Polypheny-DB: Towards Bridging the Gap Between Polystores and HTAP Systems

Marco Vogt(✉), Nils Hansen, Jan Schönholz, David Lengweiler, Isabel Geissmann, Sebastian Philipp, Alexander Stiemer, and Heiko Schuldt

Databases and Information Systems Research Group, Department of Mathematics and Computer Science, University of Basel, Basel, Switzerland
{marco.vogt,nils.hansen,jan.schonholz,david.lengweiler,isabel.geissmann, sebastian.philipp,alexander.stiemer,heiko.schuldt}@unibas.ch

Abstract. Polystore databases allow to store data in different formats and data models and offer several query languages. While such polystore systems are highly beneficial for various analytical workloads, they provide limited support for transactional and for mixed OLTP and OLAP workloads, the latter in contrast to hybrid transactional and analytical processing (HTAP) systems. In this paper, we present Polypheny-DB, a modular polystore that jointly provides support for analytical and transactional workloads including update operations and that thus takes one step towards bridging the gap between polystore and HTAP systems.

Keywords: Polystore database · Transactional workload · Adaptivity

1 Introduction

In recent years, polystore systems have gained increasing interest in the database research community. They aim to solve the demand for faster processing of heterogeneous workloads on massively growing volumes of data.

The idea of a polystore system is to combine polyglot persistence and multistore database systems. Polyglot persistence deals with the challenge of choosing the right tool (i.e., query language) for a concrete use case: When storing data used by different types of applications, it might also be beneficial to use different query languages for retrieving the data. Multistore systems, in turn, are systems which manage data across heterogeneous data stores. All data is accessed through one interface which supports one query language. A system combining polyglot persistence and multistore data management is called a *polystore* [13].

Today's IT landscapes usually consist of several applications (e.g., accounting, web shop, warehousing, product recommendation engines, newsletters, etc.). Often, these applications have not been developed in-house but have been acquired from different software companies. These applications therefore likely expect different data models and require support for different query languages.

© Springer Nature Switzerland AG 2021
V. Gadepally et al. (Eds.): Poly 2020/DMAH 2020, LNCS 12633, pp. 25–36, 2021.
https://doi.org/10.1007/978-3-030-71055-2_2

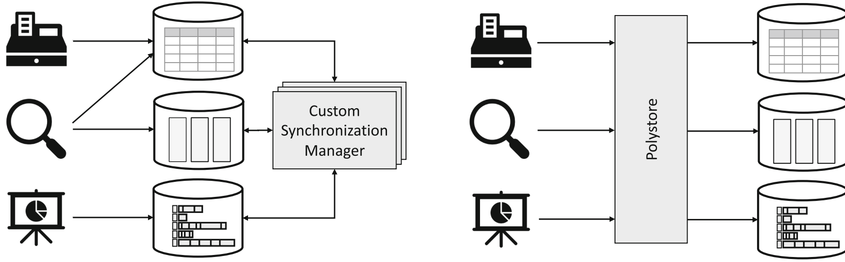

Fig. 1. Replacing a setup consisting of applications directly accessing specialized data storage solutions (left) with a polystore solution deployed between applications and specialized data storage solutions (right).

Often such applications are already shipped with an optimized data storage engine included.

While these optimized data storage engines deliver good performance for the workload of the application they shipped with, this approach only works if the data can be strictly partitioned and separated for the different applications. If this is not the case (which is rather the norm than an exception), at least parts of the data needs to be synchronized using some sort of synchronization approach, as depicted in Fig. 1 (left). Such a setup is not satisfying for multiple reasons: (i) It prevents a close integration of all applications. (ii) The applications cannot run on the latest data. And (iii) there is always the risk of inconsistencies if there is more than one application modifying data.

A possible solution for this scenario is to deploy a polystore system between the applications and the specialized data stores (see Fig. 1, right). Like a federated database system, a polystore allows to access data from different data stores. This ensures a much tighter integration of the application layer and reduces overhead and redundancy in data storage and thus the need for additional synchronization.

