Polypheny Prism: Lessons Learned from Crafting a Versatile Multimodel, Multilingual Query Interface

Martin Vahlensieck[1,†], Tobias Hafner[1,†], Heiko Schuldt[1] and Marco Vogt[1]

[1]Databases and Information Systems Group, University of Basel, Switzerland

Abstract

Query interfaces are essential components of Database Management Systems (DBMS), enabling efficient data retrieval and manipulation. The diversification of DBMSs, driven by the increasing volume and complexity of data, has led to data silos with incompatible data models and query languages. Polystores and PolyDBMSs address this issue by providing a unified interface for accessing and integrating data across heterogeneous systems. This paper introduces Prism, a conceptual model and implementation of a multimodel, multilingual query interface protocol. Prism facilitates the use of multiple query languages and data models within a single protocol, simplifying data integration and application development.

Keywords

Query Interface, Multimodel, Polystore, Polypheny

1. Introduction

Query Interfaces are an essential part of Database Management Systems (DBMS). In combination with bindings for different programming languages and environments, known as drivers, Query Interfaces enable efficient retrieval and manipulation of data maintained by the DBMS.

The growing volume and complexity of data have given rise to diverse new types of database management systems, based on different data models and requiring different query languages [1]. Optimized for specific kinds of workloads, these DBMSs are well suited for certain type of applications. This has led to a fragmentation of the data storage landscape, creating silos of data incompatible in how they represent data and which query languages they support [2].

Polystores [3], and in particular PolyDBMS [4], try to counteract this fragmentation by enabling access to data distributed across heterogeneous DBMSs through a single interface. This allows data from different silos to be combined (e.g., joined) in one query. Furthermore, these systems drastically simplify application development since all the data can be accessed using one query language and connection.

However, certain data models are better suited for specific applications and environments than others. Therefore, since each data model brings its own query language(s), these languages

LWDA'24: Lernen, Wissen, Daten, Analysen. September 23–25, 2024, Würzburg, Germany

martin.vahlensieck@unibas.ch (M. Vahlensieck); t.hafner@stud.unibas.ch (T. Hafner); heiko.schuldt@unibas.ch (H. Schuldt); marco.vogt@unibas.ch (M. Vogt)

0000-0001-9865-6371 (H. Schuldt); 0000-0002-2674-2219 (M. Vogt)

[†]These authors contributed equally.

are better suited for these applications [2]. PolyDBMSs therefore support multiple data models and query languages.

Systems like the PolyDBMS Polypheny [5, 6] require a query interface that supports both multiple data models and multiple query languages. Yet, to the best of our knowledge, there is no conceptual model, reference implementation or framework for such a query interface.

The contribution of this paper is to introduce the conceptual model and describe the implementation of a Query Interface Protocol for multimodel, multilingual systems, named *Prism*. The name has been chosen since it, like a prism that refracts light into multiple colors, supports multiple query languages and data models in one single protocol.

The remainder of this paper is structured as follows: Section 2 outlines the requirements for such a query interface protocol and discusses related approaches. In Section 3 we introduce the conceptual model of the protocol, the implementation of which is then introduced in Section 4. The results of the benchmarking are introduced in Section 5. Section 6 concludes the paper and gives an outlook on future work.

2. State of the Art

To fully take advantage of the multimodel, multilanguage capabilities of systems such as Polypheny, we need a sophisticated query interface that allows queries to be submitted using various query languages. Queries should not be mistaken as simple strings, as features like prepared statements necessitate proper handling of parameters. Moreover, the interface must support batching of DML operations to enhance efficiency and performance. Additionally, the query interface and the underlying protocol must be capable of handling multiple data models.

Implementing a single query interface and protocol, instead of multiple interfaces, is crucial for several reasons: First, transaction management is a fundamental aspect of the query interface, deeply integrated with standards like JDBC. A unified interface ensures effective and seamless transaction control across various languages and data models, which would be complex and error-prone with multiple interfaces (cf. polyglot persistence). Additionally, it ensures consistent behavior and compatibility across different query languages and data models, simplifying development and maintenance.

