users calls its `execute()` method and responds with corresponding logical query plans. For query validation, we direct Calcite's metadata calls to our Metadata Store.

Metadata. We implement our PoC's metadata service by encapsulating X-MStore, our own cross-platform metadata store that collects information from any storage and execution engine. X-MStore assists other services requiring metadata from underlying systems. It provides methods for requesting different types of metadata, such as schema information and statistics, as well as for system-specific metadata, such as system load. X-MStore collects metadata in an offline fashion by leveraging existing information and by computing missing statistics. X-MStore enables us to perform additional optimizations across multiple execution platforms as it collects more information, such as histograms for join ordering, than existing solutions for data discovery. Using our own meta store in combination with existing components shows that PolyDMS makes it easy to integrate general- and special-purpose solutions.

Optimization. A key ability of our PoC is to perform logical optimizations (i.e., query rewrites) on top of a cross-platform execution system. We particularly aimed at performing selection/projection push-down and join reordering. After receiving a query plan from the Query Interface, the Component Orchestrator calls the Logical Optimizer service. We implement the Optimizer interface with Calcite, using its heuristic planner for logical optimization and directing Calcite's calls for statistics to X-MStore through the orchestrator. Our Logical Optimizer service receives requests with a logical plan,
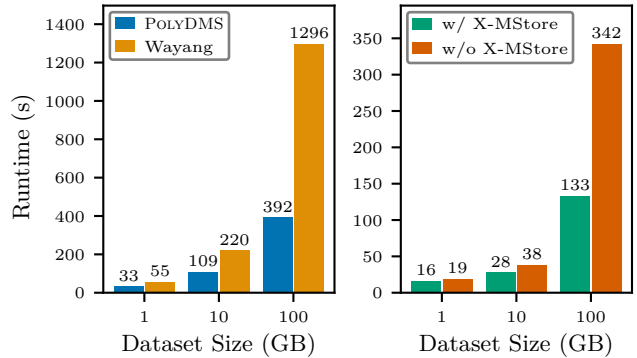
(a) Optimization synergies   **(b) Metadata impact**

Figure 4: Performance evaluation

feeds the query to Calcite's planner, and forwards the resulting optimized plan as a response to the Component Orchestrator.

In addition to Calcite, we employ a second optimization stage that aims at deriving an optimal combination of execution backends for a user query. To support optimization over multiple DMSes, we relied on Wayang [29] (formerly Rheem [2]). Given a platform-agnostic operator plan, Wayang's optimizer returns an annotated query plan where each operator is annotated with one of the available execution backends. To interact with Wayang, the Component Orchestrator first translates the Calcite plan into a Wayang plan using its Translator Repository. Like the logical optimizer, this service also interacts with the Metadata service to acquire platform and hardware information necessary for cross-platform optimization.

Execution. As Wayang's executor uses the same query representation format as its optimizer, we do not need another Translator between components. Instead, the Component Orchestrator directly forwards the optimization results to the execution service. As a cross-platform system, Wayang coordinates the execution across the selected data processing platforms by triggering the necessary sub-tasks on each execution backend. Choosing Wayang as an execution service allows us to take advantage of its conversion channel mechanism. In contrast to existing schedulers [18], this mechanism always chooses an optimal communication channel between two platforms and pipelines intermediate results if possible.

Orchestration. We bring all of the aforementioned components together by implementing the first Component Orchestrator prototype. The Component Orchestrator is a key part of PolyDMS and fulfills two functions in our prototype. First, it coordinates all components and controls the flow of information. Second, it manages the Translator Repository in which DMS composers define the conversions between different result representations. Maintaining all component translators in a central location allows for efficient reuse of the manually defined conversions.

4.2 Preliminary Evaluation

We evaluated our PolyDMS PoC with two goals in mind: First, to show the potential of PolyDMS and address one of the application scenarios, in particular the optimization synergies scenario; Second, to investigate if it is easy to integrate custom components. In the following, we first describe our setup and then discuss our results.

Experiment Setup. We implemented the Component Orchestrator in Java 8, and used HDFS 2.7, Spark 2.3, HBase 2.2, Hive 2.3, Wayang 0.6, as well as X-MStore. All of our components are accessible through a REST-API implemented with a Java Tomcat server. For our experiments, we used a proprietary dataset from our industry partner with three tables comparable to the TPC-H dataset and evaluate our prototype with the two following workloads.

Workload 1: Optimization Synergies. In our first experiment, we use a query that joins a table from HDFS with a table on HBase, applies a filter, a custom UDF, and an aggregation. We implemented a Wayang job as our baseline and an equivalent SQL query which we execute on our PoC with both logical and cross-platform optimizations. Our results in Figure 4a show that the logical optimizations always improve runtime performance compared to Wayang: it achieves an improvement factor of up to 3×.

Workload 2: Metadata Impact. In our second experiment, we use a query that joins three tables from HDFS. In this case, we execute the query once with and once without metadata support from X-MStore. Our results in Figure 4b show that while the right join ordering does not offer large improvement benefits for smaller dataset sizes, for larger dataset sizes we achieve an improvement factor of 2.5×. More interestingly, we observe that PolyDMS always achieves better runtime performance with X-MStore than without.

Discussion. It is worth noting that our PoC DMS does not exist today as a such: It provides a declarative query interface (SQL), combines traditional database optimizations with cross-platform optimizations, and executes incoming queries over multiple data processing platforms seamlessly. Overall, the preliminary evaluation of our PoC shows the potential of breaking monolithic architectures into multiple autonomous decoupled services that utilize existing components. More generally, the PolyDMS framework allows for several query planning and execution combinations (not available today), leading to significantly better runtime performance.

