

**FACULTY
OF MATHEMATICS
AND PHYSICS**
Charles University

MASTER THESIS

Filip Ježek

In-memory database for web browser

Department of Software Engineering

Supervisor of the master thesis: RNDr. Michal Kopecký, Ph.D.

Study programme: Computer Science - Software and
Data Engineering

Prague 2026

I declare that I carried out this master thesis on my own, and only with the cited sources, literature and other professional sources. I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In date

Author's signature

I would like to sincerely thank my supervisor, Michal Kopecký. Similarly, I would like to thank Irena Holubová, Pavel Koupil, and Jáchym Bártík (in no particular order). I very much appreciate the hours spent on consultations, the fresh ideas, the invaluable feedback, the patience, and the enthusiasm. I also want to express my gratitude to my friends and family for their unconditional support.

Title: In-memory database for web browser

Author: Filip Ježek

Department: Department of Software Engineering

Supervisor: RNDr. Michal Kopecký, Ph.D., Department of Software Engineering

Abstract: This thesis introduces DortDB, a TypeScript in-memory database. DortDB is entirely modular and extensible. It can be configured with different query languages and can combine multiple languages in a single query. To this end, we have designed the unified algebra that supports relational, document, and graph models. We have implemented a query optimizer that optimizes the combined logical plan across language boundaries, as well as a system for extensible secondary indices. Part of the thesis is also the DortDB Showcase, a GUI for inspecting logical plans, experimenting with various optimizations, and querying data. We have evaluated DortDB with other multimodel databases on the UniBench benchmark and compared it with two JavaScript in-memory SQL databases using the TPC-H queries.

Keywords: SQL, Relational Data Model, JSON, In-memory database, Multimodel database

Název práce: In-memory databáze pro webový prohlížeč

Autor: Filip Ježek

Katedra: Katedra softwarového inženýrství

Vedoucí bakalářské práce: RNDr. Michal Kopecký, Ph.D., Katedra softwarového inženýrství

Abstrakt: Tato práce popisuje DortDB, in-memory databázi implementovanou v TypeScriptu. DortDB je plně modulární a rozšiřitelná. V DortDB lze používat různé dotazovací jazyky, které je možné střídat i libovolně vnořovat v rámci jednoho dotazu. Za tímto účelem jsme navrhli sjednocenou algebru, která podporuje relační, dokumentový i grafový datový model. Implementovali jsme optimalizátor dotazů, který je díky sjednocené algebře schopen optimalizovat dotazy napříč různými jazyky. Dále jsme navrhli systém rozšiřitelných indexů. Součástí práce je rovněž DortDB Showcase, grafické uživatelské rozhraní pro zkoumání logických plánů, experimentování s různými optimalizacemi a dotazování se nad daty. DortDB jsme vyhodnotili spolu s jinými multimodelovými databázemi na UniBench testech. Také jsme DortDB srovnali se dvěma dalšími in-memory JavaScriptovými SQL databázemi na dotazech z TPC-H.

Klíčová slova: SQL, Relační model dat, JSON, In-memory databáze, Multimodelová databáze

Contents

Introduction	8
1 Background	11
1.1 Web Browser Environment	11
1.2 Multimodel Data	12
1.2.1 Relational Model	12
1.2.2 Document Model	13
1.2.3 Graph Model	13
1.2.4 Key-value Model	14
2 Related work	15
2.1 Existing Approaches to Cross-Model Querying	15
2.1.1 Extending a Dominant Query Language	15
2.1.2 System-Defined Query Languages	15
2.1.3 Nesting and Combining Existing Languages	16
2.1.4 Supporting Multiple Top-Level Languages	16
2.2 Multimodel Systems	17
2.2.1 ArangoDB	17
2.2.2 OrientDB	17
2.2.3 MMQL	18
2.2.4 SQL/PGQ	19
2.3 JavaScript In-memory SQL Databases	19
2.3.1 AlaSQL	20
2.3.2 SQL.js	20
3 Basic concepts	22
3.1 Design Principles	22
3.1.1 Language Neutrality	22
3.1.2 Schema Agnosticism	22
3.1.3 Algebra as Compilation Target	22
3.1.4 Multilanguage Queries	22
3.1.5 Extensibility and Modularity	23
3.2 Architecture Overview	23
3.3 Limitations	24
3.3.1 Optimizer	24
3.3.2 Data Modification	24
3.3.3 Parallelism	24
3.3.4 Language Limitations	24
4 Implemented Languages	25
4.1 SQL	25
4.1.1 Notable Features	25
4.1.2 Schema of Data Sources	26
4.1.3 Not Implemented Features	27
4.2 XQuery	27

4.2.1	Not Implemented Features	28
4.3	Cypher	29
4.3.1	Cypher in DortDB	30
5	Formalization of Cross-Model Querying	31
5.1	Language switching	31
5.2	Unified Object Representation	32
5.2.1	Comparison with Nested Relations	34
5.2.2	Comparison with Property Graph Model	34
5.3	Unified Algebra	34
5.3.1	Operator Context	35
5.3.2	Visualization	42
5.3.3	Item Operators	42
5.3.4	Tuple Operators	44
5.3.5	Universal Operators	44
5.3.6	Mapping of SQL to the Algebra	47
5.3.7	Mapping of XQuery to the Algebra	47
5.3.8	Mapping of Cypher to the Algebra	48
6	Optimization	49
6.1	Plan Operators	49
6.2	Secondary Indices	52
6.3	Implemented Optimizer Rules	52
6.3.1	Unnest Subqueries	52
6.3.2	Merge To/FromItems	54
6.3.3	Pushdown Selections	55
6.3.4	ProjectionConcat to Join	57
6.3.5	Products to Joins	58
6.3.6	Join Indices	58
6.3.7	Index Scans	59
6.3.8	Merge Projections	60
6.4	Calculation Building	61
6.5	Language-Specific Optimization	61
7	Benchmarks	63
7.1	UniBench	63
7.1.1	Data Import	64
7.1.2	Query Performance	65
7.2	SQL	66
8	Implementation	70
8.1	Core Library	70
8.1.1	Language Manager	70
8.2	Query Process	71
8.2.1	Lexer and Parser	71
8.2.2	Plan Builder	72
8.2.3	Optimizer et al.	73
8.2.4	Variable Mapper	73
8.2.5	Data Adapters	74

8.2.6	Executor	74
9	Usage	77
9.1	Showcase	77
9.2	Programmatic Usage	83
9.3	Extensibility	83
9.3.1	Authoring a New Language	83
9.3.2	Authoring a New Index Type	85
9.3.3	Authoring Other Components	87
10	Future Work	89
10.1	Language Improvements	89
10.2	Additional Optimization	89
10.3	New Features	90
	Conclusion	91
	Bibliography	93
	List of Figures	95
A	UniBench Queries	97
B	TPC-H Queries	104
C	AST in More Detail	123
D	Source Code	129

Introduction

Foreword

In recent years, the frontends of websites and web applications have grown increasingly more complex. Faster networks let us deliver more JavaScript while keeping loading times within reasonable bounds. In many aspects, the architecture and scale of an interactive webpage are similar to those of a desktop application. Often, this comes hand in hand with a large volume and variety of application data kept in memory at any given time, which must be processed efficiently. Sometimes, it may even be desirable to give end users the option to interact with such data directly. One possible solution is to introduce a way to query the JavaScript data structures using a standard query language.

In this thesis, we will present an in-memory database built on top of existing JavaScript application data, implemented as a TypeScript package called DortDB. It enables scenarios where users visiting a website work directly with application data, for example, when filtering or projecting tables. It also allows programmers to simply and declaratively execute complex operations on diverse data while potentially speeding them up compared to a naive approach.

We will also explore cross-model querying over heterogeneous data and introduce an approach that combines multiple query languages rather than designing a completely new universal language. This approach allows us to express queries spanning multiple data models intuitively. It is also inherently extensible and modular, making it suitable for scenarios that require catering to diverse requirements across applications.

Goals

This thesis was undertaken with several concrete goals in mind, each contributing to the overall aim of advancing DortDB as a capable web browser database. These goals were defined at the outset and guided the direction of the work throughout. They are as follows:

G1 (Package Size) Our target environment is the web browser, meaning every byte counts. The package must be as small as possible. In modern web development, code that uses third-party packages is usually bundled, and unused package exports and dead code branches are discarded during a process called tree shaking. Examples of bundlers offering such functionality include esbuild¹ or Rolldown.² To save space, the functionality of DortDB must be decomposed into many small, decoupled units. The consuming developer would then import only those parts of DortDB he wants to use, thus avoiding bundling features that would ultimately be just dead weight. For instance, an application working with XML documents likely has no need for graph query languages.

¹<https://esbuild.github.io/>

²<https://rolldown.rs/>

G2 (Languages) Given the many diverse shapes that application internal data structures may take, the question of a suitable query language naturally arises. Different languages are appropriate for different tasks, and each developer can have entirely different requirements. We want to ensure viability for as many users as possible.

G3 (Extensibility) Building on the previous two points, modularity and adaptability, we need to introduce opportunities for user extensions. The amount of work we can offer in this thesis is not infinite, and there are sure to be many features we will be unable to add, whether due to time constraints or a lack of imagination. We must therefore take care to design clear interfaces and extension points. Developers using this library must have the option to add their custom in-language functions, aggregators, or perhaps operators. We should explore the possibility of non-trivial plugins, e.g., new index types.

G4 (AST) Parsing a language is not easy. When designing a user interface for end users to enter their own queries, one would usually want to implement some form of syntax highlighting or code formatting. Exposing the abstract syntax tree that the package itself builds would definitely make the task more manageable. It would also enable programmatic manipulation of previously entered queries, such as inverting a sort field in response to user interaction.

G5 (Optimizer) Last but not least, we must strive to execute the queries as fast and efficiently as possible. We know that parsing text input and building ASTs will incur significant overhead, especially for simple queries on small datasets. The main benefit of DortDB will be the ease of expression when the queries get really complex. Nevertheless, we need to minimize the impact on speed as much as possible. Our approaches will include query plan optimization and the creation of secondary indices.

Thesis Structure

This thesis is structured into 10 numbered chapters. The overview of the chapters is as follows:

Background In the first chapter, we will briefly cover the required background for the thesis. We will outline the current landscape of modern web development and introduce the concept of multimodel data, highlighting its relevance and growing importance in database systems.

Related work The second chapter surveys related work in the areas of multimodel queries and in-memory JavaScript databases.

Basic concepts In the third chapter, we will introduce DortDB, the software developed as a part of this thesis. We will describe its design and the motivation behind its development, setting the foundation for the discussions in subsequent chapters.

Implemented Languages The fourth chapter takes a closer look at the query languages implemented during the development of DortDB. It outlines the core syntax and semantics of each language, compares them with their official specifications, and explains the design choices and trade-offs made during implementation.

Formalization of Cross-Model Querying The fifth chapter presents the concept of multilanguage queries and language switching. It introduces the unified algebra, which forms the formal foundation of DortDB.

Optimization In the sixth chapter, we address the query optimization strategies applied in DortDB. We describe the optimizer, the design of secondary indices, and detail the techniques used to improve performance.

Benchmarks Chapter seven presents benchmarking results for DortDB. We evaluate the framework by comparing execution times with those of other established systems.

Implementation The eighth chapter provides a deep dive into the implementation details of DortDB. It covers key modules, the overall architecture, and the query execution pipeline.

Usage In chapter nine, we demonstrate how DortDB can be used in practice. This includes the DortDB GUI and its developer API. The goal is to show how the system can be adopted and extended by users and developers.

Future Work Finally, chapter ten outlines directions for future work. We discuss potential extensions to DortDB, such as support for additional data models and languages or further optimization opportunities.

1 Background

1.1 Web Browser Environment

The web browser is an environment that differs from other runtimes in several key ways. Application code (with some exceptions) has to be sent over the network each time the user accesses a website. This often happens in less-than-ideal conditions, such as on a cellular network. In 2026, mobile devices were responsible for 55% of web requests [1]. It is imperative to reduce the application size as much as possible. The median web page size was 2900 KB and 2600 KB for desktop and mobile user agents. Of this, JavaScript made up 750 KB and 660 KB, respectively [2].

The JavaScript code used on the frontend is, in large part, asynchronous. The program must respond to many events, be it user input or network communication. Regardless, it is simple to avoid any race conditions because the code is not truly parallel. JavaScript runs in a single thread, and asynchronicity is implemented using the event loop. The event loop is a job queue that schedules code that waits for the completion of an asynchronous action via a callback. These callbacks are guaranteed to be entirely processed before any other job is handled. The event loop divides jobs into microtasks and macrotasks, with microtasks having higher priority and consisting mainly of Promises.