In the recent years, different polystore systems have been developed. While they show impressive results in accelerating analytical workloads [4,10], they cannot be used for the scenario depicted in Fig. 1. The main reason is that existing polystores lack support for data manipulation (DML) operations. Since polystore systems are limited to read-only analytical workloads, parallel transactional queries would have to be directly submitted to the underlying data stores, circumventing the polystore layer. This is infeasible, and executing long-running analytical workloads would have a huge impact on the transactional performance of these data storage systems. Hence, a dedicated layer would be needed that is aware of all queries (transactional and analytical); otherwise, it is not possible to prioritize and schedule long-running queries. Furthermore, executing queries on different data stores with parallel workloads can lead to inconsistent data.

Copying transactional data on a regular basis to the polystore solves these problems but does not necessarily allow querying the latest data. This makes it useless for most operational analytical queries requiring the latest data. But

also for long-running queries which do not require access to the latest data, this approach causes downtimes of the analytical systems while updating the data. This update process can also massively impact the performance of productive systems. Furthermore, this approach does not offer federated access for transactional workloads.

In parallel to polystores, recently also HTAP (hybrid transactional/analytical processing) systems have gained popularity. These systems provide good performance for mixed transactional and analytical workloads. While typically also offering support for data manipulation, these systems lack common features of a polystore system, especially polyglot persistence and support for a wide variety of workloads since they are usually earmarked for concrete applications and their characteristic workloads.

To deal with mixed, heterogeneous, and dynamically changing workloads (e.g., as they occur in scenarios as the one depicted in Fig. 1) requires a diverse and flexible polystore system that, at the same time, provides the transaction capabilities of an HTAP system. Because there is no off-the-shelf solution for every use case, such an 'HTAP-aware polystore' needs to be flexible in order to allow adding support for a large set of query languages and interfaces as well as underlying data stores.

In this paper we present Polypheny-DB, an implementation of the concepts formerly introduced in [15]. Most importantly, Polypheny-DB goes beyond existing polystores and also adds support for transactional workloads.

The contribution of this paper is twofold: (i) We present the implementation of Polypheny-DB which is to our best knowledge the first polystore system that tries to bridge (or at least narrow) the gap between polystore and HTAP systems and hence jointly offers full support for DML and DDL statements. (ii) We show that the implementation of Polypheny-DB follows a modular approach which can be easily adapted at runtime in various ways (in terms of data stores, data distribution and deployment, and/or query workloads).

The remainder of this paper is structured as follows: Sect. 2 surveys related work. Section 3 introduces Polypheny-DB, Sect. 4 focuses on the flexibility and adaptivity of the system, thanks to its modular architecture, and Sect. 5 concludes.

2 Related Work

The last years have seen a vast proliferation of different types of polystore systems [13]. Every system has a different focus and is developed with a different use case in mind. But all these systems have in common that they provide access to data stored in heterogeneous data stores. In what follows, we introduce and compare selected systems regarding (i) their support for DML and DDL queries, (ii) their architecture in terms of modularity, and (iii) their potential for runtime adaptation.

BigDAWG is one of the pioneer polystore systems. It organizes heterogeneous data stores into "islands" (e.g., relational or array islands) [4]. Every island has an associated query language and data model and connects to one or more data stores. BigDAWG only supports cross-data store queries within the same island. Inter-island queries are not possible without migrating the data first.

The BigDAWG system delivers great results [11] for heterogeneous read-only workloads. However, we did not find any information regarding support for DML or DDL queries. Hence, data needs to be imported into the underlying data stores prior to the start of the system.

BigDAWG's architecture is expandable and allows new data stores and islands to be added, but due to the fact that this requires changes to the BigDAWG source files, this is not "plug-and-play".

