CompoDB: A Demonstration of Modular Data Systems in Practice

Haralampos Gavriilidis
BIFOLD & TU Berlin
gavriilidis@tu-berlin.de

Lennart Behme
BIFOLD & TU Berlin
lennart.behme@tu-berlin.de

Christian Munz
TU Berlin
c.munz@campus.tu-berlin.de

Varun Pandey
BIFOLD & TU Berlin
varun.pandey@tu-berlin.de

Volker Markl
BIFOLD, TU Berlin & DFKI
volker.markl@tu-berlin.de

ABSTRACT

The increasing demand for specialization in data management systems (DMSes) has driven a shift from monolithic architectures to modular, composable designs. This approach enables the reuse and integration of diverse components, providing flexibility to tailor systems for specific workloads. In this demonstration, we present CompoDB, a framework for composing modular DMSes using standardized interfaces for query parsers, optimizers, and execution engines. CompoDB includes built-in benchmarking functionality to evaluate the performance of DMS compositions by systematically measuring trade-offs across configurations, including runtime differences and intermediate query plans. Attendees will interact with CompoDB through an interactive GUI that allows them to compose custom DMSes, execute queries, and observe detailed performance metrics in real time. Altogether, this demonstration highlights the practicality of modular DMS architectures and their potential for optimizing data management solutions for specific workloads.

1 INTRODUCTION

The evolution of data management systems (DMSes) has been driven by the recognition that "one size does not fit all". To address diverse needs, specialized systems such as columnar, graph, distributed, document, and other DMSes have emerged, each excelling in its respective domain. However, despite this diversification, the underlying architecture of most DMSes has remained largely unchanged since the 1980s: monolithic systems with tightly coupled components for query parsing, optimization, execution, and storage. This tightly coupled design not only increases development costs but also hampers innovation by making it difficult to adopt new technologies, adapt to evolving workloads, or avoid reinventing common functionalities. We argue that a *composable* approach to DMS design offers a solution by fostering reusable and extensible components – a concept gaining increasing attention within the database systems community [1, 6, 10, 12, 13].

Composable DMSes: Building on the need for modularity, composability has shown success in various contexts. Vertical composability emerged in Hadoop-era architectures, where compute is decoupled from storage – a paradigm adopted by modern systems like Spark and Snowflake. Hive [4] advanced this approach by integrating modular components, including the query interface (SQL), optimizer (Calcite), execution engine (Spark, Tez, MapReduce), and storage backend (HDFS). Horizontal composability has enabled federated systems like Presto [15] and cross-engine frameworks like Apache Wayang [2] to combine multiple DMSes for collaborative query processing. Despite these advancements, the "state of the composability union" in the DMS ecosystem remains weak, with components still being tightly coupled due to a lack of standardized interfaces and the significant manual effort required for integration.

The State of Composability: Prior work, including the "Composability Manifesto" [12] and our efforts on PolyDMS [6], introduced the composable DMS vision and demonstrated the benefits of composability. Building on these foundations, recent work has proposed the "Query Optimizer as a Service" [1] concept, further emphasizing the feasibility of modular architectures. The rise of open-source components, such as query interfaces (Ibis), optimizers (Calcite [3], Orca [16]), execution engines (DuckDB [14], DataFusion [7], Acero, Velox [11]), intermediate representations (Substrait [10]), and data formats (Parquet, Iceberg, Arrow), has also paved the way for modular DMS design. These advances lay the groundwork for realizing truly composable DMSes. However, key questions remain: How easy is it to integrate these components into cohesive systems? How effective are these composed systems with respect to performance?

The CompoDB Framework: We present CompoDB, a framework designed to enable the seamless assembly and evaluation of modular DMSes. Building on our previous work [6], CompoDB introduces standardized interfaces for integrating components such as query parsers, optimizers, and execution engines. It also includes benchmarking capabilities that systematically measure trade-offs across different configurations, providing insights into performance and workload-specific optimizations. With these capabilities, CompoDB establishes a foundation for modular "plug & play" DMS design and brings us closer to our vision: a framework that dynamically selects the optimal combination of components for any given workload.