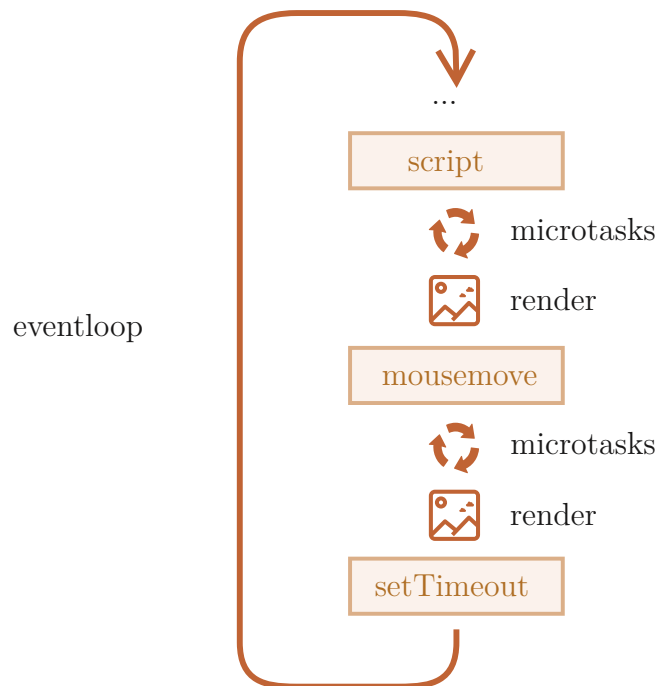


Figure 1.1 Event loop diagram. *Ilya Kantor*.¹

JavaScript in web browsers provides a limited form of multithreading via Web Workers. A Web Worker runs a separate JavaScript file in a separate thread and

¹<https://github.com/javascript-tutorial/en.javascript.info/blob/540d753e90789205fc6e75c502f68382c87dea9b/2-ui/99-ui-misc/03-event-loop/eventLoop-full.svg>

context. To avoid concurrency issues, a Web Worker cannot access the main thread's context. Instead, it must communicate with the main thread using a message API. Objects in the messages are cloned rather than referenced. Some specific objects, such as `ArrayBuffers`, can be transferred instead of copied.

Modern web browsers support running WebAssembly alongside JavaScript. WebAssembly is a compilation target for languages like Rust or C++, and it offers near-native performance. Existing applications can be compiled into WebAssembly and run in the browser. One such example is `SQL.js`,² which is, in fact, a recompiled version of `SQLite`. WebAssembly modules can also export (and import) JavaScript functions, allowing interoperability between the two. It is possible to pass complex values to WebAssembly, but this is again done by copying and serializing the data. Some values currently cannot be passed at all, including non-builtin class instances.

1.2 Multimodel Data

Historically, database systems stored data in a single model, usually the relational model. While this has many benefits, such as consistency and uniformity in modeling, in some use cases, it may make more sense to combine data across multiple models. The key motivation behind multimodel data is the increasing complexity and variety of modern data workloads. Different types of applications require different data representations for optimal performance, scalability, and flexibility. For instance, relational tables are well-suited for structured transactional data; JSON or XML documents allow semi-structured storage of units of data, e.g., in e-commerce; and graph models are ideal for capturing complex relationships, such as social networks.

The multimodel data storage can be divided into polystores and multimodel databases. Polystores are systems that combine multiple heterogeneous databases. The queries are split into subqueries issued to the underlying databases, and the results are combined by the polystore and presented back to the user. A leading example of polystores is `BigDAWG` [3]. On the other hand, multimodel databases combine multiple data models in a single database, often extending the query language of one model with constructs from the others. The existing multimodel databases are compared in a survey [4].

1.2.1 Relational Model

The relational data model is a foundational model for data management, first introduced by E. F. Codd in 1970 [5]. It organizes data into relations, which are formally defined as sets of tuples and, more intuitively, as tables consisting of rows and columns. Each relation has a schema that defines its structure. The model can be queried with relational algebra or relational calculus, but the de facto standard relational query language is SQL.

²<https://sql.js.org/>

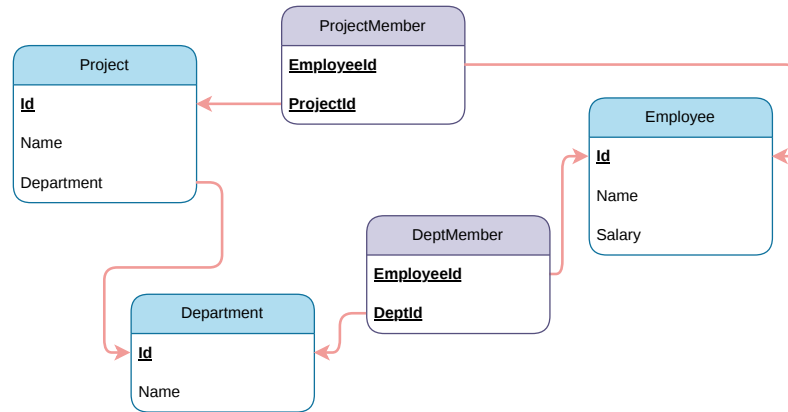


Figure 1.2 Relational model.

1.2.2 Document Model

In the document model, the primary unit of data is a document. Documents are semi-structured self-describing aggregates, typically encoded as JSON or XML. The document model offers greater schema flexibility at the cost of potential data redundancy. It is efficient for hierarchical data, but is not suitable for relational operations such as joins.

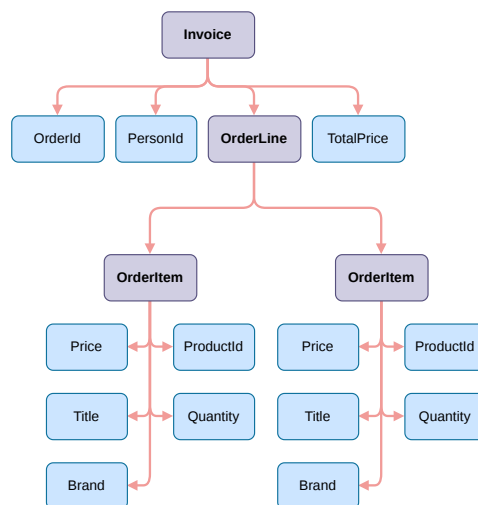


Figure 1.3 Document model.

1.2.3 Graph Model

The graph model is based on mathematical graphs, which model data into vertices (nodes) and edges (relationships). Graph query languages are either declarative, such as Cypher³ or GQL⁴, or imperative, describing how the graph is

³<https://opencypher.org/>

⁴<https://www.iso.org/standard/76120.html>

to be traversed, such as Gremlin⁵ or AQL.⁶

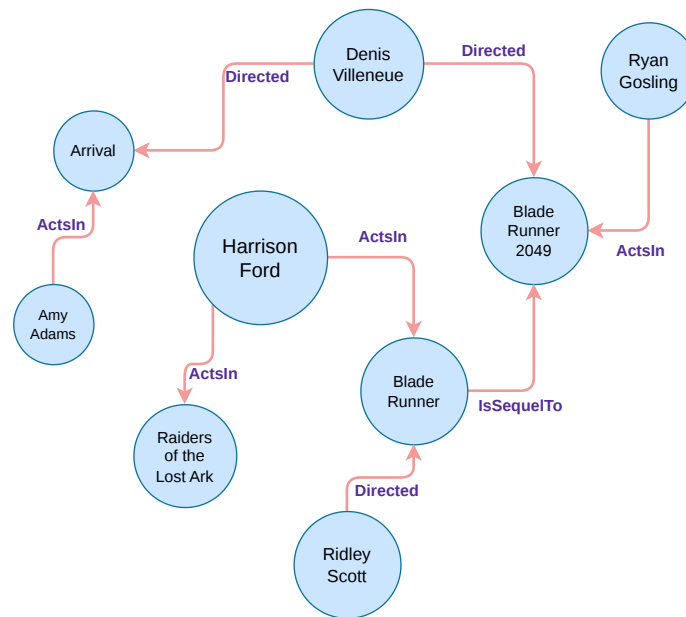


Figure 1.4 Graph model.

1.2.4 Key-value Model

The key-value model is very simple. It may be thought of as a hash table or an associative array. It is designed to store and retrieve opaque values with unique keys quickly, but it is not well-suited for transactional or analytical workloads.

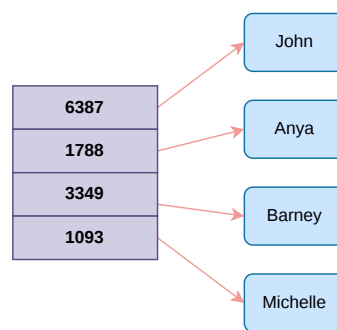


Figure 1.5 Key-value model.

⁵<https://tinkerpop.apache.org/gremlin.html>

⁶<https://docs.arango.ai/arangodb/stable/aql/>

2 Related work

Before we delve into DortDB and its concepts, we need to acquaint ourselves with the landscape of multimodel databases. First, we will explore the current approaches to querying multimodel data. Then, we will inspect several concrete systems more closely. Lastly, we will examine leading in-memory JavaScript databases.

2.1 Existing Approaches to Cross-Model Querying

There is a wide range of approaches to cross-model querying. Existing systems differ mainly in how they balance three competing goals: (1) preserving the expressive power of individual data models, (2) providing a unified querying interface, and (3) enabling effective optimization across heterogeneous data. A central design decision concerns the role of the query language. For the purposes of this thesis, we distinguish four recurring patterns: extending one dominant language with constructs for additional data models, relying on a system-defined query language, combining existing languages through controlled embedding or nesting, and supporting more than one top-level query language. The boundaries between these categories are not always strict, but the distinction helps clarify the main design choices adopted by existing systems.

2.1.1 Extending a Dominant Query Language

A common strategy is to preserve one primary query language, often SQL or an SQL-like language, and extend it with mechanisms for accessing additional data models. A survey of query languages for multimodel DBMSs [6] identifies this as one of the dominant patterns in current systems. In originally relational DBMSs such as MS SQL Server, Oracle, and PostgreSQL, this direction is reflected in standardized extensions including SQL/XML [7], SQL/JSON [8], and SQL/PGQ [9]. Related ideas also appear in systems such as Myria [10], Apache Drill [11], and OrientDB,¹ although these systems differ in architecture and in the precise role played by SQL or SQL-like syntax. The main advantage of this family of approaches is the reuse of a familiar query style. Its main limitation is that a single fixed outer language still determines the overall structure of the query and the results.

2.1.2 System-Defined Query Languages

Another line of work relies on a query language defined by the system itself rather than on extensions of an existing standard language. This is the case, for example, for Redis [12], MongoDB,² and ArangoDB.³ These systems differ

¹<https://orientdb.dev/>

²<https://www.mongodb.com/>

³<https://arangodb.com/multimodel-db/>

substantially in scope and underlying data model, and not all of them target cross-model querying in the same sense. ArangoDB is a particularly relevant example here, since its AQL (ArangoDB Query Language) provides access to document, graph, key-value, search, and other data abstractions within a single language. Neo4j⁴ provides a related but narrower case: it relies on Cypher as a system-defined language specialized for graph querying rather than for general multimodel access.

The Multi-Model Query Language (MMQL) [13] represents a more explicit multimodel example: it is grounded in category theory, uses SPARQL-inspired syntax, and translates a query into multiple model-specific queries dispatched to connected systems. The advantage of such approaches is conceptual consistency within the provided language. Their limitation is that users must adopt a system-specific language and, in many cases, also the abstraction that this language makes primary.

2.1.3 Nesting and Combining Existing Languages

A different line of work starts from the opposite design choice: instead of defining a new system-level language, it attempts to preserve existing query languages and integrate them through controlled forms of embedding or nesting. Apache AGE⁵ enables Cypher queries to be embedded into SQL through the `cypher()` function in the `FROM` clause, but only in the Cypher-to-SQL direction. SQL/XML also supports evaluating XQuery fragments passed as text arguments to selected functions. Similar to Apache AGE, OpenLink Virtuoso’s SPASQL [14] (SPARQL-in-SQL) allows for one-directional embedding of SPARQL subqueries into SQL. CloudMdsQL [15] follows a related idea in a polystore setting: it presents data in tabular form, accesses NoSQL sources through named table expressions, and incorporates native query fragments into the overall execution process. Its optimization techniques include predicate pushdown and other strategies to reduce data transfer. These approaches preserve the use of established languages, but they typically rely on a fixed outer language and do not support unrestricted bidirectional composition. This, in turn, raises the question of whether multiple languages can coexist not only as nested fragments, but also as genuine top-level entry points into a unified query-processing framework.

2.1.4 Supporting Multiple Top-Level Languages

The next step in this progression is to relax the assumption of a single outer language altogether. In practice, however, support for multiple top-level query languages remains relatively rare. Some systems nevertheless provide partial support for this idea. For example, ArangoDB allows access not only through AQL but also through GraphQL, and OrientDB supports Gremlin in addition to its SQL-style language. A more systematic example is BigDAWG [3], which introduces the notion of *islands*, i.e., access points that expose different model abstractions together with their corresponding query languages. Queries are translated through *shims*, while data movement across systems is handled by

⁴<https://neo4j.com/>

⁵<https://age.apache.org/>

casts. Such systems relax the restriction to a single top-level language, but they usually separate languages into predefined access modes and do not support their unrestricted recursive combination within a single unified query.

Overall, existing approaches provide important partial solutions, but they usually retain one dominant language, one dominant abstraction, or only limited forms of language composition. These recurring limitations motivate the design of DortDB as a framework that supports symmetric language combination through a shared algebraic foundation, without privileging one language as the fixed outer layer.

2.2 Multimodel Systems

Several of the above-mentioned databases or technologies deserve a more detailed description. ArangoDB and OrientDB later appear in the benchmark evaluation in Chapter UniBench. Their selection for the benchmark was largely practical – the original benchmark authors provided ready-made import scripts for these systems, making them straightforward to include.

