DortDB: Bridging Query Languages for Multi-Model Data Ponds

Filip Ježek
Department of Software Engineering
Charles University
Prague, Czech Republic
filip.jezek@gmail.com

Pavel Koupil
Department of Software Engineering
Charles University
Prague, Czech Republic
pavel.koupil@matfyz.cuni.cz

Michal Kopecký
Department of Software Engineering
Charles University
Prague, Czech Republic
michal.kopecky@matfyz.cuni.cz

Jáchym Bártík
Department of Software Engineering
Charles University
Prague, Czech Republic
jachym.bartik@matfyz.cuni.cz

Irena Holubová
Department of Software Engineering
Charles University
Prague, Czech Republic
irena.holubova@matfyz.cuni.cz

ABSTRACT

Multi-model data encompasses structurally distinct data, including relational, document, graph, key/value, columnar, etc., managed within a single system, such as a multi-model database or a data lake. Querying multi-model data requires strategies that balance unification and integration across diverse models and query languages. This paper presents DortDB, an extensible framework enabling cross-model queries combining well-known query languages and offering intuitive flexibility and optimization via a unified algebra. Though a small-scale in-memory prototype is to be demonstrated, its principles can be extended to distributed systems.

PVLDB Reference Format:
Filip Ježek, Pavel Koupil, Michal Kopecký, Jáchym Bártík, and Irena Holubová. DortDB: Bridging Query Languages for Multi-Model Data Ponds. PVLDB, 18(12): 5335-5338, 2025.
doi:10.14778/3750601.3750665

PVLDB Artifact Availability:
Available at https://github.com/filipjezek/dortdb.

1 INTRODUCTION

Multi-model data refers to different types of data – relational, document, graph, key/value, columnar, etc. – being managed as a whole. They can be stored in a multi-model database management system (DBMS) [9] designed to handle diverse data formats while ensuring optimized performance and querying capabilities. An alternative approach represents data lakes [1], which store raw data in its native format without requiring a predefined schema.

Choosing between multi-model DBMSs and data lakes depends primarily on particular business needs – structured querying and performance favour DBMSs, while large-scale, unstructured data analysis benefits from data lakes. But, data lakes, of course, also offer various querying and data extraction approaches, including SQL-based query engines (e.g., Apache Presto[1]), data lakehouses utilising the analytical queries of a data warehouse (e.g., Delta Lake[2]), or search engines (e.g., Elasticsearch[3]). Some approaches (e.g., AWS Athena[4]) enable federated queries over data lakes and relational or NoSQL DBMSs. However, in all these cases, we must consider the multi-model aspect of data, i.e., integration of data with highly contradictory features and thus data-access methods, such as e.g. flat relational data vs. hierarchical JSON documents or table joins vs. graph traversals.

A common strategy for multi-model querying, often employed in polystores [11], follows the principle of *unification at front-end and integration at back-end*. This approach presents a unified view of multi-model data while maintaining mappings to the original structures. Users then interact with this unified view using a single-model-specific language. The queries are translated into the underlying models' query languages, whereas an integration module combines the intermediate results into the final result.

Conversely, multi-model DBMSs [4] typically extend the query language of the original model with constructs supporting additional models (e.g., PostgreSQL's SQL extension for accessing JSON documents). This can be seen as an approximation of the inverse strategy, i.e., *integration at front-end and unification at back-end*. Several approaches exist that enable the combination of two query languages (e.g., SQL/XML or SPARQL embedding in SQL in Virtuoso). However, this approach could be generalized to allow users to flexibly combine multiple query languages, each suitable for a particular data model, with recursive embedding as needed.

Example 1.1. Consider the example in Fig. 1a, which illustrates data integration from heterogeneous sources. Customer information is obtained from a Neo4j database representing a social network, address data is stored locally in CSV format as relational tables, and order data is retrieved from a remote web service in XML.