Our Demonstration: We highlight the potential of modular DMS architectures through an interactive demo. Users can compose custom DMS configurations with the CompoDB GUI by selecting from supported query parsers and optimizers (e.g., Ibis, DuckDB, Calcite, DataFusion) as well as execution engines (e.g., DuckDB, Acero, DataFusion). The GUI also enables users to evaluate performance across diverse workloads and visualize results in real time, showcasing CompoDB's benchmarking functionality. This demonstration not only illustrates the ease of assembling modular DMSes but also provides actionable insights into how different configurations impact performance, allowing attendees to explore the trade-offs of modular system design.

2 BACKGROUND AND RELATED WORK

We summarize the evolution of DMSes from monolithic architectures to composable designs, highlighting the concepts of horizontal and vertical composability, and discuss related work.

2.1 From Monoliths to Composable Systems

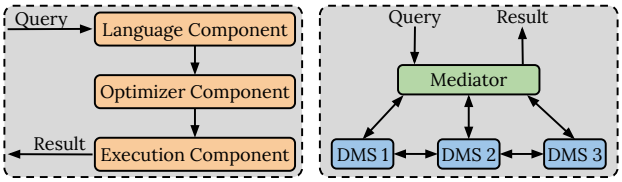
The landscape of DMSes has evolved significantly over the past decades. Traditionally, DMS architectures were designed as monolithic, vertically integrated systems that bundled all key components, such as query optimizers, execution engines, and storage backends, into a single tightly-coupled stack. While these systems provided a unified solution for many use cases, they include drawbacks, e.g., limited reusability for new requirements, high maintenance costs, and a lack of flexibility to adapt to diverse workloads. Moreover, building such systems required a significant investment of time and resources, often requiring 5-10 years of dedicated effort by expert engineers or the support of a large open-source community. Consequently, innovating in this space remained challenging.

2.2 Horizontal and Vertical Composability

The concept of composability in DMSes can be broadly categorized into two paradigms: horizontal and vertical composability (see Figure 1). Each serves distinct purposes and addresses different challenges in the evolving landscape of data management. For a more in-depth discussion, see our prior work in [6].

Horizontal composability involves integrating multiple systems to collaboratively process a single query or job. This concept originated with federated databases, which were designed not to solve the monolith problem but to enable data integration from multiple heterogeneous systems. Federated databases aimed to cover broader use cases by allowing users to query across different data sources where individual systems alone were insufficient. Modern systems, such as Presto [15], extend this approach by scaling out the computation on distributed environments. Apache Wayang [2] offers cross-engine execution functionalities, i.e., combining multiple systems for single queries, to either enable new use-cases or improve performance. In summary, horizontal composability focuses on expanding the scope of data access and query processing across platforms rather than addressing architectural modularity.

Vertical composability, on the other hand, addresses the challenge of building modular systems by composing individual components, such as query optimizers, execution engines, and storage backends, into a unified stack. Unlike horizontal composability, which spans across independent systems, vertical composability operates within a single system's architecture, decoupling components to increase extensibility and reusability. For example, a vertically composed system might use Apache Calcite as a pluggable optimizer, coupled with an execution engine like Velox [11] or DuckDB [14], allowing system architects to create custom configurations tailored to specific workloads. The ongoing industry investments in composable components from companies such as Meta (Velox [11]) or Microsoft (QOaaS [1]) demonstrate the significant interest in composable DMSes. BOSS [9] is another example of enabling efficient vertical composability across compute kernels. Widely used systems like Apache Hive [4], which seamlessly integrates a query interface, optimizer, and execution engine, and Taurus, which embeds the Orca optimizer into MySQL [8], further demonstrate the potential of vertical composition. However, these systems remain tightly coupled and monolithic, limiting their flexibility to adopt new components or address emerging requirements.