CloudMdsQL is a multistore system [9] that uses an SQL-like query language for querying heterogeneous data stores. It allows subqueries to contain embedded invocations to each data store's native query interface. The authors present an implementation which translates queries into query execution plans and a query engine based on Apache Derby for executing these plans. The focus of their work is on the query language which allows exploiting the full capabilities of the underlying data stores.

The CloudMdsQL system is read-only and has no support for DDL operations. Because only the compiler which parses a CloudMdsQL query and generates the optimized query execution plan is available for download, we cannot discuss the modularity and runtime adaptability of the overall CloudMdsQL system.

Apache Drill [7] is a distributed query engine for interactive analysis of large-scale datasets. The query engine accepts ANSI SQL and MongoDB QL, and supports a variety of NoSQL systems like HBase, Hive, and MongoDB. It also offers support for using relational databases as underlying data stores.

Drill is developed with a focus on extensibility and offers support for pluggable query languages, query planners and query optimizers. It allows new data sources to be added at runtime. Designed as a system targeted on data analytics, it does not support DML queries and its DDL support is limited to linking new data sources.

Hybrid Transactional and Analytical Processing. (HTAP) systems maintain the same data in different storage formats. By leveraging multiple query processing engines, these systems are able to improve performance of read-only workloads [5]. This allows to efficiently perform real-time analytics and transactional workloads on data stored in the same system. Examples for such systems are SAP HANA [3], HyPer [8], and Hyrise [6]. While these HTAP systems are able to provide excellent performance for mixed OLTP and OLAP workloads, they do not qualify as polystores because they do only support one query language.

SnappyData [12] aims to combine the benefits of HTAP systems with the ones of a big data analytics engine. SnappyData uses GemFire as an in-memory transactional data store and combines it with Apache Spark. Having Apache Spark as foundation makes SnappyData very flexible and adds support for a wide range of data stores. It also comes with support for DML and DDL statements. But in contrast to Polypheny-DB, SnappyData treats data stores differently. Data sources in SnappyData are primarily used as source to load data from. While queries on these data sources can be executed in general, this support is limited to the capabilities of the underlying data stores. SnappyData neither allows to replicate data to different data stores nor to partition data.

3 Polypheny-DB

Most polystores are designed with the concept of federated database systems in mind. They enable access to heterogeneous data stored on independent data stores using different data models. The data is usually stored on exactly one of these underlying data stores. As outlined in Sect. 2, these systems are optimized for long-running, read-only queries and offer at most very limited support for DML queries. They furthermore lack support for other typical features of a database management system like transactional execution guarantees.

The focus of existing polystores on long-running queries is fully justified because this is the field where a polystore can massively accelerate query performance as shown in [4,10,14]. Due to the nature of a polystore, it is hardly possible to accelerate simple transactional queries. The challenge is here to keep the overhead as small as possible while still focusing on read-only accesses.

While HTAP systems provide good performance for mixed OLTP and OLAP workload, they miss other important aspects of polystore systems required for the use case depicted in Fig. 1, especially the polyglot persistence. Moreover, in practice, more types of workloads than transactional and analytical can be found and have to be dealt with (e.g., multimedia retrieval or expert systems). Handling these requires the multi-engine and multi-model approach of a polystore system.

We therefore see the demand for a polystore system bridging the gap between HTAP systems and existing polystore systems. Such a system needs to accept queries expressed in the query languages and through the query interfaces demanded by the applications and must be able to efficiently deal with all kinds of workloads generated by the applications. As outlined before, it is furthermore crucial to jointly provide support for DML queries and transactions.

While SnappyData [12] presents an interesting approach to integrate polystore aspects into an HTAP system by combining GemFire with external data sources, with Polypheny-DB we like to introduce an alternative approach. In contrast to SnappyData, Polypheny-DB aims to fully exploit and combine the optimization provided by the underlying data stores. By replicating data to multiple data stores, Polypheny-DB is able to optimize for different types of workloads.