A sample query might "*retrieve the most frequently ordered products that appear first in the orders of customers' friends, limited to friends residing in Prague and products ordered by more than two such friends*". The result is returned as a relational table in CSV format to support, e.g., personalized recommendations and targeted advertising. Intuitively, the query requires integration of Cypher for graph traversal, SQL for relational aggregation, and XQuery for filtering hierarchical data – see Fig. 1b. □

Proceedings of the VLDB Endowment, Vol. 18, No. 12 ISSN 2150-8097.
doi:10.14778/3750601.3750665

[1] https://prestodb.io/

[2] https://delta.io/

[3] https://www.elastic.co/elasticsearch

[4] https://aws.amazon.com/athena/

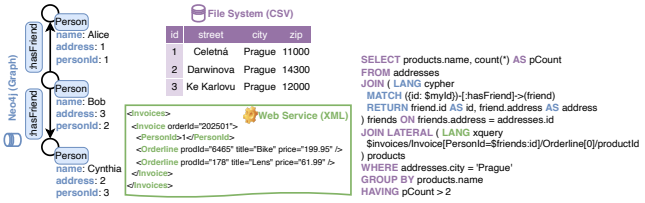

Figure 1: Sample multi-model data (a), cross-model query (b)

This paper introduces DortDB[5], an extensible framework that enables querying multi-model data using virtually any combination of query languages. Its main advantages are as follows:

- Users can seamlessly combine various data formats without the need to predefine schemas, providing unmatched flexibility in data integration.
- Users can intuitively switch between query languages based on their preferences or the specific data fragments they are working with, ensuring ease of use.
- By selecting the top-level language, the user specifies the output data format. He/she can thus transform the given data to any supported format (or a combination of formats).
- Queries are translated into a unified algebra, allowing cross-model optimization to enhance query performance.

Currently, DortDB is a small-scale prototype designed for use within a web browser as a multi-model in-memory *data pond*. However, the underlying concept can be extended to large-scale environments, such as distributed systems utilizing shared RAM.

2 BASIC CONCEPTS

The primary objective of DortDB is to offer a flexible and optimized query execution mechanism for in-memory multi-model data. Data can be imported from various sources, including web services, DBMSs, or local/remote file systems. Once imported, data is represented as an object model, which can be serialized into JSON. DortDB adopts a schema-free approach, enabling dynamic and flexible data management without the constraints of predefined schemas. Users can define secondary indexes in advance using expression-based specifications to optimise query performance.

DortDB queries are constructed using multiple query languages, currently SQL, XQuery, and Cypher (see Fig. 1b). These languages can be repeatedly nested within one another, allowing for the creation of complex and sophisticated queries. Once a query is formulated, the system parses it to generate a *syntactic tree* that captures the structure of the query. It is then transformed into a *logical plan*, represented as a directed graph. Its nodes correspond to algebraic operators (see Section 2.1). Its edges define the execution order and dependencies between operations, with a bottom-up orientation indicating data flow from sources to the final query result.

The logical plan undergoes optimization, such as operator reordering or index utilization. Then, it is executed over the object data representation following the structure defined by the logical tree and ensuring that operations are performed in the correct sequence. Data streams are processed in memory, benefiting from client-side caching to reduce redundant data retrieval from storage. The query output consists of a list of objects (e.g., a list of dictionaries or simple values). The result remains in object representation within memory. DortDB does not enforce a specific output format but allows users to transform the results into desired formats.

Example 2.1. A sample workflow of DortDB is depicted in Fig. 2. The user first imports data into DortDB from various sources (e.g., a web service providing invoice data, a local Neo4j database with person and friendship data, and a local file containing address data – see Fig. 1a). Heterogeneous data formats are normalized into a unified TypeScript object representation, where data types are denoted by letters (`O` = object, `S` = string, `N` = number, `E` = array element). The user then submits a query, which is parsed and transformed into the unified algebraic form, visualized as a simplified tree (consisting of operators and data sources), and subsequently optimized. The query is executed over the object-based data, and the result is returned. The result can be serialized into a desired format (e.g., CSV). □

2.1 Unified Algebra

The *Unified Algebra* (see Table 1) defines a formal and extensible framework for query processing, supporting both structured (e.g., relational) and semi-structured (e.g., graph or hierarchical) data. It integrates relational and XQuery algebras [10], path algebras for graphs [2], and optimizations for nested queries [6].

Table 1: Operators of the Unified Algebra

Class	Name	Signature
Tuple	**distinct**	δ(atts, src)
	groupBy	γ(keys, aggs, src)
	cartesianProduct	$\times$(left, right)
	join	$\bowtie$ (left, right, leftOuter, rightOuter, condition)
	orderBy	τ(orders, src)
	projection	π(attrs, src)
	projectionConcat	$\overset{\bowtie}{\rightarrow}$ (mapping, outer, src)
	projectionIndex	index(name, src)
	selection	σ(expr, src)
	recursion	ϕ(min, max, condition, src)
	tupleSource	**name**
	tupleFnSource	**name**(params)
	mapFromItem	fromItem(key, src)
Item	**calculation**	calc(args)
	conditional	cond(expr, whenthens, default)
	fnCall	fn(args)
	literal	literal(value)
	quantifier	quant(type, query)
	itemSource	name(params)
	tupleFnSource	name(params)
	mapToItem	toItem(key, source)
Ambivalent	**union**	$\cup$(left, right)
	intersection	$\cap$(left, right)
	difference	\(left, right)
	limit	limit(skip, limit)
	nullSource	[ ]
XQuery	**projectionSize**	size(name, source)
	treeJoin	treeJoin(expr, source)