(a) Vertical composition. **(b) Horizontal composition.**

Figure 1: Composability approaches (adapted from [6]).

Together, horizontal and vertical composability highlight two complementary approaches to addressing the complexity and diversity of modern data management. Horizontal composability emphasizes integration and interoperability across systems, whereas vertical composability focuses on reusability and modularity within a single system's architecture. While some of our recent work has focused on horizontal composability [5], this demo centers on vertical composability, exploring the potential of composable architectures for specialized OLAP DMSes.

2.3 Open-Source DMS Component Evolution

The growing appreciation for composable DMS architectures is primarily driven by open-source technologies that enable modularity and interoperability. IRs, such as Substrait [10], standardize logical and physical query plans, thus allowing seamless communication between query optimizers and execution engines. This standardization minimizes the integration overhead and accelerates experimentation with modular components. Frontends and query languages like Ibis and ZetaSQL further enhance modularity by translating user queries into IRs, providing a consistent interface across systems and enabling integration with diverse backends. Advanced query optimizers, including Apache Calcite, DataFusion's optimizer, and DuckDB's optimizer, decouple optimization from execution, offering flexibility in system configurations. On the execution side, reusable engines such as Acero, DataFusion [7], DuckDB [14], and Velox [11] employ advanced techniques like vectorized processing and compressed (encoding-aware) execution to maximize performance. Additionally, these engines dynamically adapt to runtime statistics, thereby optimizing execution for diverse workloads. Between the optimization and execution layers, Substrait ensures compatibility, therefore simplifying the integration process. By integrating standardized IRs, modular query frontends, advanced optimizers, and efficient execution engines, vertically composable architectures provide a strong foundation for specialized DMSes.

Limited Support for Metadata: Despite the overall traction and progress in open-source DMS components, ad-hoc composability of DMSes does not come without a cost. The lack of metadata standards (e.g., for sharing data statistics across components) poses a significant challenge for optimizing query plans and executing queries efficiently. Currently, DMS optimizer's differ in how they retrieve statistics. For example, DuckDB and DataFusion require tables to be registered, while Calcite has its own metadata provider. Therefore, we argue that our community needs to develop metadata standards similar to how Substrait standardizes query plan representations to fully unlock the potential of composable DMSes.

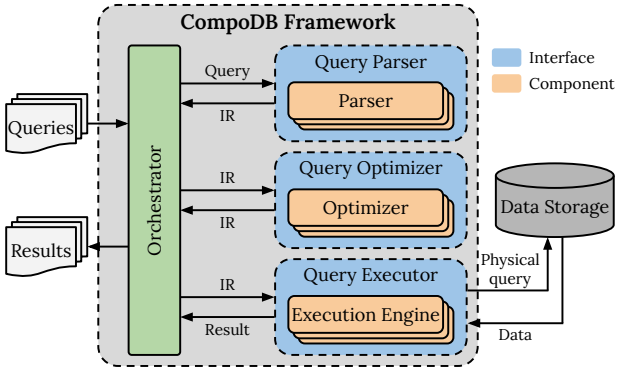


Figure 2: CompoDB architecture.

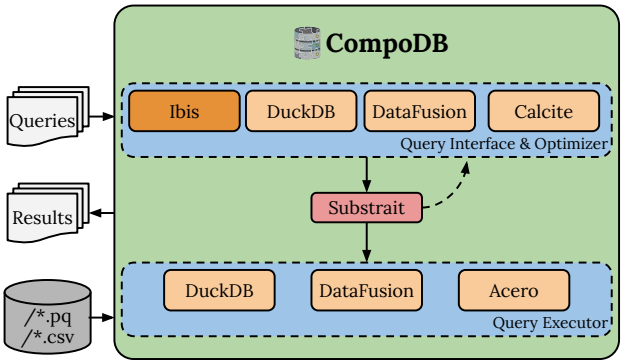


Figure 3: CompoDB prototype implementation. Query interfaces are grouped for conciseness since all but Ibis can also act as optimizers. Optimizers may be chained.