With Polypheny-DB we try to fill the gap between polystore systems and HTAP systems and present a system with a high degree of modularity and

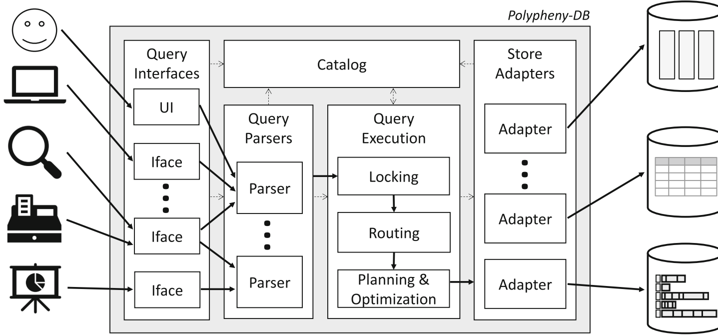

Fig. 2. Simplified architecture of Polypheny-DB. The arrows indicate the execution paths queries can take; the dashed arrows indicate internal communication paths between the modules.

support for adaptions at runtime. In this section, we provide an overview of Polypheny-DB's architecture and design decisions.

3.1 Architecture

Polypheny-DB has been developed with a strong focus on modularity and runtime adaptiveness. Figure 2 depicts the architecture of Polypheny-DB.

Queries are accepted through multiple *query interfaces* supporting one or multiple query languages. A query received through a query interface is forwarded to the matching *query parser*. The parser translates the query into a *logical query plan*. This plan is represented as a tree of relational operators based on an extended relational algebra. This algebra serves as a unified query language and allows to express all query features currently supported by Polypheny-DB.

This logical query plan is then processed by the *locking* module to guarantee the isolation of transactions. When all locks are acquired, the query is processed by the *query router*. This module decides on which of the underlying data stores a query should be executed. This is required because data can be replicated to multiple data stores.

The *routed query plan* is then optimized and translated into a *physical query plan*. This physical query plan contains implementations of the relational operators. Finally, the *data store adapters* take care of translating the physical query plan into the native query language of the underlying data stores. There can be multiple adapters involved in executing a query.

3.2 Schema Management

In Polypheny-DB, we distinguish between two types of schemas: (i) The *logical schema* which represents the structure available to the user, and (ii) the *physical schema* which is maintained on the underlying data stores. The logical schema is expressed and maintained in a relational data model.

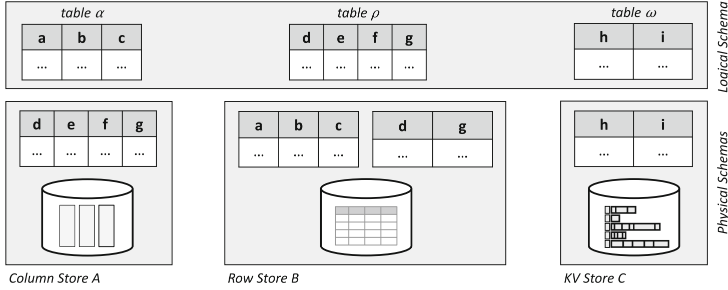

Fig. 3. The columns in tables α and ω only have one placement on the underlying data stores. Table ρ is partially stored on two data stores. All columns of ρ are placed on data store A whereas the columns d and g are additionally stored on data store B.

A major difference of Polypheny-DB compared to other polystore systems is that data stores are not treated as static data sources for a specific dataset but instead as execution engines where data can be placed on. Individual columns or whole tables can be replicated and stored on multiple data stores at the same time. This combination of data replication and vertical partitioning allows to optimize for different types of queries accessing the same data while at the same time minimizing the overhead for DML queries and economically use the available memory and disk space. In Polypheny-DB, the physically stored data is called *placement*. The placements are maintained at the column level. New placements can be added and removed at runtime. Every column needs to have at least one placement. Figure 3 depicts a possible configuration of Polypheny-DB.