5 OUTLOOK

We presented PolyDMS, our vision to break today's monolith DMS architectures. The idea of PolyDMS aims at enabling users to compose new DMSes. It has three key characteristics: First, it employs standardized component type interfaces, which wrap existing or custom-developed components; Second, it comes with an orchestrator that manages the interaction between individual components, and; Third, it provides a SDL that allows users to compose a DMS out of standalone components. Based on our lessons learned from building the PoC and ongoing work, we plan to formalize our ideas for SDL and extend our Component Orchestrator prototype to construct DMS instances based on SDL statements. Furthermore, we want to investigate the potential of advanced control flow constructs in SDL and explore more real-world PolyDMS applications.

ACKNOWLEDGMENTS

REFERENCES

[1] D. Abadi, R. Agrawal, A. Ailamaki, et al. 2016. The Beckman Report on Database Research. *Commun. ACM*.

[2] D. Agrawal, M. Ouzzani, P. Papotti, et al. 2018. RHEEM: Enabling Cross-Platform Data Processing. *PVLDB*.

[3] A. Alexandrov, A. Kunft, A. Katsifodimos, et al. 2015. Implicit Parallelism through Deep Language Embedding. SIGMOD '15.

[4] R. Alotaibi, D. Bursztyn, A. Deutsch, et al. 2019. Towards Scalable Hybrid Stores: Constraint-Based Rewriting to the Rescue. SIGMOD.

[5] M. Armbrust, R. S. Xin, C. Lian, et al. 2015. Spark SQL: Relational Data Processing in Spark. SIGMOD '15.

[6] E. Begoli, J. Camacho-Rodríguez, J. Hyde, et al. 2018. Apache Calcite: A Foundational Framework for Optimized Query Processing Over Heterogeneous Data Sources. SIGMOD '18.

[7] K. S. Beyer, V. Ercegovac, R. Gemulla, et al. 2011. Jaql: A Scripting Language for Large Scale Semistructured Data Analysis. *PVLDB*.

[8] J. Camacho-Rodríguez, A. Chauhan, A. Gates, et al. 2019. Apache Hive: From MapReduce to Enterprise-grade Big Data Warehousing. SIGMOD '19.

[9] M. J. Carey, L. M. Haas, P. M. Schwarz, et al. 1995. Towards Heterogeneous Multimedia Information Systems: The Garlic Approach. RIDE-DOM '95.

[10] S. Chawathe, H. Garcia-Molina, J. Hammer, et al. 1994. The TSIMMIS Project: Integration of Heterogenous Information Sources. IPSJ '94.

[11] B. Dageville, T. Cruanes, M. Zukowski, et al. 2016. The Snowflake Elastic Data Warehouse. SIGMOD '16.

[12] J. Dean and S. Ghemawat. 2008. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM*.

[13] J. Duggan, A. J. Elmore, M. Stonebraker, et al. 2015. The BigDAWG Polystore System. *SIGMOD Rec.*

[14] H. Gavriilidis. 2019. Computation Offloading in JVM-based Dataflow Engines. BTW '19.

[15] I. Gog, M. Schwarzkopf, N. Crooks, et al. 2015. Musketeer: All for One, One for All in Data Processing Systems. EuroSys '15.

[16] L. M. Haas, J. C. Freytag, G. M. Lohman, and H. Pirahesh. 1989. Extensible Query Processing in Starburst. SIGMOD '98.

[17] S. Hagedorn, S. Kläbe, and K.-U. Sattler. 2021. Putting Pandas in a Box. CIDR '21.

[18] S. Haines. 2022. Workflow Orchestration with Apache Airflow. In *Modern Data Engineering with Apache Spark: A Hands-On Guide for Building Mission-Critical Streaming Applications*. Apress, 255–295.

[19] J. M. Hellerstein, M. Stonebraker, and J. Hamilton. 2007. Architecture of a Database System. *FNT in Databases*.

[20] Z. Kaoudi and J.-A. Quiane-Ruiz. 2018. Cross-Platform Data Processing: Use Cases and Challenges. ICDE '18.

[21] B. Kolev, C. Bondiombouy, P. Valduriez, et al. 2016. The CloudMdsQL Multistore System. SIGMOD '16.

[22] A. P. Marathe, S. Lin, W. Yu, et al. 2022. Integrating the Orca Optimizer into MySQL. EDBT '22.

[23] C. Olston, B. Reed, U. Srivastava, et al. 2008. Pig Latin: A Not-so-Foreign Language for Data Processing. SIGMOD '08.

[24] S. Palkar, J. J. Thomas, A. Shanbhag, et al. 2017. Weld: A Common Runtime for High Performance Data Analytics. CIDR '17.

[25] R. Sethi, M. Traverso, D. Sundstrom, et al. 2019. Presto: SQL on Everything. ICDE '19.

[26] M. A. Soliman, L. Antova, V. Raghavan, et al. 2014. Orca: A Modular Query Optimizer Architecture for Big Data. SIGMOD '14.

[27] M. Stonebraker and U. Çetintemel. 2005. "One Size Fits All": An Idea Whose Time Has Come and Gone. ICDE '05.

[28] J. Wang, T. Baker, M. Balazinska, et al. 2017. The Myria Big Data Management and Analytics System and Cloud Service. CIDR '17.

[29] Wayang authors. 2022. https://wayang.incubator.apache.org/.