2.2.1 ArangoDB

ArangoDB is a highly popular multimodel database. Its base data model is a graph, but it also supports vector data, hierarchical document data, key-value, and full-text search. ArangoDB features its own query language, AQL, which can seamlessly combine all the mentioned data models. Its query optimizer is very powerful, as evidenced by our benchmarks.

Program 1 Example of an ArangoDB version of UniBench query 2 written in AQL. It selects customers who have bought a specific product and posted about it. The query combines graph and document models.

```
LET personlist = (  
  FOR post IN INBOUND CONCAT("Product/", @key) PostHasTag  
  FOR person IN INBOUND post._id PersonHasPost  
  LIMIT 100  
  RETURN person._key  
)  
FOR order IN Order  
FILTER order.OrderDate > "2022"  
AND @key IN order.Orderline[*].productId  
AND order.PersonId IN UNIQUE(personlist)  
RETURN DISTINCT(order.PersonId)
```

2.2.2 OrientDB

OrientDB is an open-source document and graph database. It claims to be the world's first multimodel database. It offers tools for ETL (extract, transform, load) pipelines. The system combines SQL-style querying with graph-oriented

constructs such as **MATCH**, **TRAVERSE**, and navigation operators such as **IN**. This allows graph-shaped data to be queried using SQL-like syntax, although the resulting language remains centered on SQL. The language, however, does not use joins. Instead, entities are connected via *links*, which store the rowids of related items as part of the original entity. This results in constant-time lookups, but links must be defined ahead of time, and there is no support for ad-hoc joining.

Program 2 The same query written in OrientDB SQL.

```
SELECT $person
LET $list = (
  SELECT IN('PostHasTag').IN('PersonHasPost').id AS pid
  FROM `Product`
  WHERE productId = :id
), $person = (
  SELECT PersonId, Orderline.productId
  FROM Order
  WHERE OrderDate > "2022"
  AND PersonId IN $list
  AND :id IN Orderline.productId
)
```

Program 3 Example of link traversal in OrientDB.

```
-- instead of joins like this:
SELECT *
FROM Employee A, City B
WHERE A.city = B.id
AND B.name = 'Rome'

-- OrientDB uses links like so:
SELECT * FROM Employee WHERE city.name = 'Rome'
```

2.2.3 MMQL

Multimodel query languages are generally designed with a selected set of data models in mind. MMQL is a query language based on category theory that attempts to cover all possible data models. Practically, it is based on SPARQL⁶ syntax, where SPARQL is a W3C-endorsed graph query language developed for querying linked data. The implemented proof of concept parses the query into multiple model-specific queries, which are dispatched to connected database systems. MMQL then combines the results, joining data and handling data redundancies. The proposed approach bears similarities to polystores, but it aims to be universal in its support of existing database systems and data models.

⁶<https://www.w3.org/TR/sparql11-query/>

Program 4 Example of a query written in MMQL, taken from the original thesis [16].

```
SELECT {
  ?customer ordered ?productName ;
    name ?customerName .
}
WHERE {
  ?product 49 ?productName ;
    -39/36 ?order .

  ?order -23/21 ?customer .
  ?customer 3 ?customerName .

  FILTER(?productName = "Lord of the Rings")
}
```

2.2.4 SQL/PGQ

SQL/PGQ (Property Graph Queries) became an ISO standard in 2023. It extends SQL with a `GRAPH_TABLE` construct in the `FROM` clause, enabling graph traversal through a powerful pattern matching syntax [17]. Unlike SQL/XML, references to variables from an outer scope feel natural, since graph pattern matching is integrated directly into the language rather than being passed around as text literals.

Program 5 Example of a query written in SQL/PGQ.

```
SELECT account_holder, SUM(amount) AS total_transacted_with_Nikita
FROM GRAPH_TABLE ( financial_transactions
  MATCH (p IS Person) <-[IS owner]- (account1 IS Account),
    -- match both incoming and outgoing transactions
    (account1) -[t IS transaction]- (account2),
    (account2 IS Account) -[IS owner]-> (owner IS Person|Company)
  WHERE p.name = 'Nikita'
  COLUMNS (owner.name AS account_holder, t.amount)
)
GROUP BY account_holder
```

2.3 JavaScript In-memory SQL Databases

The main stated goal of this thesis is to design an in-memory JavaScript database that supports querying application data using a query language. In this section, we examine existing JavaScript packages that promise the same functionality.

2.3.1 AlaSQL

AlaSQL⁷ is an open-source in-memory SQL database for JavaScript. It comes with a large number of various features. It supports most of the SQL-99 standard, includes syntax for interacting with JSON documents, and a limited form of graph queries. It allows the developer to query existing data structures and to define custom functions and aggregates. It is also possible to query and modify data stored in XLSX, CSV, or JSON files.

At the same time, the accumulation of these features appears to have evolved in a rather ad hoc way. As a result, the codebase is difficult to maintain and extend, and its performance is limited for more complex queries. Query plans are compiled into a single JavaScript function using text concatenation. Although this design can make execution efficient for simpler cases, it does not provide a suitable basis for more advanced optimization.

AlaSQL exposes the ASTs created by parsing queries. The classes themselves, however, cannot be imported (and thus instantiated, unless resorting to less savory programming practices), and there is no straightforward way to execute a modified AST. Additionally, AlaSQL’s monolithic architecture means its rich feature set comes at the cost of a large package size that is not amenable to tree-shaking.

2.3.2 SQL.js

SQL.js⁸ is SQLite⁹ compiled into WebAssembly. As a result, it is highly performant and stable, owing to more than 20 years of development. It can be used as an in-memory database, but it can also store data in special database files. SQLite claims to be the world’s most used database engine. It is possible to define custom functions and aggregates.

On the other hand, since SQL.js runs in WebAssembly, all data it receives from JavaScript must be serialized and deserialized. It can therefore not be used to query existing JavaScript data structures. Additionally, it naturally does not expose the AST or the logical plan of a parsed query (besides the output of the EXPLAIN command).

⁷<https://github.com/AlaSQL/alasql>

⁸<https://sql.js.org/>

⁹<https://sqlite.org/>

Program 6 Example of a query written in AlaSQL.

```
// CSV file as a virtual table
// id, product_code, qty
// 1, P100, 3
// 2, P200, 5
alasql(`CREATE TABLE sales SOURCE("sales.csv")`);

const products = [
  {"code": "P100", "info": {
    "name": "Bananas", "category": "Fruit"
  }},
  {"code": "P200", "info": {
    "name": "Bread", "category": "Bakery"
  }},
];

// multimodel join
const res = alasql(`
  SELECT
    s.id AS sale_id,
    s.qty,
    p.info->name AS product_name,
    p.info->category AS product_category
  FROM sales s
  JOIN ? p ON s.product_code = p.code
`, [products]);
```

Program 7 Example of a query written in SQL.js.

```
await initSqlJs(); // loads WASM

const db = new SQL.Database();
db.run("CREATE TABLE test (col1, col2);");
db.run("INSERT INTO test VALUES (?,?), (?,?)", [1,111,2,222]);

const contents = db.exec("SELECT * FROM test");
```

3 Basic concepts

In this chapter, we will introduce DortDB, the software written as part of this thesis. DortDB is a TypeScript framework for querying existing in-memory data. It aims to be as configurable and modular as possible. DortDB is released as an open-source library on NPM¹ (Node.js package manager).

Additionally, we have developed a graphical user interface for DortDB. It can be used to experiment with query optimizer strategies, inspect logical query plans, or query data.

3.1 Design Principles

First, we will provide an overview of the core ideas behind DortDB.

3.1.1 Language Neutrality

The framework aims to offer a common ground for querying data regardless of the language used. Different languages have different strengths; some may be more intuitive or familiar for a given set of tasks, and some may achieve the same task more concisely. Ultimately, though, the operations each language allows are rather similar. For example, pattern matching in graphs could be understood as a sequence of selections and indexed joins.

We separate the languages from the actual query planning and execution, and instead let the user select which languages to use via plugins.

3.1.2 Schema Agnosticism

DortDB is written for the web browser. Its intended purpose is querying in-memory JavaScript objects. JavaScript is a dynamically typed and very permissive language. As such, DortDB adopts a schema-free approach, enabling dynamic and flexible data management without the constraints of predefined schemas.

3.1.3 Algebra as Compilation Target

To properly decouple the languages from the rest of the framework, each language expression is parsed into the unified algebra. DortDB can then work with algebraic operators rather than language-specific constructs.

3.1.4 Multilanguage Queries

Since all languages are parsed into a common representation, DortDB allows users to mix multiple languages in a single query. This is done using nested language subqueries.

¹<https://www.npmjs.com/package/@dortdb/core>

3.1.5 Extensibility and Modularity

A fundamental goal of DortDB is extensibility. While new languages can rely on the framework to do most of the heavy lifting, each part of the process can be customized. If the new language uses a new concept or even a completely new data model, it is easy to extend the algebra with new logical operators. The same may be done for optimization purposes. For example, XQuery path steps are achieved with the XQuery custom `TreeJoin` operator, which combines the functionalities of `ProjectionConcat`, `ProjectionIndex`, and `ProjectionSize` operators into one (see Section Unified Algebra for the description of the operators). Similarly, while we could implement shortest-path matching in graphs using a combination of `OrderBy` and `Limit` operators, if we wanted the query to ever finish, we should implement a smarter custom operator instead.

Furthermore, the framework aims to be as modular as possible to achieve small bundle sizes when deploying to the web.

3.2 Architecture Overview

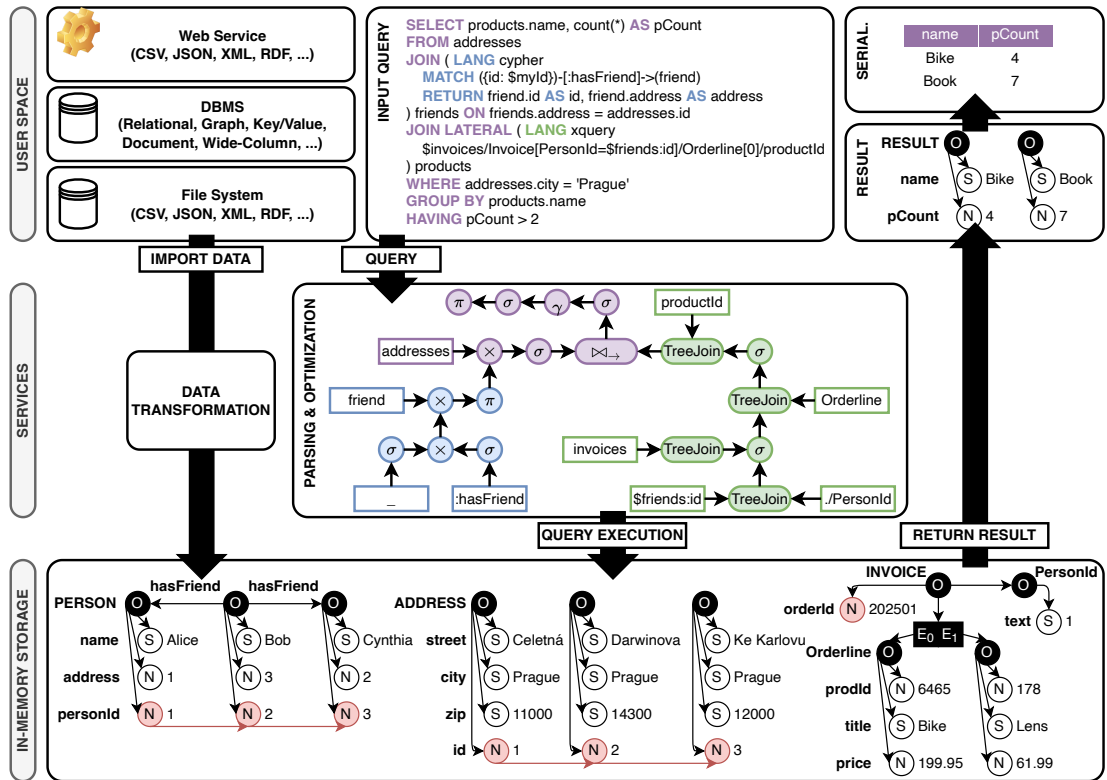


Figure 3.1 Data import and query evaluation workflow in DortDB [18].

A high-level overview of DortDB is as follows. First, existing in-memory data needs to be registered as a DortDB data source. Optionally, the registered data source may be indexed, but otherwise, it is an $\mathcal{O}(1)$ operation. The database can be configured to work with almost any data shape. Once a query is submitted, it is first parsed by an appropriate language plugin into an abstract syntax tree, which is then converted into a logical plan made of unified algebra operators. The plan is optimized by the DortDB optimizer and executed. The DortDB executor

uses an internal representation of the queried data, so it must be serialized by a language plugin serializer before being returned to the user.

3.3 Limitations

While DortDB offers a range of capabilities and has been designed with flexibility and performance in mind, it is important to acknowledge that it is not without limitations. The following section identifies the key weaknesses of the framework. We outline them now in order to avoid setting unrealistic expectations for the rest of the thesis.

3.3.1 Optimizer

DortDB currently does not collect any statistics on registered data sources. It is therefore not viable to implement cost-based optimization. The current rule-based optimizer cannot, for example, reorder joins.

3.3.2 Data Modification

DortDB is currently a read-only query framework. While this could change in the future, currently, the only way to insert or update data is to interact with the registered data sources directly from JavaScript.