3 COMPODB

CompoDB[1] is a framework for vertical DMS composition designed to integrate modular components such as query optimizers and execution engines into custom systems. By leveraging standardized interfaces and a shared IR, CompoDB facilitates flexibility, extensibility, and tailored performance for diverse workloads.

3.1 Overview

Architecture: We show CompoDB's architecture in Figure 2. At its core, the framework consists of an orchestrator that manages workflows, connects components, and coordinates query execution. It allows users to compose unique systems tailored to their workload requirements by selecting a combination of components, which communicate using a common IR. The orchestrator processes queries starting with query submission, followed by parsing, optimization, and execution.

Composing a System: System architects can compose systems via configuration files, which the orchestrator uses to instantiate a tailored DMS. A configuration file defines the sequence and types of components to be used. In the simplest case, users specify a query parser, optimizer, and execution engine. However, CompoDB also enables users to experiment with other configurations, such as handling query parsing and optimization in a single component or using multiple optimizers in sequence. CompoDB's clear boundaries and interfaces enable flexible and rapid composition of systems without requiring deep integration efforts.

Integrating New Components: To integrate new components into CompoDB, system architects may implement one of three predefined interfaces for query parsing, optimization, and execution.

- Query parser: Accepts queries written in a domain-specific or general-purpose language and produces a query plan. A component implementing this interface could optionally also apply optimization techniques, such as operator push-down.
- Query optimizer: Accepts a query plan and produces an optimized query plan, applying arbitrary optimization techniques. CompoDB offers the flexibility to combine multiple optimization strategies encapsulated in different components, such as adding a cost-based optimizer on top of a rule-based optimizer.
- Query executor: Accepts a query plan, compiles it into an internal execution plan, executes it, and returns query results.

By abstracting the internal details of each component, the IR facilitates interoperability and simplifies the integration of diverse components. This enables CompoDB to compose novel DMS designs, such as combining the imperative Python-based query interface of Ibis with the query optimizer of DataFusion and the vectorized execution engine of DuckDB.

3.2 Prototype Implementation

Figure 3 illustrates the prototype implementation of CompoDB, in which we implemented the query parsing, optimization, and execution interfaces for several open-source components and used Substrait as the common IR. For query parsing, we integrated Ibis, DuckDB, DataFusion, and Calcite, whereby all but the first can also act as optimization components. Not all optimizers accept Substrait plans as input as of now, wherefore some of these components currently work as integrated parser-optimizer solutions. For the query executor interface, we integrated the execution engines of DuckDB, DataFusion, and Acero, all of which can consume Substrait plans. Our prototype can be easily extended with additional components that support Substrait (de)serialization by implementing one of the three interfaces discussed in Section 3.1.

3.3 Benchmarking Composed DBMSes

To evaluate the performance of different component compositions, we enhanced CompoDB with a benchmarking framework. This framework allows users to test various configurations of optimizers and execution engines, input diverse queries, and specify data file formats, e.g., Parquet or CSV. The framework automates the benchmarking process and captures metrics such as execution runtime and resource utilization to showcase performance trade-offs across configurations. These insights ultimately enable the design of workload-specific composed DMSes.

Our preliminary experimental investigation produced interesting findings, an excerpt of which is illustrated in Figure 4. The execution time measurements show that for several TPC-H queries, combining the Ibis parser and DuckDB engine outperformed chaining DuckDB's own optimizer and execution engine through Substrait[2], indicating that diverse compositions can indeed be beneficial. Inspecting the intermediate Substrait plans revealed that the optimizers generated different plans. For instance, DuckDB's and DataFusion's optimizers introduced additional projection operators in Q3, leading to performance overhead. In other queries, the differences stemmed from varying join orders, which we attribute to the lack of metadata access.

[1] github.com/polydbms/composable-dms

[2] We note that Substrait support in DuckDB is currently experimental.