Current industry standards like JDBC [7] have set the precedent for relational database interactions. JDBC provides a well-defined and widely adopted API for Java applications to interact with relational databases. It includes support for transaction management, prepared statements, and batched operations. Ensuring compatibility with such standards is critical, as it allows existing applications to interface seamlessly with new systems without extensive modifications. The protocol therefore needs to support building drivers compatible with these industry standard APIs.

However, beyond relational databases, there are no clearly defined standards for interacting with non-relational databases similar to JDBC or Python's DB-API. This absence highlights the necessity of developing a versatile query interface and protocol that can unify interactions across different data models and query languages.

ArangoDB [8] is a multimodel DBMS, supporting document, graph, and key-value data models. It uses its proprietary ArangoDB Query Language (AQL), designed for both document

and graph data models. Integration options include a RESTful API, the Arango Shell, a Web UI, and drivers for various programming languages such as Python, Java, and Go. These drivers offer methods for operations like data insertion, database creation, and edge addition, using objects with getters and setters for serialized data, which is serialized using VelocyPack, ArangoDB's proprietary binary JSON format. Despite that, ArangoDB does not provide a multimodel multilanguage query interface.

OrientDB [9], another multimodel DBMS, supports the document, graph, object-oriented, and key-value data models. It employs an SQL-like query language and Gremlin for graph queries. OrientDB provides drivers for various programming languages that communicate via a proprietary binary protocol. Similar to ArangoDB, OrientDB's approach requires users to adapt to proprietary query languages, complicating integration and usage across different data models. This design necessitates significant effort to maintain and extend the proprietary languages, reducing overall flexibility and efficiency. Consequently, OrientDB also lacks a true multimodel multilanguage query interface.

Apache Calcite Avatica [10] is a framework for developing drivers compatible with the JDBC standard. It serves as an abstraction layer for client-server communication for SQL-based relational systems. Avatica provides common structures for JDBC drivers, eliminating the need to develop these components from scratch. Avatica consists of a universal JDBC client, which is database-independent, utilizing Avatica's own communication mechanisms and protocols. The driver is adapted to the respective database system by implementing interface methods on the server side. These methods are called from the server-side Avatica communication endpoint. Parameters and return values use Avatica-specific representations of internal database concepts. The implemented interface contains methods to map from Avatica structures to internal database concepts and vice versa [10]. However, Avatica is primarily designed for relational systems and does not natively support the integration of non-relational data models. This limitation is evident in Polypheny's current query interface, which, being based on Avatica, is restricted to relational queries.

In summary, while systems like ArangoDB and OrientDB offer support for multiple data models, their reliance on a single proprietary query language introduces complexity and reduces flexibility. On the other hand, frameworks like Apache Calcite Avatica provide robust solutions for relational databases but lack native support for multimodel environments. This highlights the pressing need for a versatile query interface and protocol that can seamlessly integrate multiple data models and query languages, ensuring compatibility with existing standards and simplifying the development and maintenance process.

3. Conceptual Model

The proposed conceptual model defines a query interface for a DBMS that supports multiple data models and query languages. The protocol is based on *sessions* which the client (e.g., an application) establishes with the server (the DBMS). Within these sessions, the client manages transactions and executes queries using requests to which the server returns responses. Queries return data in so-called *result sets*, which can be of different types. The data values both in the results and the requests are serialized as *Prism Values*.

Table 1
Basic request and response types.

Request type	Response type
ConnectionRequest	ConnectionResponse
PrepareStatement	Statement
ExecuteStatement, Fetch	Frame
Commit, Rollback, Close, CloseResultSet, CloseStatement	Success

3.1. Sessions

A *session* is an abstraction for the connection between the client and the server. It is not to be confused with a *transaction*. An arbitrary number of transactions can take place within a single session; however, there may only be one transaction at a time. More details on the protocols handling of transactions are provided in Section 3.5.

After establishing a session, the client sends requests. The server responds to each request with one or more responses. Each request has an id that is unique within the session. Each response includes the id of the request and a flag whether it is the last response for that request. The session ensures multiple responses are delivered in the order they are returned. We write request and response types with small capitals, so a request of type commit would be Commit.

Sessions can be closed with a Close request or externally (e.g., by a timeout in the underlying transport protocol).

3.2. Request and Response Types

Each request type has a fixed response type. We list basic request types and their responses in Table 1. The implementation may define additional request and response types.