Tuple operators manipulate *streams of named tuples*, which are analogous to, e.g., relational tuples. Core operators include *selection* (σ), *projection* (π), *join* ($\bowtie$), *cartesianProduct* ($\times$), *groupBy* (γ), *orderBy* (τ), and *recursion* (ϕ). Additional operators, such as *projectionConcat* and *projectionIndex*, facilitate advanced tuple transformations. Furthermore, *tupleSource* and *tupleFnSource* generate streams of tuples dynamically. *Item operators* process values as

[5]Inspired by Josef Čapek's fairy tale "How Doggie and Kitty were making a cake".

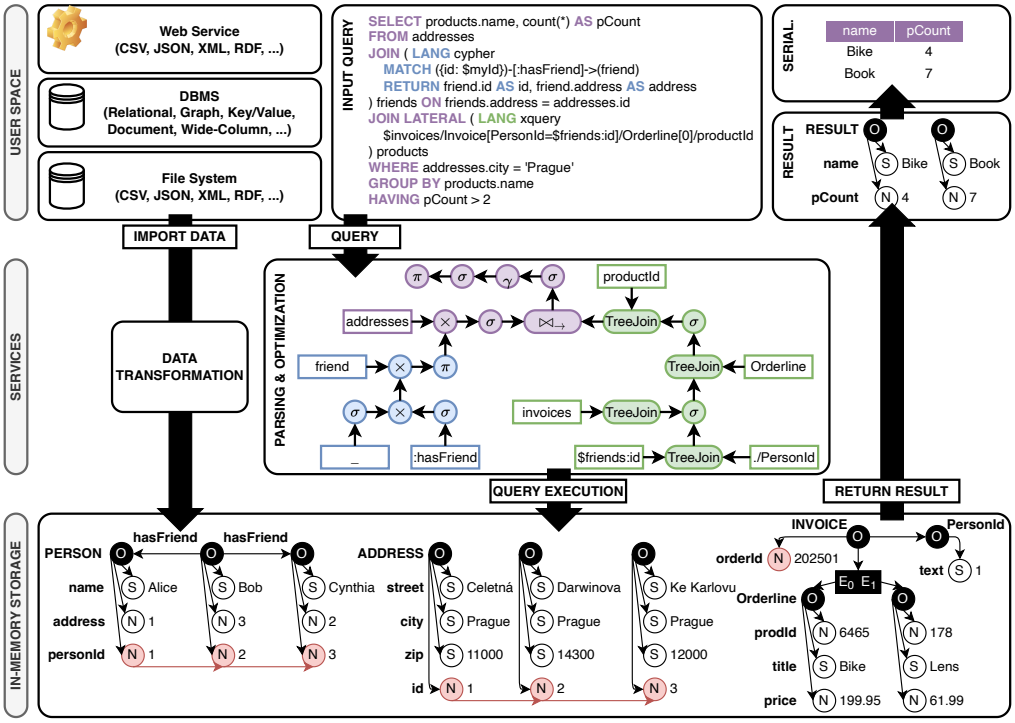

Figure 2: Data import and query evaluation workflow in DortDB.

opaque items, supporting expression evaluation, function calls, and SQL-style quantifiers. The *calculation* (calc) operator represents general expressions, while, e.g., *conditional*, *fnCall*, and *literal* operators construct complex computations. *Ambivalent operators* operate on both tuple and item streams. The *union* (∪), *intersection* (∩), and *difference* (\) operators enable set operations. The *limit* operator constrains output size, while *nullSource* emits a null value. Finally, *XQuery-specific operators* extend tuple operations to hierarchical data structures. The *treeJoin* operator generalizes dependent joins by incorporating hierarchical context, whereas *projectionSize* computes the cardinality of tuple streams.