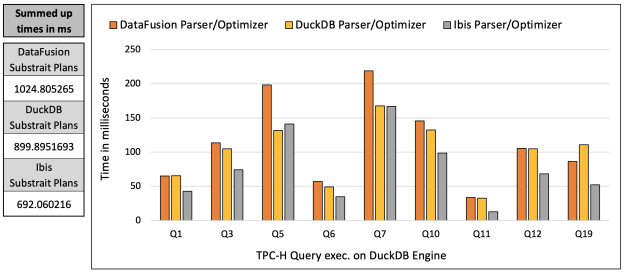


Figure 4: CompoDB experiments with DuckDB engine and different optimizers on selected TPC-H queries (sf 10).

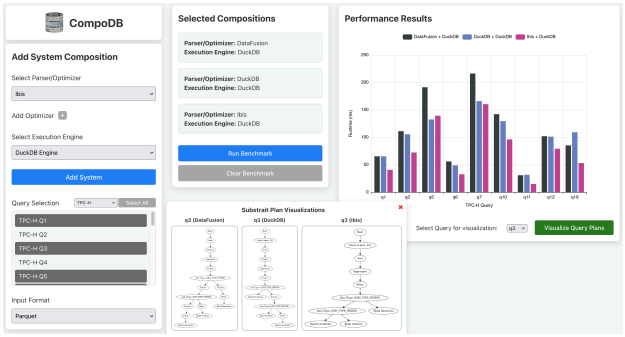


Figure 5: CompoDB benchmark GUI. Allows composing systems by selecting DMS components, benchmarking created compositions, and visualizing intermediate query plans.

4 DEMONSTRATION SCENARIOS

Our demonstration offers three dedicated scenarios that focus on different aspects of composable DMSes: integrating new components, measuring performance tradeoffs between compositions, and analyzing query plans to understand performance differences. Together, these scenarios make the end-to-end process of vertical DMS composition more accessible to system architects.

Ease of Using CompoDB Interfaces: In the first part of the demo, we will showcase how CompoDB simplifies the integration of modular DMS components through well-defined interfaces. We will walk through the code to demonstrate the ease of implementing the query parsing, optimization, and execution interfaces, as well as their interaction via the shared IR. Essentially, adding a component involves implementing a substrait producer (for parsing and optimization) or consumer (for execution). By showing how new components can be plugged into the framework with minimal effort, we highlight CompoDB's flexibility and extensibility. This scenario emphasizes how developers can rapidly integrate existing systems and experiment with new components.

Instantiating and Benchmarking Systems: The second part of the demo will showcase CompoDB's user interface, which allows attendees to interactively create system compositions, select queries, and run benchmarks (see Figure 5 on the left). Users can compose their own system flavor and select a set of queries from the TPC-H benchmark to start a performance analysis. Conceptually, CompoDB supports all queries that can be expressed in the newest Substrait version, making it highly flexible. While some compositions fully support standard TPC-H queries, others may not work due to differences in Substrait versions and its evolving support across systems. To ensure compatibility, we have carefully curated a set of slightly modified TPC-H queries that work seamlessly across all Substrait producers and consumers. Additionally, we encourage the audience to experiment with their own queries, which can be easily added to CompoDB's benchmarking framework. Once the benchmark runs are completed, the GUI visualizes the results, i.e., query runtime, allowing users to compare the performance of different system compositions. This scenario demonstrates the practicality of composable systems and provides insights into tradeoffs across various configurations.

Visualizing CompoDB Substrait Plans: In the final part of the demo, we will explore the Substrait query plans generated by each composition for the selected benchmark queries. This visualization helps to explain why specific query and system configurations exhibit different behaviors. By analyzing the query plans, attendees can identify how optimizers and execution engines affect query execution and how they each excel for different workloads. For example, the popup window of CompoDB's GUI in Figure 5 shows the three different Substrait query plans produced by DataFusion, DuckDB and Ibis for TPC-H Q3. This scenario highlights CompoDB's ability to benchmark system compositions while offering insights into query processing by enabling comparisons of query plans across different optimizers.