An Error response is used to report failures. This response type is special because it can be returned to any request and is always the last response for that request. Errors do not close sessions. The error response may contain relevant diagnostic information.

Some requests listed in Table 1 only define a Success response. These requests can fail, but in those cases the Error response is returned.

The very first request in a session is always of type ConnectionRequest. It contains credentials used for authentication and the client version. The server returns a ConnectionResponse with the server version number and a flag that is set when both versions are compatible.

3.3. Statement Handling

To execute queries, so-called *statements* are used. A statement is created or *prepared* by sending the query to the server. The server then compiles the query and returns a *statement handle*, that can be used to refer to that query. The statement handle can be then used to *execute* the query once or multiple times. When executed, the statement produces zero or more result tuples. Multiple result tuples are then chunked together into *result sets* which are sent to the client.

Queries can also have placeholders. Placeholders are filled with values when executing the statement. Placeholders can be of two types: Positional and named. Each positional placeholder

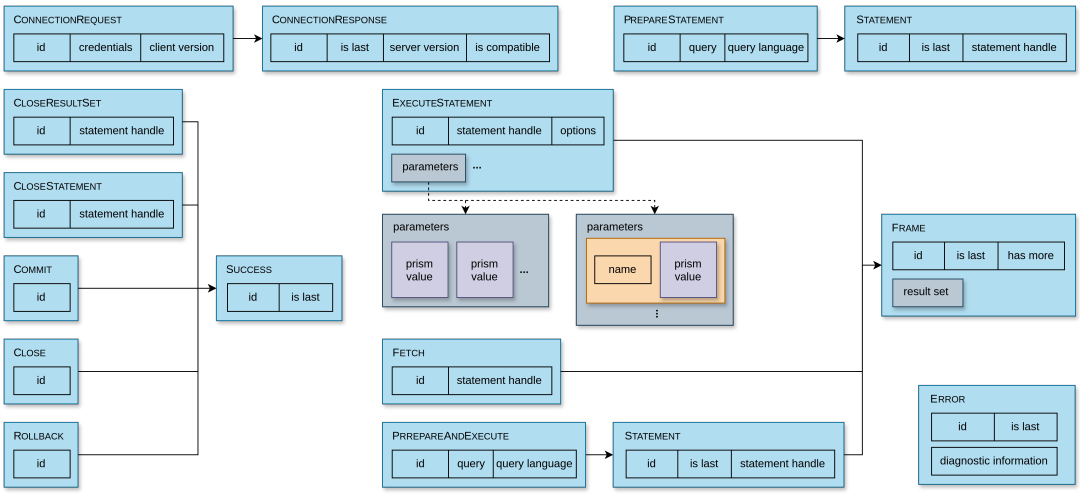

Figure 1: Overview of the requests and the corresponding responses.

has an index, which matches the position it appears in the query. Named parameters get their names from the specification in the query. Which parameter types (if any) are supported, and how parameters are specified, is defined by the query language. Positional and named parameters may not be mixed within a query.

The client prepares a query using the PrepareStatement request, which contains the query and a string indicating the query language. The server compiles the query and returns a Statement response. This response contains a statement handle, which can be used to refer to the statement within the same session until it is closed.

The prepared statement is then executed by sending an ExecuteStatement request with three fields: statement handle, options and parameters. Options are used to send information related to the query, such as a priority or freshness requirement. Parameters is used to specify values for the placeholders present in the query. The format of parameters depends on the placeholders used: For positional placeholders, parameters is an array with the same number of elements as number of placeholders. The placeholder with index i is then filled by the array element at index i. For named placeholder, parameters specify for each name the value with which it should be filled. ExecuteStatement returns a Frame containing the first result set and a flag if more result sets are available. Further Frames with result sets are fetched with Fetch requests that contain the statement handle.

ExecuteStatement can also take a batch of parameters. The query is then executed once for each item in the batch.

For queries with no placeholders, there is also a PrepareAndExecute request that prepares the query and then immediately executes it. This request returns two StatementResponses. The first contains the statement handle and is sent after compiling the query, the second contains the first result set after execution. Subsequent result sets are fetched like for ExecuteStatement.