3.3.3 Parallelism

JavaScript is a single-threaded language. There is limited support for parallelism with Workers, but Workers cannot (with some exceptions) share data with each other or with the main thread. If a data structure is to be passed to a Worker, it may be necessary to copy it rather than simply transfer a reference. For these reasons, DortDB runs in a single thread only. A possible limited solution would be to explicitly shard registered data sources and let separate Workers work on separate shards. This is left as future work.

3.3.4 Language Limitations

None of the currently implemented languages is implemented fully. In addition to data modification operations, there is no support for, e.g., SQL `GROUP BY ROLLUP` or XQuery type validation. Some of these features are planned for future work; others are not planned at all, especially type checking, as it conflicts with DortDB's schema-agnostic design. We do, however, believe that the current DortDB languages are implemented sufficiently for most uses.

4 Implemented Languages

As DortDB aims to support any query language, we have implemented three initial languages, one for each major data model. None of the languages was implemented completely. Given that DortDB does not aim to support data definition and manipulation (at least initially), the languages were trimmed to their data selection subset. Additionally, most of the typing and schema-validation features were removed, as they couple the language too closely with the underlying data representation.

4.1 SQL

SQL (Structured Query Language) is a relational query language based on relational algebra and relational calculus. It became an ISO standard in 1987; the latest version was published in 2023 [19]. The language operates in terms of tables (relations). Each table has a schema consisting of a name and a set of attributes. Tables contain rows (tuples), whose values are called columns and correspond to the schema attributes. Important SQL operations include relational joins and data aggregation.

The DortDB SQL language is based on the PostgreSQL SQL flavor.¹ It also includes several of the PostgreSQL JSON access operators.

4.1.1 Notable Features

PostgreSQL includes several features that are not usual in other SQL flavors. Among them are the following:

Lateral Joins

Lateral joins allow joining of correlated subqueries. In other words, it is possible for the joined subquery to refer to previous tables, and thus to be re-evaluated for different outer contexts.

Program 8 Example of a lateral join.

```
SELECT t1.attr1, s.attr2 FROM t1
JOIN LATERAL (
  SELECT t2.attr2 FROM t2
  -- the WHERE clause refers to the outer context
  -- the subquery is reevaluated for each t1 row
  WHERE t2.attr3 + t1.attr4 > t1.attr5
) AS s
```

¹<https://www.postgresql.org/>

DISTINCT ON

The `DISTINCT` modifier filters out duplicate values. PostgreSQL allows customizing this behavior.

Program 9 Example of a `DISTINCT` query filtering only specific expressions.

```
SELECT DISTINCT ON (attr1, attr2 % 10) attr1, attr2, attr3
FROM t1
```

Complex Aggregate Calls

The arguments of aggregate calls, such as `count` and `sum`, may be filtered or sorted.

Program 10 Example of filtered and ordered aggregate calls.

```
SELECT
  count(DISTINCT attr1) AS distinct_count,
  count(attr1) FILTER (WHERE attr1 % 2 = 0) AS even_count,
  collect(attr1 ORDER BY attr1) AS ordered_array
FROM t1
```

4.1.2 Schema of Data Sources

SQL expects tables to have a clearly defined schema. This enables users to write queries that would otherwise be ambiguous.

```
-- which tables are a, b, or c from?
SELECT a, b, c
-- what columns are used in the natural join?
FROM t1 NATURAL JOIN t2
```

DortDB data sources, however, are schemaless by design. Therefore, the allowed SQL queries are subject to some restrictions. It must be clear which tables specific columns originate from. Non-qualified column names are only allowed when there is a single, non-joined data source. Natural joins are disabled. All `FROM` clause subqueries must have an alias. Selecting all attributes using the asterisk `*` is also disabled.

Non-qualified attributes are always interpreted as belonging to the current `FROM` clause table. This is not always desirable, for example, when inserting SQL subqueries into languages that do not prefix identifiers (e.g., Cypher). In this case, the attribute can be prefixed by the `nonlocal` schema. DortDB will not try to assign the attribute to the current table schema; instead, it will refer to where the attribute was defined in an outer context.

Program 11 An example of the `nonlocal` schema. If not for the schema, the `attr1` attribute in the subquery would refer to the current table `t2`. Instead, the closest other instance of `attr1` is looked up. The usage of `nonlocal` here is completely unnecessary, but when mixing languages, it may enable otherwise impossible queries.

```
SELECT
  /*t1.*/ attr1
FROM t1
WHERE (
  SELECT
    /*t2.*/ attr2
  FROM t2
  WHERE /*t2.*/ attr3 < nonlocal.attr1 /*t1.attr1*/
) = 1
```

4.1.3 Not Implemented Features

Besides data types, there are features that are not currently part of the DortDB SQL. The parser recognizes them, but they have not yet been implemented. The functionality will be added at some point in the future. This includes window functions or special `GROUP BY` features such as `ROLLUP`.

4.2 XQuery

XQuery² is a functional programming language designed for querying structured and semi-structured XML data. It contains a superset of XPath,³ an expression language for querying XML. XPath operates on sequences of values. Sequences cannot be nested; a sequence of sequences is interpreted as if it were flattened. Conversely, the values can be interpreted as sequences of length 1. Null values are treated as empty sequences. XQuery adds SQL-like FLWOR expressions, named after some of the available clauses: For, Let, Where, Order by, and Return. Within FLWOR expressions, data is streamed as sequences of named tuples of values, but a FLWOR expression may return only sequences of opaque items. XQuery cannot accept or produce tuples.

Program 12 Example of an XQuery query combining XPath and FLWOR expressions.

```
for $x in $source/foo (: $source/foo is an XPath expression :)
let $y := $x/bar (: $y is a sequence :)
(: data is passed between clauses
   as tuples with attributes x and y :)
return $y (: the result of the FLWOR is a flattened sequence :)
```

Aggregation in XQuery is done by passing a sequence into a function. For example, summing numbers from 1 to 10 can be done like so:

²<https://www.w3.org/TR/2014/REC-xquery-30-20140408>

³<https://www.w3.org/TR/xpath-31/>

```
sum(1 to 10)
```

XPath defines the static and dynamic context of an expression. The static context consists of information available during the expression's static analysis, such as statically known namespaces and in-scope variables. The dynamic context contains the current context item (referenced by a single dot or by the special variable `$fs:dot`), the current context position (referenced by the function `fn:position()` or by the special variable `$fs:position`), and the current context size (referenced by the function `fn:last()` or by the special variable `$fs:last`).

Program 13 Example of the XQuery static and dynamic context.

```
declare namespace foo = "http://example.org";
(: the static context now includes the foo namespace :)

(5 to 10)[
  (:
    the predicate can access the current item
    (numbers from 5 to 10),
    the current position (numbers from 1 to 5),
    and the current context size (number 5)
  :)
  . mod 2 eq 1
]
(: the result is the sequence (5,7,9) :)
```

DortDB XQuery is a subset of XQuery 3.0.

4.2.1 Not Implemented Features

The DortDB XQuery implementation does not include any typing except `cast` expressions. This means that the following is not (and is not ever planned to be) available:

- `as` keyword
- `typeswitch` expressions
- `instance of` expressions
- `treat` expressions
- `castable` expressions
- `validate` expressions
- any function types except `function(*)`
- function references containing arity information (`fname#arity`)

The following is also not implemented, but may be added in the future:

- `try catch` expressions
- `pragma` expressions
- `window FLWOR` clause
- inline functions
- `%private` and `%public` modifiers
- any module prolog except namespace definition

4.3 Cypher

Cypher is a declarative graph query language developed for the Neo4j database.⁴ The DortDB Cypher language is based on openCypher, which is an open source specification of Cypher. Since the release of GQL (Graph Query Language) by ISO⁵ in April 2024, the aim of the openCypher project is to converge towards GQL and ease the transition of graph databases towards the ISO standard.

Cypher operates on labeled property graphs, a graph data model in which both nodes and relationships may contain properties. Furthermore, nodes may have zero or more labels, while each relationship has exactly one type. Relationships are always directed.

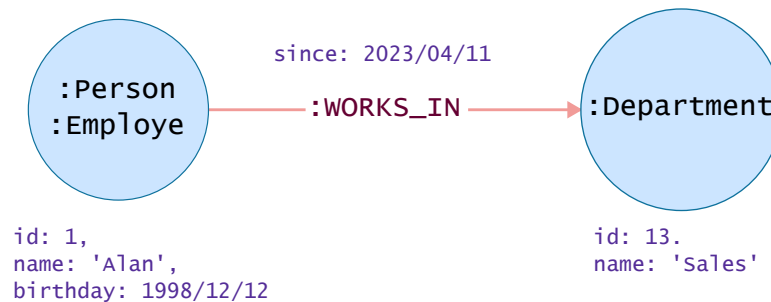


Figure 4.1 Labeled property graph

⁴<https://neo4j.com/docs/cypher-manual/25/introduction/>

⁵<https://www.iso.org/standard/76120.html>

Cypher provides a visual way of matching patterns and relationships using ASCII art. It is possible to match recursive paths. A matched path cannot contain duplicate relationships, but each node may be visited multiple times.

Program 14 Example of Cypher pattern matching.

```
// match each neighbor of Alan, regardless of direction  
// the neighbors will be stored in the 'neighbor' variable  
MATCH (:Person {name: "Alan"})-[]-(neighbor)  
  
// match Alan's friends of friends, up to three hops far,  
// and then find their employers  
// the p variable will contain a list of relationships  
MATCH ({name: "Alan"})-[p:KNOWS *1..3]->(foaf)  
      <-[:EMPLOYS]-(employer)  
  
// the recursive path may match patterns like  
// (alan)-[x]->(alan)-[y]->()  
// but it cannot match the same relationship twice  
// (alan)-[x]->(alan)-[x]->()
```

It is then possible to process the matched values similarly to SQL.

Program 15 Example of processing the matched results.

```
MATCH (x)-[]->(y)  
WHERE y.age < 15  
RETURN x.name, y.age  
ORDER BY x.name  
SKIP 3
```

4.3.1 Cypher in DortDB

DortDB can parse the whole openCypher 9 grammar. There is, however, no support for data modification or data creation clauses, such as **CREATE** or **SET**. Furthermore, only a subset of the standard library functions and operators is currently implemented. The users can provide their own implementations as user-defined functions, though.

5 Formalization of Cross-Model Querying

In DortDB, each language is translated into the same formal algebra. This decouples specific languages from the core framework. It also allows us to combine snippets from different languages and process them as a single query. This enables the user to separate their query into logical parts and express each in a language they find the most intuitive. It also allows the formation of multimodel queries using the languages initially developed for each model. In this chapter, we will define and explore the algebraic model, as well as what the multilanguage queries look like for the actual user.

5.1 Language switching

Our approach builds upon Virtuoso’s SPASQL to enable the embedding of subqueries from any language into any other language. It is necessary to establish clear rules for the language-to-language interfaces. In some cases, the transition is intuitive. For example, a SQL query that selects attributes from a Cypher subquery. The Cypher subquery returns a set of tuples, just like a SQL subquery would. On the other hand, SQL may sometimes require a subquery to produce only a single value (see Program 16). Some languages, such as XPath, do not operate on tuples. Our solution is to separate the subqueries into those that produce tuples and those that produce opaque items. Each language is then responsible for converting the results of these subqueries as necessary.

Program 16 Example of two SQL subqueries. The first subquery produces tuples, the second produces only a single value.

```
SELECT attr1, attr2 FROM (  
    -- subquery  
);  
  
SELECT (SELECT count(*) FROM table1) AS total, count(*)  
FROM table1  
GROUP BY item_id;
```

The language switch subqueries are demarcated by the **LANG** keyword on one side and either by a scope exit (a closing bracket or parenthesis) or by the **LANG EXIT** keywords on the other. The nested language can access variables from the parent scope. See Program 17. In the case of languages such as SQL, the attributes used by the nested language are used to infer the schema of referenced relations. See Figure 5.1 for an example of how the SQL parser uses information provided by the nested XQuery and SQL parsers to infer the schema of the `foo` relation. The logical operator tree is partially optimized for better readability. Diagrams like this are explained in Section Unified Algebra. For now, the important thing is

that the grey brackets in some nodes denote the tuple schema of the corresponding operator.

Program 17 Cypher nested into SQL refers to the outer relation `people`.

```
SELECT id, ARRAY(
  LANG cypher
  MATCH (:person {id: people.id})-[:KNOWS]->(friend)
  RETURN friend
) AS friends
FROM people
```

5.2 Unified Object Representation

At the core of the query pipeline, the system adopts a uniform internal representation that abstracts over the heterogeneity of the underlying data models. Every plan operator is viewed as producing one of two fundamental kinds of structures: *tuples* or *items*. A tuple is a collection with named attributes, analogous to a relational tuple, where each attribute consists of a sequence of identifiers (for example, a table name and a column name in SQL). An item source, by contrast, contributes opaque items for which the system assumes no intrinsic structure, making no commitments about shape or schema.

Definition 1 (Item). *An item is an element of an arbitrary, uninterpreted domain $\mathcal{V}$. No structural commitments (e.g., schema, attribute decomposition) are made about elements of $\mathcal{V}$. Items are treated as opaque values by the system.*

Definition 2 (Attribute Name). *An attribute name is a finite, non-empty sequence of identifiers $\langle id_1, id_2, \dots, id_k \rangle$, where each id_i is a non-empty string over some fixed alphabet Σ . The set of all attribute names is denoted $\mathcal{A}$.*

Definition 3 (Tuple). *A tuple over a finite attribute set $S \subseteq \mathcal{A}$ is a total function*

$$t : S \rightarrow \mathcal{V}$$

The pair of set S and attribute ordering $<$ is called the schema of t , written $\text{schema}(t)$. If a plan operator produces tuples, all of its tuples have an identical schema. We can therefore introduce operator schema, written $\text{schema}(\text{Op})$. The set of all tuples over all finite $S \subseteq \mathcal{A}$ is denoted $\mathcal{T}$.