ACKNOWLEDGMENTS

We gratefully acknowledge funding from the German Federal Ministry of Education and Research under the grants BIFOLD24B and 01IS17052 (as part of the Software Campus projects PolyDB and FDaaS).

REFERENCES

[1] R. Alotaibi, Y. Tian, S. Grafberger, et al. 2025. Towards Query Optimizer as a Service (QOaaS) in a Unified LakeHouse Ecosystem: Can One QO Rule Them All?. In *CIDR*. https://doi.org/10.48550/arXiv.2411.13704

[2] K. Beedkar, B. Contreras-Rojas, H. Gavriilidis, et al. 2023. Apache Wayang: A Unified Data Analytics Framework. *SIGMOD Rec.* 52, 3 (2023), 30–35. https://doi.org/10.1145/3631504.3631510

[3] E. Begoli, J. Camacho-Rodríguez, J. Hyde, et al. 2018. Apache Calcite: A Foundational Framework for Optimized Query Processing Over Heterogeneous Data Sources. In *SIGMOD*. https://doi.org/10.1145/3183713.3190662

[4] J. Camacho-Rodríguez, A. Chauhan, A. Gates, et al. 2019. Apache Hive: From MapReduce to Enterprise-Grade Big Data Warehousing. In *SIGMOD*. https://doi.org/10.1145/3299869.3314045

[5] H. Gavriilidis, K. Beedkar, J. Quiané-Ruiz, and V. Markl. 2023. In-Situ Cross-Database Query Processing. In *ICDE*. https://doi.org/10.1109/ICDE55515.2023.00214

[6] H. Gavriilidis, L. Behme, S. Papadopoulos, et al. 2022. Towards a Modular Data Management System Framework. In *CDMS@VLDB*. https://cdmsworkshop.github.io/2022/Proceedings/ShortPapers/Paper2_HaralamposGavriilidis.pdf.

[7] A. Lamb, Y. Shen, D. Heres, et al. 2024. Apache Arrow DataFusion: A Fast, Embeddable, Modular Analytic Query Engine. In *SIGMOD*. https://doi.org/10.1145/3626246.3653368

[8] A. P. Marathe, S. Lin, W. Yu, et al. 2022. Integrating the Orca Optimizer into MySQL. In *EDBT*. https://doi.org/10.48786/EDBT.2022.45

[9] H. Mohr-Daurat, X. Sun, and H. Pirk. 2023. BOSS-An Architecture for Database Kernel Composition. *PVLDB* 17, 4 (2023), 877–890. https://doi.org/10.14778/3636218.3636239

[10] J. Nadeau and W. McKinney. [n.d.]. Substrait: Cross-Language Serialization for Relational Algebra. https://substrait.io

[11] P. Pedreira, O. Erling, M. Basmanova, et al. 2022. Velox: Meta's Unified Execution Engine. *PVLDB* 15, 12 (2022), 3372–3384. https://doi.org/10.14778/3554821.3554829

[12] P. Pedreira, O. Erling, K. Karanasos, et al. 2023. The Composable Data Management System Manifesto. *PVLDB* 16, 10 (2023), 2679–2685. https://doi.org/10.14778/3603581.3603604

[13] P. Pedreira, D. Majeti, and O. Erling. 2024. Composable Data Management: An Execution Overview. *PVLDB* 17, 12 (2024), 4249–4252. https://doi.org/10.14778/3685800.3685847

[14] M. Raasveldt and H. Mühleisen. 2019. DuckDB: an Embeddable Analytical Database. In *SIGMOD*. https://doi.org/10.1145/3299869.3320212

[15] R. Sethi, M. Traverso, D. Sundstrom, et al. 2019. Presto: SQL on Everything. In *ICDE*. https://doi.org/10.1109/ICDE.2019.00196

[16] M. A. Soliman, L. Antova, V. Raghavan, et al. 2014. Orca: A Modular Query Optimizer Architecture for Big Data. In *SIGMOD*. https://doi.org/10.1145/2588555.2595637