There are two request types to free resources, which contain only the statement handle. The first type, CloseResultSet is used by the client to indicate that the execution can be stopped

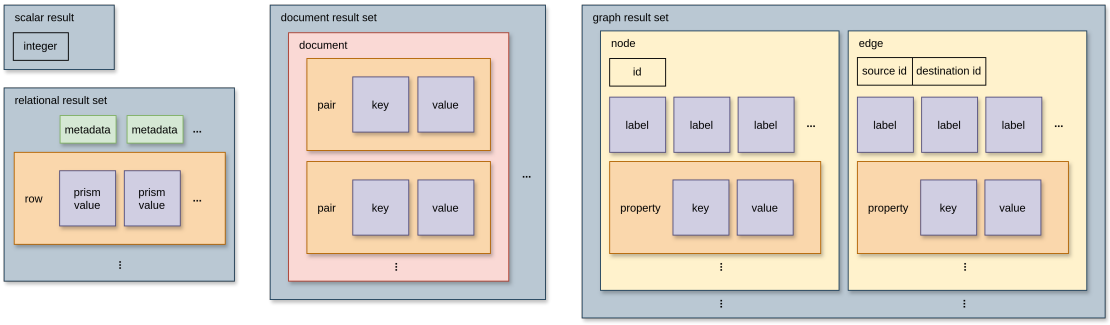

Figure 2: Overview of the possible result sets.

and no more result sets will be fetched (it can be seen as the counterpart to ExecuteStatement). The second type CloseStatement closes the statement and the server can free the associated resources. The statement handle must no longer be used. It can be seen as the counterpart to the PrepareStatement request. Closing a session also closes all statements and result sets.

3.4. Result Set Types

We consider four types of result sets: scalar, relational, document and labeled property graph. Scalar results are simply an integer number. Scalar results are common for DML queries, e.g., the number of affected rows in an update.

Relational result sets consist of two parts: *metadata* and *rows*. Let n be the number of columns in the relational result and m the number of tuples returned. The metadata consists of elements $c_1, ..., c_n$ where c_i describes column i. This metadata is relational information such as column name, type, nullability etc. Rows is a list of m tuples $r_1, ..., r_m$. Each tuple consists of $v_1, ..., v_n$ prism values where v_i is the value in column i.

Document result sets are a list of zero or more documents. Each document consists of an arbitrary number of key-value pairs. Values may contain again documents, i.e., they can be nested.

Graph result sets consist of nodes and edges. Nodes and edges can have labels and properties. Labels are strings like "PERSON" or "KNOWS". Properties are key-value pairs. Values cannot contain properties, i.e., they cannot be nested. A node is represented as a tuple of the form (*id*, *labels*, *properties*). *id* is a unique value identifying the node. Edges are tuples of the form (*source_id*, *destination_id*, *labels*, *properties*). Both *source_id* and *destination_id* refer to the id value of nodes.

Against intuition, the result type model is not solely defined by the query language, but also by the query itself. Examples include SQL and Cypher: A DML SQL query will produce a scalar result, while a DQL SQL query produces a relational result. Cypher queries can produce scalar, relational or graph results. It is therefore advisable that the client is prepared to handle results of all types, independent of which language is selected.