A logical schema therefore needs to be mapped to different physical schemas at the same time. The user always interacts with the logical schema, independent of the physical schema(s) the data is represented in. This is especially important when it comes to schema changes which are supported by Polypheny-DB as well.

Another major difference to most existing polystore approaches is the support for schema modifications. The logical schema can be modified at runtime through every query interface supporting schema changes. Currently, the SQL interface and the browser-based user interface come with support for DDL operations. The schema is persistently stored and maintained in the internal catalog. This allows the system to be restarted and all persistent placements to be restored.

3.3 Implementation

Polypheny-DB is implemented in Java. The source is available under the Apache open source license on Github[1]. Major parts of Polypheny-DBs query processing engine are based on Apache Calcite [1]. This allows Polypheny-DB to make use of a reliable cost-based query optimizer also used in several other projects.

[1] https://github.com/polypheny/Polypheny-DB.

Query Execution. In the query planning and optimization process, the query is translated into a tree of physical operators. These operators are later translated into the native query language of the underlying data store(s). For every operator (except for the scan operator) exists an implementation which allows every query to be executed purely within Polypheny-DB.

The scan operator—which allows to sequentially read all data from a specified entity—is provided by the data store adapters. In order to make use of the specific optimizations of the underlying data stores, every adapter can additionally provide implementations of the other physical operators.

Polypheny-DB tries to "push down" as many operators to the underlying data stores as possible by using the provided operators. However, if parts of the query plan cannot be "pushed down" (e.g., joining data from different data stores), Polypheny-DB relies on its own implementations. Furthermore, this approach allows to provide support for all query features independent of the data stores on which the data is physically stored.

Transactions. Due to the fact that Polypheny-DB supports DML queries, it requires locking for ensuring the consistency of the data and the correctness of concurrent queries. For the isolation of concurrent queries, we use strong strict two-phase locking [2]. Further, Polypheny-DB provides full transaction support, but only if the underlying data stores offer support for transactions. The system can therefore provide durability—and together with the other guarantees full ACID support—if there is at least one placement for all involved data on a persistent data store.

Data Types. Polypheny-DB supports different integer, float, date, and character types. Within Polypheny-DB, the data is represented and processed using Java types. The mapping to the native data types of the underlying data stores is defined individually for every store. Additionally, Polypheny-DB supports arrays of all supported data types with an arbitrary cardinality and dimension.

4 The Modular Polystore

In this section we particularly focus on the modularity of Polypheny-DB's architecture and its support for DML and DDL statements.

4.1 Query Interfaces

Polypheny-DB can be queried through multiple query interfaces. With the JDBC-SQL interface, we provide an industry standard solution for querying the system using SQL. It offers support for retrieving meta data and for controlling transactions. SQL is the most mature query language supported by Polypheny-DB providing all available query features. It can also be used to manage data stores and other system configurations and the schemas using DDL queries.

With Polypheny-UI, the system has a user interface for managing the schema, modifying the system's configuration, monitoring the status, browsing and modifying the data, managing the underlying data stores, and querying the system by using different query methods and languages.

Polypheny-DB applies the idea of polyglot persistence not only to the data access but also to the management of the system itself. This is achieved by providing two methods for modifying the schema and the system configuration.

Beside JDBC-SQL and the user interface, Polypheny-DB provides a REST-based query interface for accessing and modifying data. Polypheny-DB is also able to directly express queries using its relational algebra representation. This is either possible by submitting the query plan as JSON or by using the graphical builder provided in the Polypheny-UI.

With an Explore-by-Example interface and the dynamic query builder, Polypheny-DB also supports innovative query methods. Explore-by-Example allows to select rows of a result set which should or should not be part of the final result. Polypheny-DB then derives a query fulfilling these requirements. The dynamic query builder assists users in formulating queries containing joins and filters. This is done by dynamically generating a user interface containing specific controls based on a statistical analysis of the currently stored data.