Definition 4 (Tuple Concatenation). *Some tuple operators produce tuples that are combinations of the operator source tuples. This operation on tuples is called Tuple Concatenation, denoted $\oplus$.*

$$\oplus : \mathcal{T} \times \mathcal{T} \rightarrow \mathcal{T}$$

$$t, u \in \mathcal{T} : \text{schema}(t) \oplus \text{schema}(u) = (A_t \cup A_u, <_{tu})$$

Where $tu_{<}$ is the total order on $A_t \cup A_u$, such that:

```
SELECT foo.first, (
  LANG xquery
  for $x in $xs
  return $foo:second + (
    LANG sql
    SELECT foo.third FROM bar
  )
) AS nested
FROM foo
```

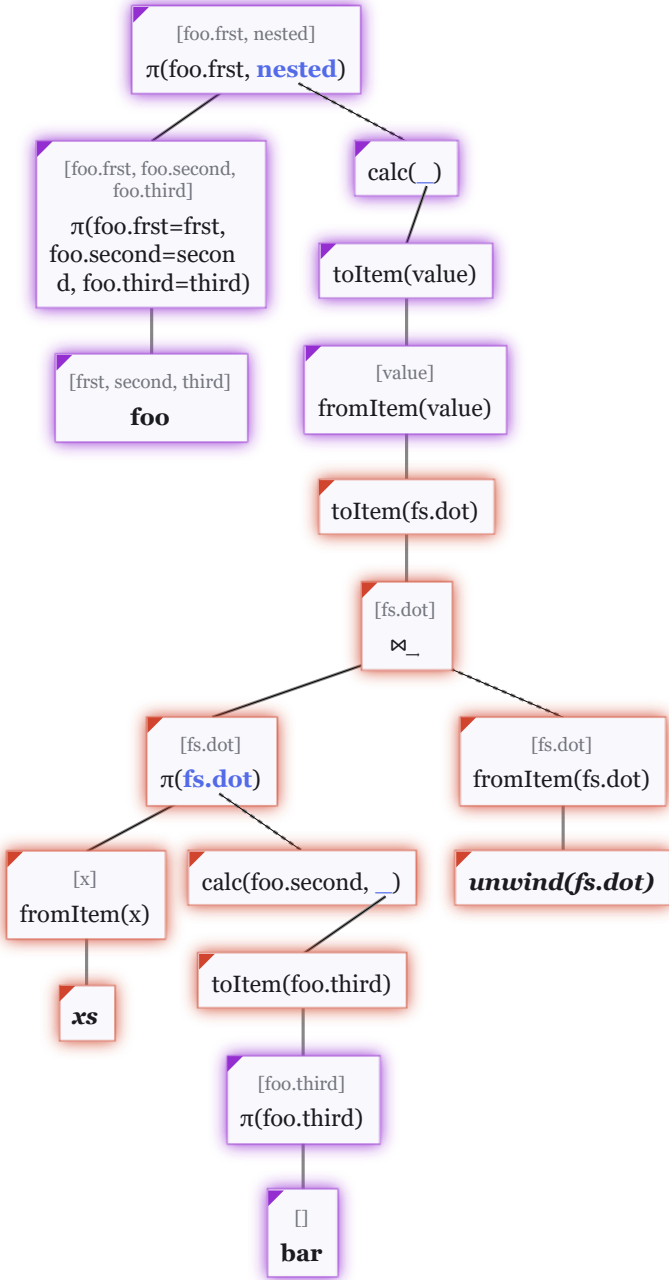


Figure 5.1 SQL schema inference example

- $\forall x, y \in A_t : x <_{tu} y \iff x <_t y$
- $\forall x, y \in A_u \setminus A_t : x <_{tu} y \iff x <_u y$
- $\forall x \in A_t, y \in A_u \setminus A_t : x <_{tu} y$

$$t, u \in \mathcal{T} : t \oplus u = \{(k, v) \in t \mid k \notin \text{dom}(u)\} \cup u$$

Definition 5 (Stream). A stream is a possibly infinite ordered sequence of values of type X , denoted $\text{Stream}(X)$. Furthermore, $\text{Stream}_1(X)$ is a stream that produces exactly one value. Similarly, $\text{Stream}_{01}(X)$ is a stream that produces exactly one or no values. A stream producing a specific sequence of values is denoted

$$\langle a, b, c \rangle \text{ or perhaps } \langle f(x) \mid x \in X \rangle$$

The set of all streams is denoted $\mathcal{I}$.

5.2.1 Comparison with Nested Relations

The nested relational model [20] offers a strong formal framework. For our purposes, however, it is too rigid. To properly use it, we would need to know the complete schema of the items in the registered data sources beforehand, which goes against DortDB’s schema-agnostic approach. Nested relations also cannot easily cover semi-structured and unstructured data, unlike our opaque items. The tuples we use, however, are essentially relational tuples, so there is overlap.

5.2.2 Comparison with Property Graph Model

Both models are viable to describe multimodel data. In fact, anything can be mapped as a graph. In our case, we chose the tuple and item model because of existing related work [21] that we felt could be well adapted for our use.

5.3 Unified Algebra

Queries are parsed into a logical operator tree, where each node corresponds to an operator from *unified algebra*. The unified algebra covers most of the expected query operations. It is, however, possible for any language to extend the algebra and to define its own operators. This way, it is possible to cover any future languages regardless of their model. This process is more closely explained in Section Extensibility.

The algebra is based on algebras for XQuery [21], graph paths [22] and nested relational algebra [20]. We divide the operators into two main groups. *Tuple operators* operate on streams of named tuples, similarly to relational algebra. *Item operators* operate on streams of opaque items. Depending on their arguments, some operators may be considered either item or tuple operators. One such example is the `Limit` operator.

5.3.1 Operator Context

Some operators behave differently based on the outer context of their containing operators. This context is denoted as $\Gamma \in \mathcal{T}$. The context is a concatenation of the tuples produced by some upstream operators, similar to how programming language variable scopes work. In general, Γ of an operator contains information that is available when the operator was instantiated, as well as the latest tuples produced by its direct input operators (if the input operators indeed produce tuples).

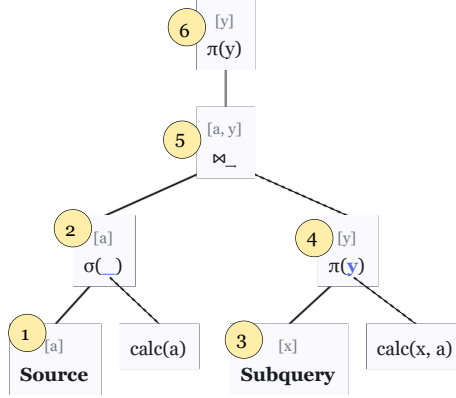


Figure 5.2 Operator context example.

For example, see Figure 5.2. The operators receive the following context:

1. The **TupleSource** receives no context.
2. The **Selection** and its **Calculation** receive the last tuple produced by the **TupleSource Source**.
3. The **TupleSource** receives the latest tuple produced by the **Selection**.
4. The **Projection** and its **Calculation** receive the concatenation of the latest tuples produced by the **Selection** and the **TupleSource Subquery**.
5. The **ProjectionConcat** receives the latest tuple produced by the **Selection**.
6. The **Projection** receives the latest tuple produced by the **ProjectionConcat**.

Definition 6 (Operator Instantiation). *Usually, operators are instantiated when their containing operator is instantiated. For example, **CartesianProduct** creates its left and right source streams once and keeps them for its whole lifetime. Conversely, some operators are recreated multiple times in response to context changes. Formally,*

$$\text{stream} \subseteq \mathcal{I}, \text{inst}_{\text{stream}} : \mathcal{T} \rightarrow \text{stream}$$

describes an operator that is instantiated based on the provided tuple context.

Definition 7 (Vertical input operators). Vertical input operators of an operator, denoted $\text{vertical}(\text{Op})$, are those input operators that are instantiated only once, when their containing operator is instantiated. As an example, consider the **source** of a **Selection**.

Definition 8 (Horizontal input operators). Horizontal input operators of an operator, denoted $\text{horizontal}(\text{Op})$, are those input operators that are instantiated multiple times during the lifetime of their containing operator. As an example, consider a subquery in the **condition** of a **Selection**.

Algebraic Operator	Operator Signature	Operator Schema
Calculation Intermediaries AggregateCall Conditional FnCall Literal Quantifier	$\text{agg}(\text{args})$ $\text{cond}(\text{expr}, \text{whenthens}, \text{default})$ $\text{fn}(\text{impl}, \text{args})$ $\text{literal}(\text{value})$ $\text{quant}(\text{type}, \text{query})$	
Other Calculation ItemSource ItemFnSource MapToItem	$\text{calc}(\text{impl}, \text{args})$ name $\text{name}(\text{impl}, \text{params})$ $\text{toItem}(\text{key}, \text{source})$	

Table 5.1 Operators of the Unified Algebra: Item Operators

Algebraic Operator	Operator Signature	Operator Schema
SPJ		
CartesianProduct	$\times(\text{left}, \text{right})$	$\text{left} \oplus \text{right}$
Join	$\bowtie(\text{left}, \text{right}, \text{leftOuter}, \text{rightOuter}, \text{conditions})$	$\text{left} \oplus \text{right}$
Projection	$\pi(\text{attrs}, \text{source})$	attrs
ProjectionConcat	$\bowtie\rightarrow(\text{mapping}, \text{outer}, \text{source})$	$\text{source} \oplus \text{mapping}$
ProjectionIndex	$\text{index}(\text{key}, \text{source})$	$\text{source} \oplus (\text{key})$
Selection	$\sigma(\text{expression}, \text{source})$	source
Other		
Distinct	$\delta(\text{attributes}, \text{source})$	source
GroupBy	$\gamma(\text{keys}, \text{aggs}, \text{source})$	$\text{keys} \oplus \text{aggs}$
MapFromItem	$\text{fromItem}(\text{key}, \text{source})$	(key)
OrderBy	$\tau(\text{orders}, \text{source})$	source
Recursion	$\phi(\text{min}, \text{max}, \text{condition}, \text{source})$	source
TupleSource	name	unknown
TupleFnSource	name (impl, params)	possibly unknown

Table 5.2 Operators of the Unified Algebra: Tuple Operators – SPJ and Other

Algebraic Operator	Operator Signature	Operator Schema
ProjectionSize	$\text{size}(\text{key}, \text{source})$	$\text{source} \oplus (\text{key})$
TreeJoin	$\text{treeJoin}(\text{expr}, \text{source})$	$\text{source} \oplus (\text{fs.dot}, \text{fs.position}, \text{fs.last})$

Table 5.3 Operators of the Unified Algebra: Tuple Operators – XQuery

Algebraic Operator	Operator Signature	Operator Schema
NullSource	$\square$	
Limit	$\text{limit}(\text{offset}, \text{limit}, \text{source})$	
Set		
Union	$\cup(\text{left}, \text{right})$	
Intersection	$\cap(\text{left}, \text{right})$	
Difference	$\setminus(\text{left}, \text{right})$	

Table 5.4 Operators of the Unified Algebra: Universal Operators

Algebraic Operator	Input Domain	Output Domain	Semantics
Calculation Intermediaries $\text{cond}(\text{expr}, \text{whenthens}, \text{default})$ $\text{fn}(\text{impl}, \text{args})$	$\text{expr: Stream}_{01}(\mathcal{V}),$ $\text{whenthens: Seq}((\text{Stream}_{01}(\mathcal{V}), \text{Stream}_{01}(\mathcal{V}))),$ $\text{default: Stream}_{01}(\mathcal{V})$ $\text{impl: Seq}(\text{Stream}(\mathcal{V})) \rightarrow \text{Stream}_1(\mathcal{V}),$ $\text{args: Seq}(\text{Stream}(\mathcal{V}))$	$\text{Stream}_1(\mathcal{V})$ $\text{Stream}_1(\mathcal{V})$	$\text{cond}(e, w, d) = \begin{cases} \langle w_{i,1} \rangle \text{ where } i \text{ is the first index} \\ \text{with } w_{i,0} = e, \text{ else } d, & \text{if } e \text{ is provided} \\ \langle w_{i,1} \rangle \text{ where } i \text{ is the first index} \\ \text{with } w_{i,0} = \langle \text{true} \rangle, \text{ else } d, & \text{otherwise} \end{cases}$ $\text{fn}(i, A) = \langle i(A) \rangle$
Other $\text{calc}(\text{impl}, \text{args})$ $\text{name}(\text{impl}, \text{params})$ $\text{toItem}(\text{key}, \text{source})$	$\text{impl: Seq}(\text{Stream}(\mathcal{V})) \rightarrow \text{Stream}_1(\mathcal{V}),$ $\text{args: Seq}(\text{Stream}(\mathcal{V}))$ $\text{impl: Seq}(\text{Stream}_1(\mathcal{V})) \rightarrow \text{Stream}(\mathcal{V}),$ $\text{params: Seq}(\text{Stream}_1(\mathcal{V}))$ $\text{key: } \mathcal{A},$ $\text{source: Stream}(\mathcal{T}), \text{ where }$ $\text{key} \in \text{schema}(\text{source})$	$\text{Stream}_1(\mathcal{V})$ $\text{Stream}(\mathcal{V})$ $\text{Stream}(\mathcal{V})$	$\text{calc}(i, A) = \langle i(A) \rangle$ $\text{name}(i, A) = i(A)$ $\text{toItem}(k, S) = \langle t.k \mid t \in S \rangle$