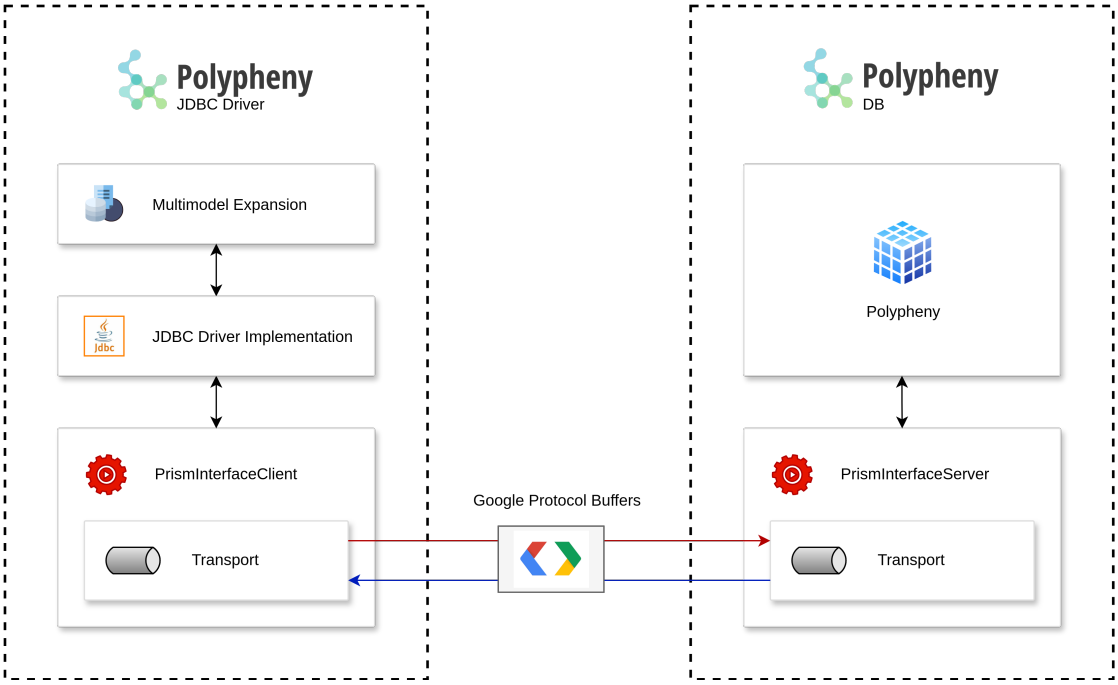

Figure 3: Basic building blocks of the implementation of Prism in Polypheny and our JDBC driver featuring the multimodel extensions.

3.5. Transaction Handling

Transactions are opened implicitly by preparing statements. Transactions can be committed with a COMMIT request and rolled back with a ROLLBACK request. When the session is closed (either by a CLOSE request or externally) the current transaction is rolled back. However, transactions are not only terminated by the client. The DBMS may commit some transactions automatically when certain queries are executed (such as DDLs) or rollback transactions due to deadlocks or other issues. The client must therefore be able to deal with changes to the state of the transaction.

3.6. Prism Values

Prism Values consists of two classes: Primitive values and complex values. Primitive values are integers, floating-point numbers, strings, bytes, day, time, timestamp, time interval and null. Complex values consist of other prism values. There are two complex values: lists and documents. Lists are a sequence of zero or more Prism values. Documents are key-value pairs, where both key and value can be Prism values.

4. Implementation

We have implemented the protocol introduced in Section 3 in the PolyDBMS Polypheny. Our implementation includes the query interface (server) in Polypheny and various drivers for different programming languages. In this section, we give an overview of the parts related to client-server communication, including serialization and message transport. Furthermore, we provide some insights and discuss some lessons learned from implementing drivers for our query interface in Java and Python.

4.1. Client-Server Communication

To communicate between client and server, we need to serialize messages and their content and transmit them over a transport protocol.

For serialization, we chose Google Protocol Buffers (Protobuf) [11]. Protobuf is based on definition files that are compiled into language bindings. Definition files contain the specification of so-called messages, which are (de-)serialized to native objects using the language bindings. Messages can contain other messages, primitive types (such as int or bytes), lists and unions.

We chose Protobuf because it has bindings to many (10+) programming languages and because the binary serialization aims to be backwards compatible. The protobuf definition files of our implementation of the Prism interface are available on GitHub[1].

Initially, we used gRPC, a remote procedure call (RPC) library developed by Google, where the client sends RPCs to the server. However, gRPC is call-based rather than connection-based, meaning each client must explicitly close the session, or it will remain open indefinitely. To make sure each session is closed, each client would have to maintain a persistent state (in case it crashes) that is used in a subsequent invocation to close any open sessions. The other option is to aggressively use timeouts, which comes with other problems.

To solve this, we decided to (de-)multiplex messages ourselves and send them over so-called *transports*. A transport handles the details of how to send messages over another transport protocol. We have two transports: one based on TCP and one based on UNIX domain sockets. Transports propagate close events to the session (e.g., a connection reset in TCP). The session is then considered externally closed and cleans up all resources. This works very well in practice, the OS kernel closes associated TCP connections when an application exits.

With transports, cleanup of sessions is easier, and it is straightforward to support other transport protocols. With our initial gRPC-based implementation, we have also observed fluctuations in the round-trip time of RPCs. With our own implementation, these are no longer present.