To adapt the system according to the specific requirements of a concrete use case, query interfaces can easily be added, modified, or removed. Furthermore, the modular architecture simplifies the implementation of new query interfaces.

4.2 Query Routing

Query routing is the process of selecting on which of the underlying data stores a query or parts of it should be executed. As depicted in Fig. 3, tables and columns can be partitioned and replicated to multiple underlying data stores.

DML queries always need to be executed eagerly on *all* underlying data stores holding a placement of the modified data. This includes the insertion of new rows. As a consequence, the most suitable placement for executing read-only queries can be arbitrarily selected.

Polypheny-DB currently supports two query routing implementations. The first implementation models the behavior of only storing the data on one of the data stores. The data store is selected when creating the table. This implementation requires no sophisticated query analysis to select the best store and is therefore a suitable base line to compare with the second implementation.

This second implementation is based on the approach introduced in [14]. The basic assumption of this approach is that queries generated by applications are usually derived from templates. Because queries derived from the same template are similar regarding execution time on a specific underlying data store, it is possible to route queries based on structural similarity of queries. While the approach introduced in [14] determines the structural similarity based on the SQL query, the implementation in Polypheny-DB determines the similarity based on the logical query plan.

The query routing is fully modular. New implementations can be provided on classpath without modifying Polypheny-DB. It is also possible to change the router implementation at runtime. It is the responsibility of the query router to make sure that the transition to a different implementation happens without conflicts. This feature is useful if there are maintenance tasks or data imports.

4.3 Data Storage

As a polystore system, Polypheny-DB supports different heterogeneous data stores. These data stores are fully maintained by the system. In order to be able to guarantee correctness, the stores are only accessed through Polypheny-DB.

The specific optimizations and advantages of a data store are exploited by pushing down the entire query or parts of it whenever possible. If supported by the data store, results of a query are only read on demand. Data stores can be added, modified and removed at runtime via the Polypheny-UI or by using SQL commands. The connection to the data stores is handled by means of *adapters*.

An adapter is responsible for everything related to a data store. This includes maintaining the connection and providing the implementations for the supported relational operators. The adapter is also responsible for maintaining the physical schema on the underlying data store and for performing schema migrations. The physical entity names itself are independent from the logical names. An adapter can support multiple database systems.

Polypheny-DB is shipped with adapters for various JDBC-SQL stores including PostgreSQL, MonetDB, and HSQLDB. Furthermore, there is an adapter for Apache Cassandra and for plain CSV files and Polypheny-DB can be extended with new adapters without any modifications due to its strong encapsulation. Some of the provided adapters support an embedded mode to simplify the deployment process.

5 Conclusion and Future Work

With Polypheny-DB, we present a modular polystore system which provides full support for DML and DDL queries. First performance evaluations have shown very promising results which we, due to the lack of space, have not been able to report here. It is planned to present them together with further evaluations we are currently doing and which we plan for our future work. Evaluations of a previous version of our DML-capable query routing system can be found in [14].

Polypheny-DB's modular architecture makes it extensible and adaptable. Several aspects of the system can be modified and exchanged at runtime. In future work, we plan to use this runtime adaptability to automatically adjust the system according to user-defined requirements and changes in the workload. We also plan to expand the set of integrated features. By extending the internal data model, we want to further exploit the advantages of different data models.

Acknowledgments. This work has been partly funded by the Swiss National Science Foundation (SNSF) in the context of the project Polypheny-DB (contract no. 200021_172763).