In DortDB's GUI, the algebra is visualized as a tree, where each inner node represents a logical plan operator. The root of the tree corresponds to the final query, while the leaves represent the individual data sources. Tuple operators display their output schema at their respective nodes. The tree structure uses two types of edges: a *solid edge*, which indicates that the parent operator is bound to an existing data stream, and a *dashed edge*, which signifies that the child operator is dynamically created or reevaluated multiple times.

3 ARCHITECTURE AND IMPLEMENTATION

DortDB is a three-layer library for querying multi-model data (in formats like, e.g., CSV, XML, JSON) directly within a web browser. It is implemented entirely on the client side as a browser module, with no server communication, assuming all data resides in memory. Its architecture comprises an *in-memory storage* layer, a *services* layer, and a *user-space* layer (see Fig. 2), accessible via a unified API for data import and query execution.

The *in-memory storage* layer provides a uniform TypeScript object representation for heterogeneous data formats, enabling model-independent processing. The *service* layer is responsible for data acquisition and query processing. It supports loading data from external sources such as web services, file systems, and local or remote DBMSs. It performs internal transformations of source-specific models to the unified object model. For querying, it accommodates multiple languages through translation wrappers that convert input queries into a unified algebraic form, which is subsequently optimized and executed. Query results are returned as TypeScript objects, with optional serialization to standard formats such as JSON. Finally, the *user-space* layer includes graphical and programmatic interaction. A graphical interface facilitates the construction and validation of queries.

4 RELATED WORK

Embedding one query language within another is commonly used but remains limited in scope. A survey of query languages for multi-model DBMSs [4] highlights a typical pattern – a primary language, typically SQL, embeds (often restricted) subqueries from other data models. This approach is widely adopted by relational DBMSs such as MS SQL Server, Oracle, and PostgreSQL through SQL/XML, SQL/JSON, and SQL/PGQ. It is also employed by key/-value systems like Redis and multi-model DBMSs such as Myria [7] or Apache Drill [5], which extend SQL to access non-relational sources. Similarly, CloudMdsQL [8], a polystore, represents data as tables and accesses NoSQL sources via named table expressions.

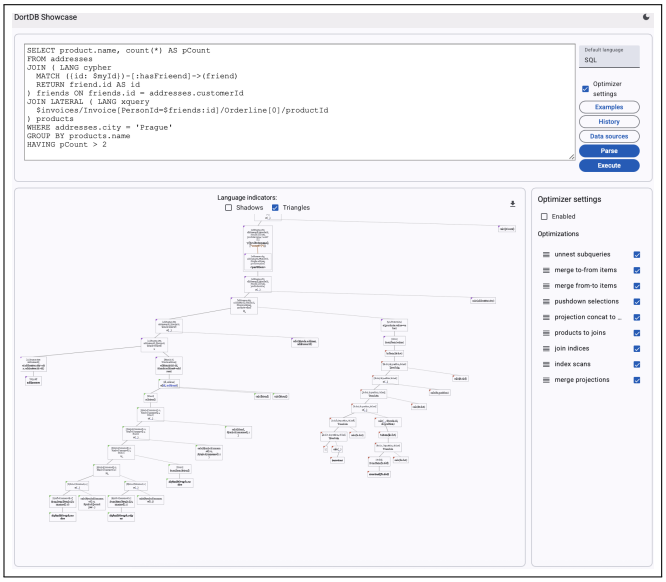

Figure 3: Scenario A – testing and validation GUI for DortDB

```
const db = new DortDB({
  mainLand: SQL(),
  additionalLangs: [
    Cypher( { defaultGraph 'defaultGraph' } ), XQuery(),
  ],
});

db.registerSource(['addresses'], addressCSV);
db.registerSource(['invoices'], invoicesService.firstChild);
db.registerSource(['friends'], friends);

const result = db.query(`
SELECT product.name, count(*) AS pCount
FROM addresses
JOIN ( LANG cypher
  MATCH ({id: $myId})-[:hasFriend]->(friend)
  RETURN friend.id AS id, friend.address AS address
) friends ON friends.address = addresses.id
JOIN LATERAL ( LANG xquery
  $invoices/Invoice[PersonId=$friends:id]/Orderline[0]/productId
) products
WHERE addresses.city = 'Prague'
GROUP BY products.name
HAVING pCount > 2`);
```

Figure 4: Scenario B – TypeScript API for DortDB

Apache AGE[6], a PostgreSQL extension, enables Cypher queries via the embedded `cypher()` function. However, it only supports translation from Cypher to SQL, without bidirectional interoperability.