4.2. Driver Implementation

We have implemented drivers for Java, Python, Go and C++. Three of these languages have a standard on database drivers: *JDBC* for Java, *DB API* for Python and *db/sql* for Go. In this section, we will focus on our implementations for Java and Python, since they some represent the two extremes. The JDBC standard [7] is massive compared to the rather minimal DB API

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

standard. Furthermore, Java is a compiled, statically and strictly typed language, while Python is interpreted and dynamically typed.

When implementing a driver for a programming language or environment, we want to, when present, implement the standard as closely as possible, while still offering the full capabilities of our interface without compromise. Since existing standards only cover relation database interactions, this means finding a proper approach for supporting additional data models while preserving compatibility.

Both JDBC and the Python DB API were designed for relational SQL DBMSs. Both standards define classes that need to implement certain methods. We have added additional methods for multimodel, multilanguage queries to those classes. In Python, these methods can be called directly (because it is dynamically typed). In Java, the client has to first cast the generic JDBC Connection object to our implemented subclass, which can be done with a method defined by the JDBC standard.

Another challenge is how to map Prism values to the type system of the programming language. A Prism value may not have a native counterpart in a certain programming language. When no native counterpart exists, we declare a wrapper object ourselves. One example is the interval type, which consists of two integers: Number of months and number of milliseconds (because a month cannot be clearly defined in terms of milliseconds). In Python, the values are converted directly. In Java, they are converted to subclasses of an abstract `TypedValue` object. This object defines methods to retrieve the internal value as a Java type.

An additional challenge with implementing compatibility with the JDBC standard were the calls to retrieve metadata about the database. The JDBC standard defines many methods to retrieve metadata about database entities, such as tables or functions. To retrieve the information for each of the methods specified in the standard, we have added specific request types to the protocol to retrieve that information.

In the future, we would like to implement a more holistic approach for retrieving metadata. Instead of extending the Prism protocol with specific metadata requests to accommodate every standard, we plan to take advantage of our versatile interface and design a new query language with the sole purpose of metadata retrieval. This query language can then be used by all drivers and can return metadata that is relevant to our system. Since the Prism protocol does not require a fixed mapping between query language and result type, this metadata retrieval language will be able to specify metadata using any of the available data models. Compared to general metadata requests that may require sophisticated processing of the metadata within the driver to fulfill the specifications of the standard (e.g., regarding the order of the information), a metadata retrieval language keeps the driver lightweight since the processing is done on the server.

5. Evaluation

We have benchmarked the performance of our implementation of the Prism query protocol in the Polypheny system. For this, we have compared the performance of the new Prism-based implementation with both the existing implementation based on the Avatica framework and a simple JSON endpoint. We compiled a list of database tasks as workload and executed it using the different interfaces.

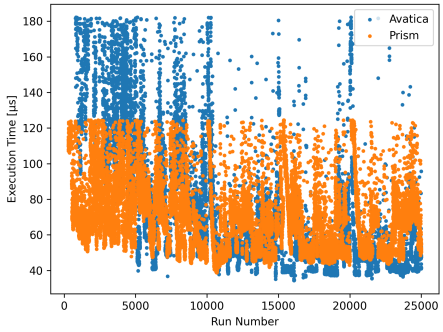

Figure 4: Execution times without outlier across the 25'000 repetitions of the benchmark.

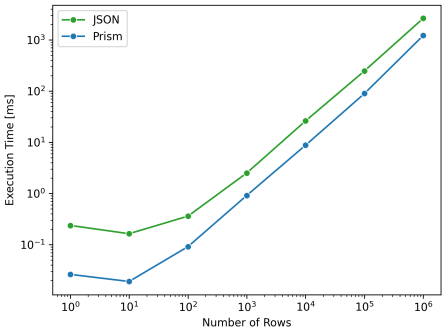

Figure 5: Times required to serialize relational results of various sizes

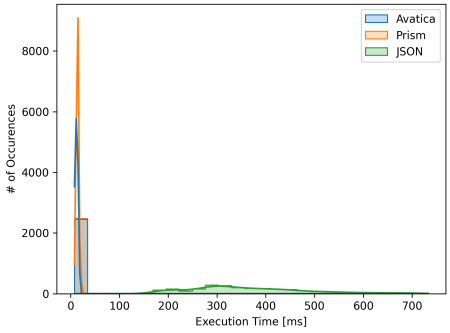

Figure 6: Histogram of the execution times required to insert 1'000 rows into a relation.