Table 5.5 Selected Operators of the Unified Algebra – formal: Item Operators

Algebraic Operator	Input Domain	Output Domain	Semantics
$\times(\text{left}, \text{right})$	left: $\text{Stream}(\mathcal{T})$, right: $\text{Stream}(\mathcal{T})$	$\text{Stream}(\mathcal{T})$	$\times(L, R) = \langle l \oplus r \mid l \in L, r \in R \rangle$
$\bowtie(\text{left}, \text{right}, \text{leftOuter}, \text{rightOuter}, \text{conditions})$	left: $\text{Stream}(\mathcal{T})$, right: $\text{Stream}(\mathcal{T})$, leftOuter: Boolean, rightOuter: Boolean, conditions: $\text{Set}(\text{inst}_{\text{Stream}_1(\mathcal{V})})$	$\text{Stream}(\mathcal{T})$	$\bowtie(L, R, \text{false}, \text{false}, C, \Gamma) = \langle l \oplus r \mid l \in L, r \in R, \forall c \in C : c(\Gamma \cup l \cup r) = \langle \text{true} \rangle \rangle$ If leftOuter (or rightOuter) is true, the left (or right) unmatched tuples are also returned, padded with null values.
$\pi(\text{attrs}, \text{source})$	attrs: $\text{Seq}(\text{inst}_{\text{Stream}_1(\mathcal{V})} \times \mathcal{A})$, source: $\text{Stream}(\mathcal{T})$	$\text{Stream}(\mathcal{T})$	$\pi(A, S, \Gamma) = \langle \{a \mapsto c(\Gamma \oplus s) \mid (c, a) \in A\} \mid s \in S \rangle$
$\overset{\bowtie}{\rightarrow}(\text{mapping}, \text{outer}, \text{source})$	mapping: $\text{inst}_{\text{Stream}(\mathcal{T})}$, outer: Boolean, source: $\text{Stream}(\mathcal{T})$	$\text{Stream}(\mathcal{T})$	$\overset{\bowtie}{\rightarrow}(M, \text{false}, S, \Gamma) = \langle s \oplus m \mid m \in M(\Gamma \oplus s), s \in S \rangle$ If outer is true, tuples from source for which there are no corresponding mapping tuples are also returned, padded with null values.
$\text{index}(\text{key}, \text{source})$	key: $\mathcal{A}$, source: $\text{Stream}(\mathcal{T})$	$\text{Stream}(\mathcal{T})$	$\text{index}(k, S) = \langle s_i \oplus \{k \mapsto i\} \mid s_i \in S \rangle$
$\sigma(\text{expr}, \text{source})$	expr: $\text{inst}_{\text{Stream}_1(\mathcal{V})}$, source: $\text{Stream}(\mathcal{T})$	$\text{Stream}(\mathcal{T})$	$\sigma(E, S, \Gamma) = \langle s \mid s \in S, E(\Gamma \oplus s) = \langle \text{true} \rangle \rangle$

Table 5.6 Selected Operators of the Unified Algebra – formal: Tuple Operators (SPJ)

Algebraic Operator	Input Domain	Output Domain	Semantics
$\gamma(\text{keys}, \text{aggs}, \text{source})$	keys: $\text{Seq}(\text{inst}_{\text{Stream}_1(\mathcal{V})} \times \mathcal{A})$, aggs: $\text{Seq}((\text{Stream}(\mathcal{T}) \rightarrow \mathcal{V}) \times \mathcal{A})$, source: $\text{Stream}(\mathcal{T})$	$\text{Stream}(\mathcal{T})$	$\text{group}(K, V, S, \Gamma) = \langle s \mid s \in S, \forall i : K_i(\Gamma \oplus s) = V_i \rangle$ $\gamma(K, A, S, \Gamma) = \langle$ $\quad \{a \mapsto k(\Gamma \oplus s) \mid (k, a) \in K\} \oplus$ $\quad \{a \mapsto f(\text{group}($ $\quad \quad K, \langle k(\Gamma \oplus s) \mid (k, _) \in K \rangle, S, \Gamma$ $\quad \quad)) \mid (f, a) \in A\}$ $\quad \mid s \in S$ $\quad \rangle_{\text{unique}}$
$\text{fromItem}(\text{key}, \text{source})$	key: $\mathcal{A}$, source: $\text{Stream}(\mathcal{V})$	$\text{Stream}(\mathcal{T})$	$\text{fromItem}(k, S) = \langle \{k \mapsto s\} \mid s \in S \rangle$
$\phi(\text{min}, \text{max}, \text{condition}, \text{source})$	min: $\mathbb{N}$, max: $\mathbb{N}$, condition: $\text{inst}_{\text{Stream}_1(\mathcal{V})}$, source: $\text{Stream}(\mathcal{T})$	$\text{Stream}(\mathcal{T})$	$\phi(0, 1, C, S, \Gamma) = \langle \{a \mapsto \langle s.a \rangle \mid a \in s\} \mid s \in S \rangle$ $\phi(0, \text{max}, C, S, \Gamma) = \langle$ $\quad \{a \mapsto \text{acc}.a \cup \langle \text{curr}.a \rangle \mid a \in \text{acc}\}$ $\quad \mid \text{curr} \in S,$ $\quad \quad \text{acc} \in \phi(0, \text{max} - 1, C, S, \Gamma),$ $\quad \quad C(\Gamma \oplus \text{zip}(\text{curr}, \text{acc})) = \langle \text{true} \rangle$ $\quad \rangle$
$\text{name}(\text{impl}, \text{params})$	impl: $\text{Seq}(\text{Stream}_1(\mathcal{V})) \rightarrow \text{Stream}(\mathcal{T})$, params: $\text{Seq}(\text{Stream}_1(\mathcal{V}))$	$\text{Stream}(\mathcal{T})$	$\phi(\text{min}, \text{max}, C, S, \Gamma) = \langle p \mid p \in \phi(0, \text{max}, C, S, \Gamma), \forall a \in p : p.a \geq \text{min} \rangle$ $\text{name}(i, A) = i(A)$

Table 5.7 Selected Operators of the Unified Algebra – formal: Tuple Operators – Other

Algebraic Operator	Input Domain	Output Domain	Semantics
size(key, source)	key: $\mathcal{A}$, source: Stream($\mathcal{T}$)	Stream($\mathcal{T}$)	$\text{size}(k, S) = \langle s \oplus \{k \mapsto S \} \mid s \in S \rangle$
treeJoin(expr, source)	expr: $\text{inst}_{\text{Stream}(\mathcal{V})}$, source: Stream($\mathcal{T}$)	Stream($\mathcal{T}$)	$\text{treeJoin}(E, S, \Gamma) = \langle$ $s_i \oplus \{$ $\langle \text{fs}, \text{dot} \rangle \mapsto e,$ $\langle \text{fs}, \text{position} \rangle \mapsto i,$ $\langle \text{fs}, \text{last} \rangle \mapsto \text{treeJoin}(E, S, \Gamma) $ $\} \mid s_i \in S, e \in E(\Gamma \oplus s_i)$ $\rangle$

Table 5.8 Selected Operators of the Unified Algebra – formal: Tuple Operators – XQuery

Algebraic Operator	Input Domain	Output Domain	Semantics
$\square$ limit(offset, limit, source)	offset: $\mathbb{N}$, limit: $\mathbb{N}$, source: Stream($\mathcal{V} \cup \mathcal{T}$)	Stream ₁ ($\mathcal{V} \cup \mathcal{T}$) Stream($\mathcal{V} \cup \mathcal{T}$)	$\square = \langle \emptyset \rangle$ $\text{limit}(o, l, S) = \langle s_i \mid s_i \in S, o < i \leq o + l \rangle$
Set $\cup(\text{left}, \text{right})$ $\cap(\text{left}, \text{right})$ $\setminus(\text{left}, \text{right})$	left: Stream($\mathcal{V} \cup \mathcal{T}$), right: Stream($\mathcal{V} \cup \mathcal{T}$) left: Stream($\mathcal{V} \cup \mathcal{T}$), right: Stream($\mathcal{V} \cup \mathcal{T}$) left: Stream($\mathcal{V} \cup \mathcal{T}$), right: Stream($\mathcal{V} \cup \mathcal{T}$)	Stream($\mathcal{V} \cup \mathcal{T}$) Stream($\mathcal{V} \cup \mathcal{T}$) Stream($\mathcal{V} \cup \mathcal{T}$)	$\cup(L, R) = L \cup R$ $\cap(L, R) = \langle x \mid x \in R \wedge x \in L \rangle$ $\setminus(L, R) = \langle x \mid x \in L \wedge x \notin R \rangle$

Table 5.9 Selected Operators of the Unified Algebra – formal: Universal Operators

5.3.2 Visualization

The DortDB GUI includes query plan visualization. The plans are visualized as a tree where each node corresponds to an operator. The tree's root is the final query; the leaves are individual data sources. Each node may be colored according to the language it originated in.¹ Tuple operator nodes include the operator schema. The edges are either solid or dashed. Solid edges mean that the child operator is created only once at the same time as its parent. Dashed edges indicate a child operator that is dynamically recreated multiple times during its parent's lifetime.

The GUI source code is available as an appendix to this work. The GUI is also available online at <https://filipjezek.github.io/dortdb>. The reader is encouraged to try things out for themselves.

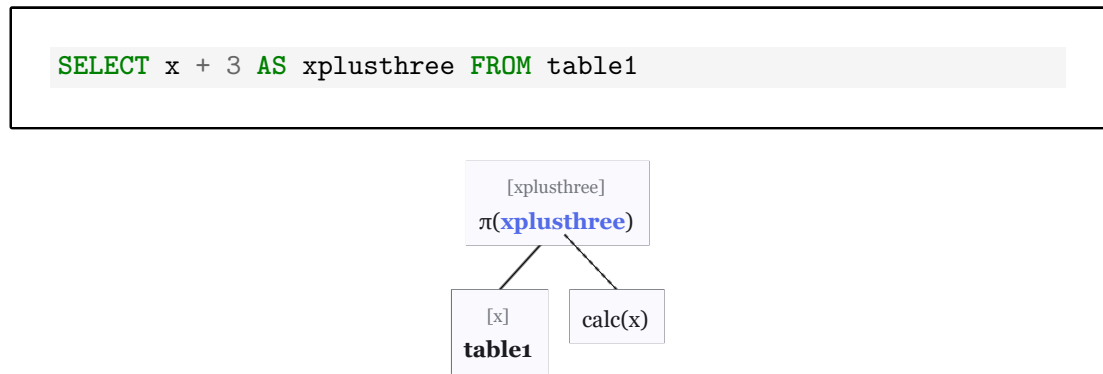


Figure 5.3 The different edge types: TupleSource **table1** is created once while the Calculation is reevaluated for each source row. The Calculation edge points directly to its representative attribute **xplusthree** shown in blue color.

5.3.3 Item Operators

Most of the item operators are calculation intermediaries. They do not participate in the final query plan. Instead, they serve as building blocks for the Calculation operator. The Calculation operator represents any calculated value, such as Projection attributes or Selection condition. The Calculation arguments can be either attribute identifiers or other (non-calculation-intermediate) plan operators. It is possible to express subqueries using a Calculation with e.g. a Projection as its argument. In such cases, the Calculation tracks whether the subquery is supposed to produce at most one or multiple values. The Quantifier operator represents a SQL-style quantified query, for example `x > ALL(SELECT y FROM t)`. The AggregateCall operator represents the result of an aggregation applied to an earlier GroupBy operator.

The remaining full-fledged item operators are data sources, and the MapToItem operator, which converts tuples to items. It selects the **key** attribute from each tuple. If the **key** is the *allAttrs* symbol `*`, the whole tuple is treated as the new item.