In contrast, DortDB allows any supported query language to be nested within any other, removing the constraint of a fixed top-level language (as seen in polystore BigDAWG [3]). Furthermore, nesting is not restricted to a single level but can extend to arbitrary depths. Unlike, e.g., OrientDB and ArangoDB, which rely on custom syntax extensions, DortDB supports multiple query languages natively, enabling seamless reuse and reducing the learning curve. Additionally, DortDB does not restrict specific query languages to particular data models, allowing users to choose based on their preferences – whether for efficiency, backward compatibility, or familiarity.

5 DEMONSTRATION OUTLINE

In the DortDB demonstration, the users will engage in querying multi-model data using the representation and query language they consider intuitive and/or most efficient. The system supports iterative refinement to improve performance and expressiveness.

Scenario A: Users can interact with a GUI designed for testing and validation (see Fig. 3). They can create their own queries by combining SQL, Cypher, and XQuery or select predefined queries from the UniBench benchmark [12] for benchmarking multi-model applications, executed on preloaded UniBench datasets. Upon submission, DortDB parses the query and generates a logical execution plan visualised as a tree, allowing users to inspect and verify its correctness without executing the query. This scenario emphasizes syntactic and structural validation in a multi-model context.

Scenario B: The second scenario offers a programmatic interface through the TypeScript API (see Fig. 4). Users may work with the UniBench dataset or upload and query a dataset of their choice imported from external web services, remote databases, or local files. After data import, users construct and execute queries using the API, which internally translates them into a unified algebra and applies query optimization. The results are returned in an object-oriented format, enabling further processing or serialization.

[6]https://age.apache.org/

ACKNOWLEDGMENTS

Supported by grants: GAČR no. 23-07781S, and GAUK no. 292323.

REFERENCES

[1] Amazon Web Services. 2023. What is a Data Lake? - Introduction to Data Lakes and Analytics. *AWS* (2023). https://aws.amazon.com/what-is/data-lake/
[2] Renzo Angles, Angela Bonifati, Roberto García, and Domagoj Vrgoc. 2025. Path-based Algebraic Foundations of Graph Query Languages. In *Proc. of EDBT 2025*. OpenProceedings.org, 783–795.
[3] Jennie Duggan, Aaron J. Elmore, Michael Stonebraker, Magdalena Balazinska, Bill Howe, Jeremy Kepner, Sam Madden, David Maier, Tim Mattson, and Stanley B. Zdonik. 2015. The BigDAWG Polystore System. *SIGMOD Record* 44 (2015).
[4] Qingsong Guo, Chao Zhang, Shuxun Zhang, and Jiaheng Lu. 2023. Multi-Model Query Languages: Taming the Variety of Big Data. *J. of Distributed and Parallel Databases* 42 (2023), 1–41.
[5] Michael Hausenblas and Jacques Nadeau. 2013. Apache Drill: Interactive Ad-Hoc Analysis at Scale. *J. of Big Data* 1, 2 (2013), 100–104.
[6] Jürgen Hölsch, Michael Grossniklaus, and Marc H. Scholl. 2016. Optimization of Nested Queries using the NF2 Algebra. In *Proc. of SIGMOD 2016*. ACM.
[7] Jingjing Wang et al. 2017. The Myria Big Data Management and Analytics System and Cloud Services. In *Proc. of CIDR 2017*. www.cidrdb.org.
[8] Boyan Kolev, Carlyna Bondiombouy, Patrick Valduriez, Ricardo Jiménez-Peris, Raquel Pau, and José Pereira. 2016. The CloudMdsQL Multistore System. *Proc. of SIGMOD 2016* (2016), 2113–2116.
[9] Jiaheng Lu and Irena Holubová. 2019. Multi-Model Databases: A New Journey to Handle the Variety of Data. *ACM Comput. Surv.* 52, 3, Article 55 (June 2019).
[10] Christopher Ré, Jérôme Siméon, and Mary F. Fernández. 2006. A Complete and Efficient Algebraic Compiler for XQuery. In *Proc. of ICDE 2006*. IEEE, 14.
[11] Ran Tan, Rada Chirkova, Vijay Gadepally, and Timothy G. Mattson. 2017. Enabling query processing across heterogeneous data models: A survey. In *Proc. of BigData 2017*. IEEE, 3211–3220.
[12] Chao Zhang, Jiaheng Lu, Pengfei Xu, and Yuxing Chen. 2019. UniBench: A Benchmark for Multi-model Database Management Systems. In *Proc. of TPCTC 2018*. Springer, Cham, 7–23.