For the benchmarks, both Polypheny and the test client were run on the same computer, equipped with an Intel Core i7-1280P processor and 32GB of RAM. The operating system used was Ubuntu 20.04.6 LTS, and the Java version deployed was OpenJDK 17.0.8. Each task has been executed 25'000 times, split into five runs of 5'000 each, using an automated test client.

We made three comparisons: First, we compared the round-trip time of an empty RPC call between Avatica and Prism to determine the pure overhead (5.1). We then compare the time to serialize relational result sets between the JSON endpoint and Prism (5.2). At the end, we compare how long it takes to insert 1'000 rows using each of the three interfaces (5.3).

5.1. RPC Overhead

The first aspect of the Prism implementation that we have benchmarked is the round-trip time of a single RPC (Remote Procedure Call). This metric is crucial since any significant overhead

in the RPC system directly impacts the overall communication and thus the performance. Both the Avatica-based and Prism-based interfaces rely on custom RPC implementations rather than existing open source RPC frameworks. For this benchmark, we measured the time taken for an RPC call without a payload across 25'000 calls. Figure 4 depicts the measured time for each test run. One notable observation is the initially high execution time of the Avatica-based implementation, which only decreases after 10,000 calls, while still displaying significant inconsistencies, with a standard deviation of 31.4404μs. The Prism interface in contrast is more consistent (standard deviation of 18.1609μs) with short execution times from the start of the benchmark. Despite the inconsistencies, execution times are relatively similar, with a mean of 67.0194μs for the Prism interface and 65.6982μs for the Avatica-based implementation. This benchmark therefore shows that there are no relevant differences in the execution times caused by the RPC stacks that might skew the validity of the following measurements.

5.2. Value Serialization

Another important parameter is the time required to serialize values, impacting both parameterized statements and query results. To benchmark this, we compared the serialization mechanism of the Prism interface with a naive JSON-based implementation. We generated relational results of exponentially increasing sizes, ranging from 1 row to 1'000'000 rows, and measured the time required to serialize the responses. The results are depicted in Figure 5. Our results indicate that the Prism serialization is consistently faster than the JSON one. The largest difference is in the range of 1 row to 100 rows, commonly used as fetch sizes within applications applying pagination to the data. The plateauing of execution times noticed for small results in both methods is due to the time required to initialize the serialization frameworks. This overhead is constant in time complexity and thus becomes negligible with larger results. Despite a warm-up period being applied to reduce the impact of the JIT compiler, there still is a decrease for the 10 rows test due to the JIT compiler optimizing during the 1 row test, as all tests were executed sequentially.

5.3. Insert Performance

To highlight the importance of a stateful protocol like the Prism interface or Avatica in contrast to a state-less approach (for instance a simple JSON endpoint), we have evaluated a scenario involving the insertion of 1'000 rows using prepared statements. For the Prism and the Avatica interface, we leveraged their capability of handling prepared statements by first preparing an insert statement, which was then batch-executed with the values for the 1'000 rows. In contrast, with the stateless JSON endpoint, each row needs to be inserted using its own individual statement. The results, depicted in Figure 6, show that a stateful protocol is not only necessary to provide features like transaction control, but also significantly improves performance due to the availability of features like batching. This demonstrates the necessity of a well-defined protocol for query interfaces.

5.4. Discussion

Our evaluation demonstrates that the Prism query protocol and its implementation in the Polypheny system are performance-wise comparable to the Avatica framework, showing no significant differences in execution times. Furthermore, we demonstrated the advantages of a typed and efficient serialization protocol in contrasted to a naive JSON serialization. Such a naive stateless approach, which is unfortunately quite common in NoSQL systems due to the lack of well-defined query interface protocols and standards, is not only performance-wise inferior due to the lack of features like batching, it is also unable to provide support for essential features of a DBMS like transaction control.

6. Conclusion and Future Work

With Prism, we introduced a conceptual model for a multimodel, multilingual query interface protocol, which we have successfully implemented within the Polypheny system. Benchmarking results indicate that Prism delivers performance comparable to the established Avatica framework, with the added advantage of supporting multiple data models and query languages. This makes Prism not only a valid implementation for Polypheny, but also a conceptual model and reference for other multimodel and multilanguage database systems.