References

1. Begoli, E., Camacho-Rodríguez, J., Hyde, J., Mior, M.J., Lemire, D.: Apache calcite: a foundational framework for optimized query processing over heterogeneous data sources. In: Proceedings of the 2018 International Conference on Management of Data, pp. 221–230. ACM Press (2018). https://doi.org/10.1145/3183713.3190662
2. Bernstein, P., Hadzilacos, V., Goodman, N.: Concurrency Control and Recovery in Database Systems. Addison-Wesley Longman Publishing Co. Inc., Boston (1987)
3. Färber, F., Cha, S.K., Primsch, J., Bornhövd, C., Sigg, S., Lehner, W.: Sap HANA database: data management for modern business applications. SIGMOD Rec. **40**(4), 45–51 (2012). https://doi.org/10.1145/2094114.2094126
4. Gadepally, V., et al.: The BigDAWG polystore system and architecture. In: 2016 IEEE High Performance Extreme Computing Conference, HPEC 2016, Waltham, MA, USA, 13–15 September 2016, pp. 1–6. IEEE (2016). https://doi.org/10.1109/HPEC.2016.7761636
5. Giceva, J., Sadoghi, M.: Hybrid OLTP and OLAP, pp. 1–8. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-63962-8_179-1
6. Grund, M., Krüger, J., Plattner, H., Zeier, A., Cudre-Mauroux, P., Madden, S.: HYRISE: a main memory hybrid storage engine. Proc. VLDB Endow. **4**(2), 105–116 (2010). https://doi.org/10.14778/1921071.1921077
7. Hausenblas, M., Nadeau, J.: Apache drill: interactive ad-hoc analysis at scale. Big Data **1**, 100–104 (2013). https://doi.org/10.1089/big.2013.0011
8. Kemper, A., Neumann, T.: HyPer: a hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In: Abiteboul, S., Böhm, K., Koch, C., Tan, K. (eds.) Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, Hannover, Germany, 11–16 April 2011, pp. 195–206. IEEE Computer Society (2011). https://doi.org/10.1109/ICDE.2011.5767867
9. Kolev, B., Bondiombouy, C., Valduriez, P., Jimenez-Peris, R., Pau, R., Pereira, J.: The CloudMdsQL multistore system. In: Proceedings of the 2016 International Conference on Management of Data, SIGMOD 2016, pp. 2113–2116. Association for Computing Machinery, New York (2016). https://doi.org/10.1145/2882903.2899400
10. Kolev, B., Pau, R., Levchenko, O., Valduriez, P., Jimenez-Peris, R., Pereira, J.: Benchmarking polystores: the CloudMdsQL experience. In: 2016 IEEE International Conference on Big Data (Big Data), pp. 2574–2579. IEEE (2016). https://doi.org/10.1109/BigData.2016.7840899
11. Mattson, T., Gadepally, V., She, Z., Dziedzic, A., Parkhurst, J.: Demonstrating the BigDAWG polystore system for ocean metagenomic analysis. In: Proceedings of the 8th Biennial Conference on Innovative Data Systems Research (CIDR), p. 9. http://cidrdb.org/cidr2017/papers/p120-mattson-cidr17.pdf
12. Ramnarayan, J., et al.: SnappyData: a hybrid transactional analytical store built on spark. In: Proceedings of the 2016 International Conference on Management of Data, pp. 2153–2156. ACM Press (2016). https://doi.org/10.1145/2882903.2899408

13. Tan, R., Chirkova, R., Gadepally, V., Mattson, T.G.: Enabling query processing across heterogeneous data models: a survey. In: Nie, J., et al. (eds.) 2017 IEEE International Conference on Big Data, BigData 2017, Boston, MA, USA, 11–14 December 2017, pp. 3211–3220. IEEE Computer Society (2017). https://doi.org/10.1109/BigData.2017.8258302
14. Vogt, M., Stiemer, A., Schuldt, H.: Icarus: towards a multistore database system. In: Proceedings of the 2017 IEEE International Conference on Big Data (Big Data), pp. 2490–2499. IEEE (2017). https://doi.org/10.1109/BigData.2017.8258207
15. Vogt, M., Stiemer, A., Schuldt, H.: Polypheny-DB: towards a distributed and self-adaptive polystore. In: 2018 IEEE International Conference on Big Data (Big Data), pp. 3364–3373. IEEE (2018). https://doi.org/10.1109/BigData.2018.8622353