¹In this thesis, we will color nodes in diagrams with multiple languages.

```

SELECT x FROM table1
WHERE x < (
  SELECT y FROM table2
)

```

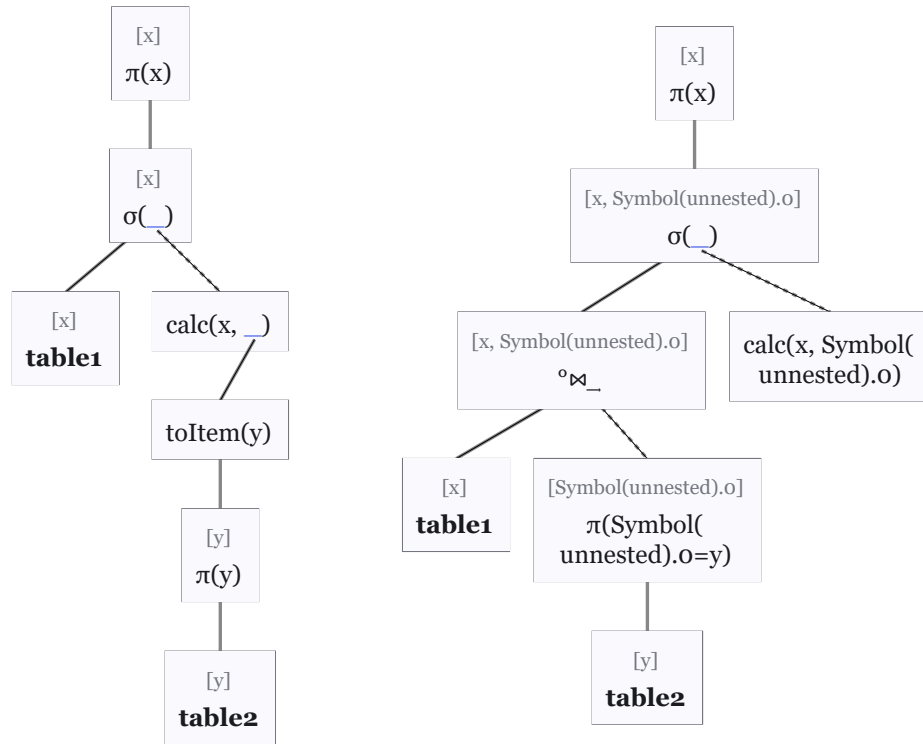


Figure 5.4 A subquery can be expressed as a Calculation with a Projection as an argument. Alternatively, the optimizer could convert the nested Projection to an outer ProjectionConcat. Note that this is not yet fully optimized. Because the nested Projection does not depend on **table1** context, the ProjectionConcat can be further converted to a left outer Join.

5.3.4 Tuple Operators

Tuple operators include the well-known **Selection**, **Projection**, **Join**, **CartesianProduct**, and **OrderBy**. **Join** may include multiple conditions; in that case, all must be satisfied for the two source tuples to be joined. Multiple separated conditions make it easier for the optimizer to recognize certain opportunities. It is also possible to not specify any condition at all and to use the **Join** operator solely for its **leftOuter** or **rightOuter** properties.

ProjectionIndex adds a new attribute to each tuple that tracks its ordinal number. **ProjectionConcat** could be also called **depend-join**. It is similar to **Join**, in that it joins **source** tuples to **mapping** tuples. For each **source** tuple, the **mapping** stream is reevaluated, and each of its tuples is joined to the **source** tuple. The **mapping** may depend on the context of the **source** stream. In terms of SQL, the **ProjectionConcat** operator can be utilized in correlated subqueries in the **SELECT** or **WHERE** clause, or in **LATERAL JOIN**.

The **Distinct** operator filters out duplicate tuples. Duplicates are determined based on **attributes**. Similarly to **MapToItem**, **attributes** may be the *allAttrs symbol* *. The **MapFromItem** operator converts an item to a tuple with one attribute named **key**, which contains the original item. It is not possible to interpret the original item as a tuple.

GroupBy groups **source** values into partitions based on **keys**. Each partition is then processed by **AggregateCall** operators in **aggs**. Each **AggregateCall** may contain additional operators that preprocess the partition stream before aggregation, for example, when the aggregate requires certain ordering or filtering.

The **Recursion** operator is a self-join repeated up to **max** times.²³

Finally, there are two operators used only by XQuery. They serve as examples of the algebra's extensibility and are not included in the core framework but are provided by the XQuery language package. The **ProjectionSize** operator adds a new attribute to each tuple that contains the cardinality of the tuple stream. **TreeJoin** is a utility operator, which combines **ProjectionConcat**, **ProjectionIndex** and **ProjectionSize**. It is used for path steps, such as **a/b/c**. For each step in the path, XQuery must provide a context containing the current item, its position, and the total item count. One notable difference between **TreeJoin** and **ProjectionConcat** is that **expr** is a **Calculation** instead of a tuple operator.

5.3.5 Universal Operators

The universal operators, which can process both tuples and items, include classic set operators, such as **Union**, **Intersection**, and **Difference**. The **Limit** operator allows only a set amount of results. **NullSource** is a data source that emits only one empty item.

²Its **condition** can access each **source** attribute as a pair of an array of accumulated values and the potential next value.

³The query executor is required to execute the recursion using breadth-first search, so that the shortest values are generated first.

```

SELECT
  count(id) FILTER (WHERE sex = 'M') AS men,
  count(id) FILTER (WHERE sex = 'F') AS women,
  collect(DISTINCT id ORDER BY id) AS all_ids
FROM sales
GROUP BY brand

```

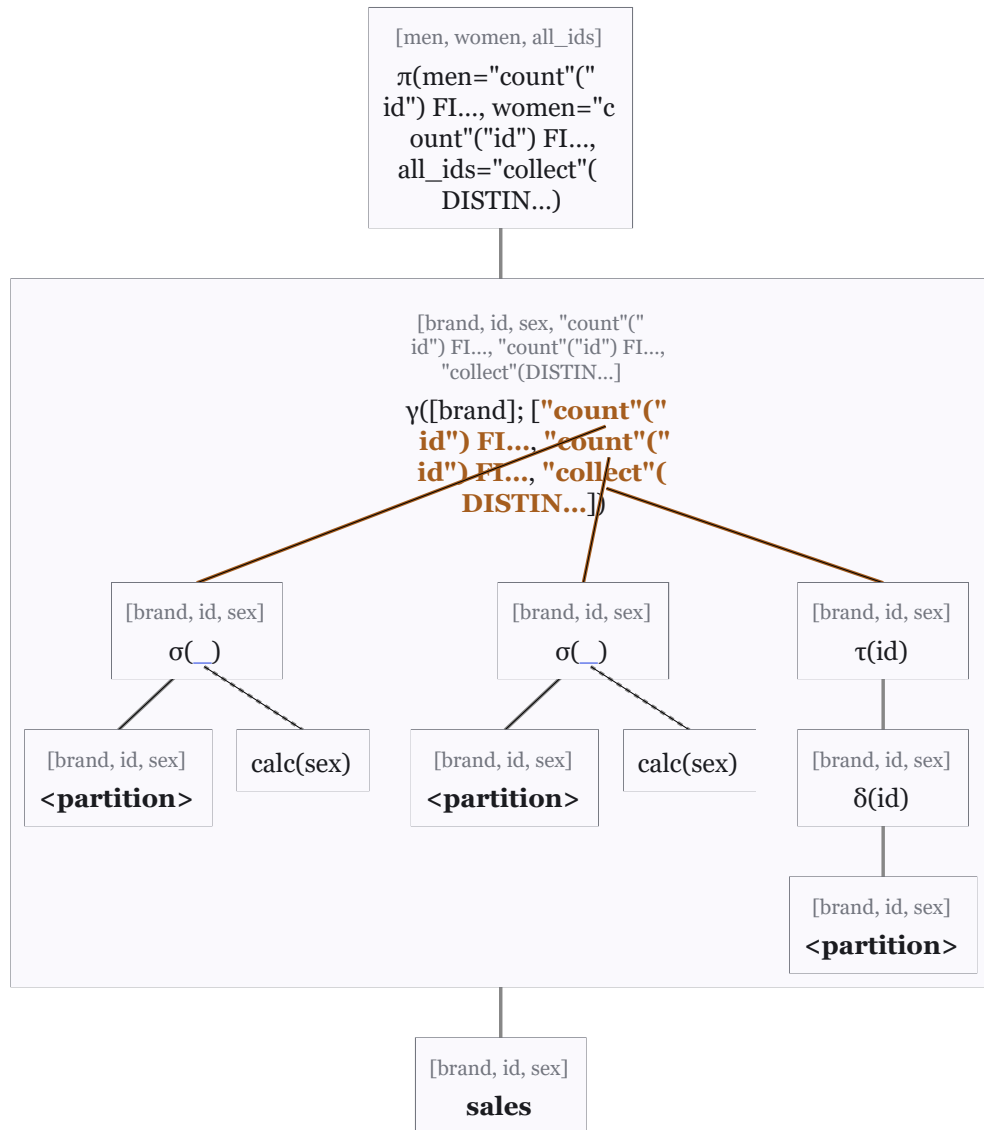


Figure 5.5 Example of a complex SQL aggregation. The **FILTER** clause is defined in SQL:2003 as an optional feature. It is implemented in PostgreSQL and SQLite.

```
MATCH ()-[path *4]->()
RETURN path
```

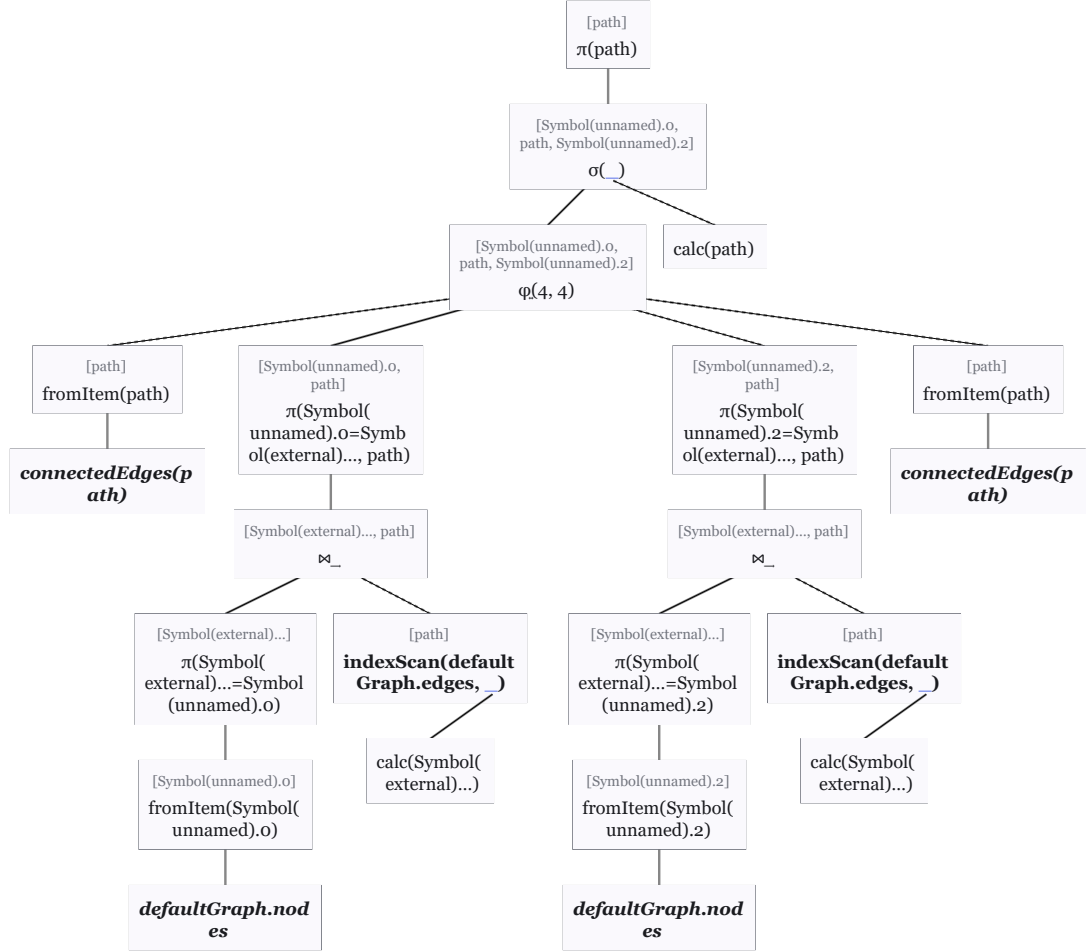


Figure 5.6 This Cypher query first selects all nodes from a graph. It then selects all of their outgoing and incoming edges using **IndexScans**. These edges are recursively expanded and connected into four-segment paths from both directions with the **BidirectionalRecursion** operator. The final **Selection** removes paths with duplicate edges created by joining the forward and backward subpaths.

```
SELECT 1 AS one
```

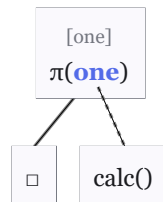


Figure 5.7 One possible application for the **NullSource** operator.

Extensibility

The algebra currently covers all of our needs, but it may not remain so. If a language requires functionality that cannot be easily expressed with the core logical operators, adding a new operator is as simple as defining its behavior. This is done by extending relevant visitor classes in the language itself; there is no need to modify anything in the DortDB core. This is exactly how the XQuery-specific operators are implemented.

5.3.6 Mapping of SQL to the Algebra

- Set operations are mapped to the `Union`, `Intersection` and `Difference` operators, possibly combined with `Distinct`.
- `SELECT` `attrs`
`FROM` `table`
`WHERE` `cond`
`GROUP BY` `attrs`
`HAVING` `cond`
`ORDER BY` `attrs`
maps intuitively to the `Projection`, `Selection`, `GroupBy`, `Selection` and `OrderBy` operators. Care must be taken with the `ORDER BY` clause, as it can include attributes that are not part of the `SELECT` `Projection`. In such a case, it may be necessary to extend the original `Projection` and add one more after the `OrderBy`.
- (Outer) joins are translated to the `CartesianProduct` and `Join` operators.
- Lateral joins and correlated subqueries are translated to the `ProjectionConcat` operator. Correlated subqueries in e.g. `WHERE` clause can also be translated as operators nested in the `Selection` condition `Calculation`.
- Quantified queries (`col > ALL(subquery)`) are translated to the `Quantifier` operator and later rebuilt into a `Calculation`.