Currently, we are developing drivers for various programming languages and environments. Notably, there is a Google Summer of Code project dedicated to implementing a driver for .NET[2].

Future work includes several key areas of development. First, we aim to implement robust encryption mechanisms to ensure data security and privacy during transmission. Additionally, we plan to expand the conceptual model and add support for more data models such as time-series, thereby enabling a wider range of applications and use cases. Another area of focus is adding the ability for random reads in Binary Large Objects (BLOBs). Finally, we envision developing transport abstraction layers, which could lead to innovative use cases like sending Protocol Buffers over web sockets for real-time user interfaces.

Acknowledgments

This work is partly funded by the Swiss National Science Foundation, project Polypheny-DDI (contract no. 200020_213121 / 1), and Innosuisse, project SwissRenov (contract no. 107.512 FS-EE).

References

[1] M. Stonebraker, U. Çetintemel, “One size fits all”: An idea whose time has come and gone, in: Proceedings of the 21st International Conference on Data Engineering, IEEE, 2005, pp. 2–11. doi:10/ctkd2n.

[2] https://summerofcode.withgoogle.com/programs/2024/projects/Z5ZX5ekd

[2] Q. Guo, C. Zhang, S. Zhang, J. Lu, Multi-model query languages: taming the variety of big data, Distributed and Parallel Databases 42 (2024-03) 31–71. URL: https://link.springer.com/10.1007/s10619-023-07433-1. doi:10/gt38bq.
[3] R. Tan, R. Chirkova, V. Gadepally, T. G. Mattson, Enabling query processing across heterogeneous data models: A survey, in: Proceedings of the 2017 IEEE International Conference on Big Data, IEEE, 2017, pp. 3211–3220. doi:10/gnr5rf.
[4] M. Vogt, D. Lengweiler, I. Geissmann, N. Hansen, M. Hennemann, C. Mendelin, S. Philipp, H. Schuldt, Polystore systems and DBMSs: Love marriage or marriage of convenience?, in: E. K. Rezig, V. Gadepally, T. Mattson, M. Stonebraker, T. Kraska, F. Wang, G. Luo, J. Kong, A. Dubovitskaya (Eds.), Heterogeneous Data Management, Polystores, and Analytics for Healthcare – VLDB Workshops, Poly 2021 and DMAH 2021, volume 12921 of *Lecture Notes in Computer Science*, Springer International Publishing, 2021, pp. 65–69. doi:10/gn8qvm.
[5] M. Vogt, N. Hansen, J. Schönholz, D. Lengweiler, I. Geissmann, S. Philipp, A. Stiemer, H. Schuldt, Polypheny-DB: Towards bridging the gap between polystores and HTAP systems, in: V. Gadepally, T. Mattson, M. Stonebraker, T. Kraska, F. Wang, G. Luo, J. Kong, A. Dubovitskaya (Eds.), Heterogeneous Data Management, Polystores, and Analytics for Healthcare – VLDB Workshops, Poly 2020 and DMAH 2020, Lecture Notes in Computer Science, Springer International Publishing, 2020, pp. 25–36. doi:10/gnxv2h.
[6] M. Vogt, Adaptive Management of Multimodel Data and Heterogeneous Workloads, Ph.D. thesis, University of Basel, 2022. doi:10/j44k.
[7] L. Andersen, JDBC 4.3 specification, JSR 221, 2017.
[8] ArangoDB, What is arangodb?, 2024. URL: https://www.arangodb.com/docs/stable/index.html.
[9] D. Ritter, L. Dell'Aquila, A. Lomakin, E. Tagliaferri, OrientDB: A NoSQL, open source MMDMS, in: Proceedings of the British International Conference on Databases 2021, London, United Kingdom, March 28, 2022, volume 3163 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2021, pp. 10–19.
[10] Apache Software Foundation, Avatica, 2023. URL: https://calcite.apache.org/avatica/docs/.
[11] C. Currier, Protocol buffers, in: C. Hummert, D. Pawlaszczyk (Eds.), Mobile Forensics – The File Format Handbook, Springer International Publishing, 2022, pp. 223–260. doi:10/m7rq.