5.3.7 Mapping of XQuery to the Algebra

- Constructors of XML values are translated to `FnCalls` and later rebuilt into `Calculations`.
- FLWOR expressions are translated as follows:
 - `for` is translated to the `FromItem` operator. If there are multiple iteration sources, they are combined using the `CartesianProduct` operator. If there is an `at $pos` specifier, the `ProjectionIndex` is included.
 - `let` is translated to the `Projection` operator.
 - `count` is translated to the `ProjectionIndex` operator.
 - `where` is translated to the `Selection` operator.
 - `group by` is translated to the `GroupBy` operator.

- `order by` is translated to the `OrderBy` operator.
- `return` is translated to the `ProjectionConcat` operator, as the returned item may be a sequence that needs to be flattened into the FLWOR output.
- Path expressions are translated to the XQuery `TreeJoin` operator, often combined with `Selections`. The `TreeJoin` operator provides the necessary context: `$fs:dot`, `$fs:position` and `$fs:last`.
- Aggregate functions are translated into `GroupBy` operators.

5.3.8 Mapping of Cypher to the Algebra

- The iteration of nodes and edges is done by iterating `defaultGraph.nodes` and `defaultGraph.edges` `ItemSources`. The default graph name can be configured globally when registering the language, or per query with a proprietary `FROM` clause.
- The `MATCH` clause is translated into a combination of `ItemSources`, `Projections`, `Selections` and `ProjectionConcats`.
 - Recursive path matches can be translated into any recursion operator. We use the `BidirectionalRecursion` operator to achieve the best performance.
 - If matching connected nodes and edges were to be done using only `ProjectionConcats`, `Selections`, and `ItemSources`, the queries would be prohibitively slow, and the whole underlying graph structure would not be properly utilized. The DorteDB optimizer translates these first into joins and then into index scans using a `ConnectionIndex`.
- The `WITH` and `RETURN` clauses are translated into `Projections`, possibly with `Limit`, `Distincts` or `OrderBy` operators included.
- Aggregate calls in `WITH` and `RETURN` clauses are translated into `GroupBy` operators.
- The `UNWIND` clause is translated into a `ItemFnSource`.

6 Optimization

One benefit of having a unified algebra for all languages is that a query can be optimized holistically, regardless of the languages in which it is written. DortDB comes with a rule-based optimizer. Even though cost-based optimizers generally perform better, they require statistics about the data being queried. DortDB data sources are, by design, schema-less, and no statistics collection currently takes place. The optimizer thus consists of an extensible set of query-rewriting and operator-reordering rules.

In Figure 6.1, we can see how the optimizer works through all the languages at once. Notice the orange-purple subtree, which has been moved into the middle of the green Cypher operators. This is the XQuery filter originally specified at the end of the SQL query.

6.1 Plan Operators

Not required by theoretical algebra, but included for better optimization, there are three specialized operators. Two are used to enable secondary indices. **IndexScan** replaces a **TupleSource** or an **ItemSource** paired with a **MapFromItem**. It contains an **access Calculation**, which selects relevant items from the underlying indexed data source. The self-join nature of **Recursion** does not leave any opportunity for indices, so it may be more efficient to use the **IndexedRecursion** operator, which acts as a recursive depend-join. While a little more difficult to set up, the **BidirectionalRecursion** operator offers significant asymptotic improvements in both memory and time.

Algebraic Operator	Operator Signature	Operator Schema
IndexScan	indexScan (name, access, fromItemKey)	unknown
IndexedRecursion	$\xrightarrow{\phi}$ (min, max, mapping, source)	source
BidirectionalRecursion	$\xleftrightarrow{\phi}$ (min, max, mappingFwd, mappingRev, target, source)	source $\oplus$ target

Table 6.1 Operators of the Unified Algebra: Tuple Operators – Optimizer

```

SELECT products.name
FROM products
JOIN (
  LANG cypher
  MATCH (p:person)-[:HAS_INTEREST]->(c:category)
  RETURN p, c.name AS category
) AS interests
ON products.category = interests.category
WHERE interests.p->'id' IN (
  LANG XQuery
  $Invoices/Invoice[
    orderDate < date:sub(now(), interval('1 month'))
  ]/personId/fn:data()
)

```

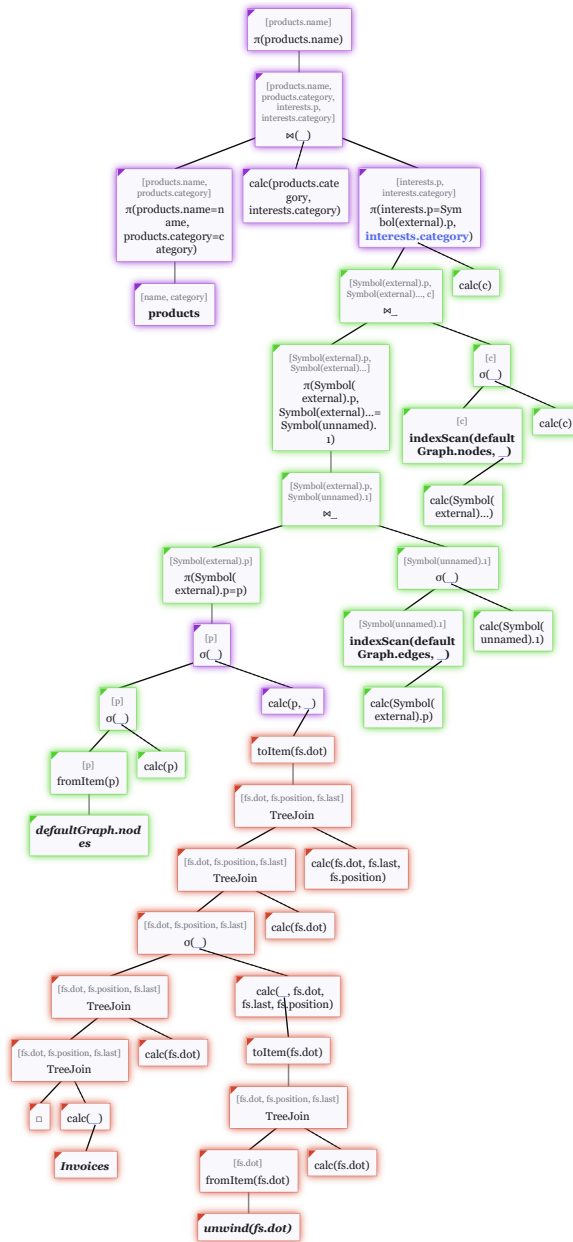


Figure 6.1 Example of cross-model optimization.

Algebraic Operator	Input Domain	Output Domain	Semantics
$\xrightarrow{\phi}(\text{min}, \text{max}, \text{mapping}, \text{source})$	min: $\mathbb{N}$, max: $\mathbb{N}$, mapping: $\text{inst}_{\text{Stream}(\mathcal{T})}$, source: $\text{Stream}(\mathcal{T})$	$\text{Stream}(\mathcal{T})$	$\xrightarrow{\phi}(0, 1, M, S, \Gamma) = \langle \{a \mapsto \langle s.a \rangle \mid a \in s\} \mid s \in S \rangle$ $\xrightarrow{\phi}(0, \text{max}, M, S, \Gamma) = \langle$ $\quad \{a \mapsto \text{acc}.a \cup \langle \text{curr}.a \rangle \mid a \in \text{acc}\}$ $\quad \mid \text{curr} \in M(\Gamma \oplus \text{acc}), \text{acc} \in \xrightarrow{\phi}(0, \text{max} - 1, C, S, \Gamma)$ $\quad \rangle$ $\xrightarrow{\phi}(\text{min}, \text{max}, M, S, \Gamma) = \langle$ $\quad p \mid p \in \xrightarrow{\phi}(0, \text{max}, M, S, \Gamma), \forall a \in p : p.a \geq \text{min}$ $\quad \rangle$

Table 6.2 Selected Operators of the Unified Algebra – formal: Tuple Operators (Optimizer)

6.2 Secondary Indices

DortDB allows users to define secondary indices. Unlike single-model databases, multimodel database systems generally require a broader range of index types [23]. As DortDB aims to be extensible and support as many query languages and models as possible, it is necessary to allow the definition of custom index types. Furthermore, different index types might require different conditions to be applied. For example, a hash table index generally requires the query to include an equality check on a specific attribute, whereas a range index can also support inequality comparisons.

Our solution is to allow the indexing of any expression composed of **Calculation** intermediaries. This means that any **Calculation** that does not contain subqueries is indexable. During the query optimization phase, registered indices receive expressions that access a data structure and determine whether the indices are applicable. Currently, the first matching index is selected. This behavior is contained in two replaceable optimizer rules, so developers using DortDB may implement a better algorithm.

In Figure 5.6, an index provided by the Cypher language detects conditions for connected nodes and edges. In Figure 6.2, a simple hash table index recognizes a relevant equality check.

6.3 Implemented Optimizer Rules

The rules in the optimizer follow a simple format. Each rule specifies a set of starting operators. As the rule is processed, the optimizer traverses the query tree from the root down, looking for any operator of a matching type. After such an operator is found, the rule evaluates its **match** method. If successful, the **match** function produces bindings, which are supplied to the rule's **transform** method alongside the subtree starting with the matched operator. The **transform** method performs any necessary query rewriting or operator reordering and returns a new subtree, which replaces the original one.

DortDB comes with a set of implemented rules. During configuration, the user can choose which rules to use and in what order. In the following sections, the rules will be explained in the order they are applied by default. Each rule description will include a formal specification (which may be slightly simplified) and an example plan before and after the optimization.

6.3.1 Unnest Subqueries

This rule finds **Calculations** that contain safe subqueries (nested operators). A subquery is safe if it produces at most one value (the containing **Calculation** does not accept a sequence of values), and if it is always evaluated (usually true, unless the containing **Calculation** is a logical operator like **AND** or **OR**). For example, `(SELECT number FROM table1) + 5` will be matched while

`val IN (SELECT number FROM table1)` will not. The subquery is then moved in front of the original operator as a **ProjectionConcat**, and the **Calculation** argument is replaced with an identifier. Unless the subquery is guaranteed to always produce a value (for example, if it is a **COUNT(*)** expression), the

```
// DortDB programmatic configuration:
db.createIndex(['t2'], ['a + b / 2'], MapIndex)
```

```
SELECT t1.foo, t2.bar FROM t1
JOIN t2
ON t2.a + t2.b / 2 = t1.id
```

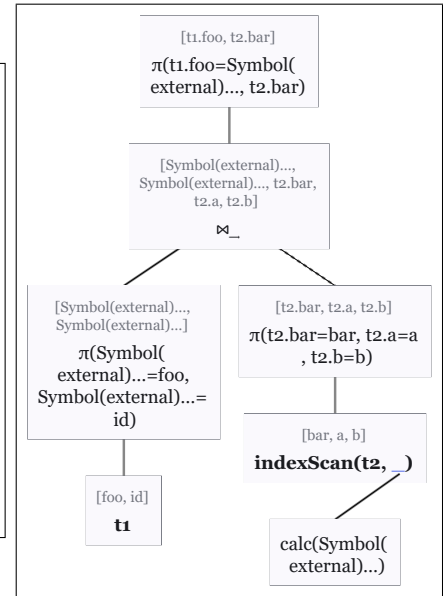
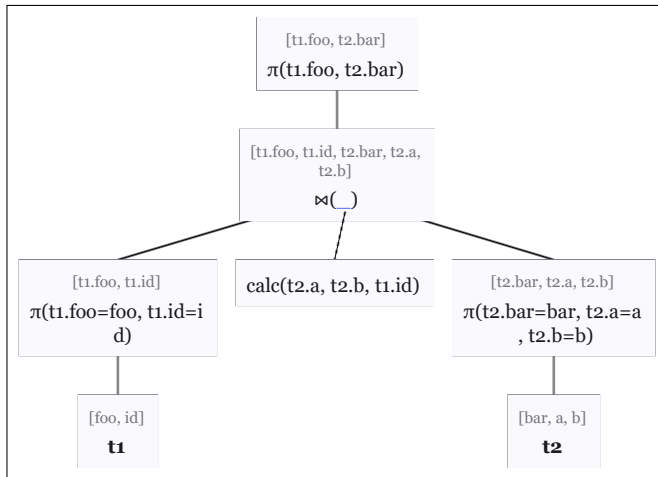


Figure 6.2 The registered index recognizes the equality check on the expression it was registered with. Plan before and after the optimization. For more details, see optimizer rules Join Indices and Index Scans.

ProjectionConcat must be outer. This unnesting does not offer any speedup on its own, but it presents opportunities for further optimization.

$$\begin{array}{c}
Op_1(Args, Source) \\
Op_1 \in \{\pi, \sigma, \gamma, \delta, \tau\} \\
Arg_i = calc(Cargs) \\
isOperator(Carg_j) \\
\neg acceptSequence(Carg_j) \\
\neg maybeSkipped(Carg_j) \\
\hline
Id_1 = uniqId() \\
Op_2 = Op_1(Args \setminus Arg_i \cup calc(Cargs \setminus Carg_j \cup Id_1), Source) \\
outer = \neg guaranteedValue(Carg_j) \\
\pi(schema(Op_1) \setminus Id_1, \overset{\boxtimes}{\rightarrow} (MapFromItem(Id_1, Carg_j), outer, Op_2))
\end{array}$$

Rule 6.1 Unnest Subqueries

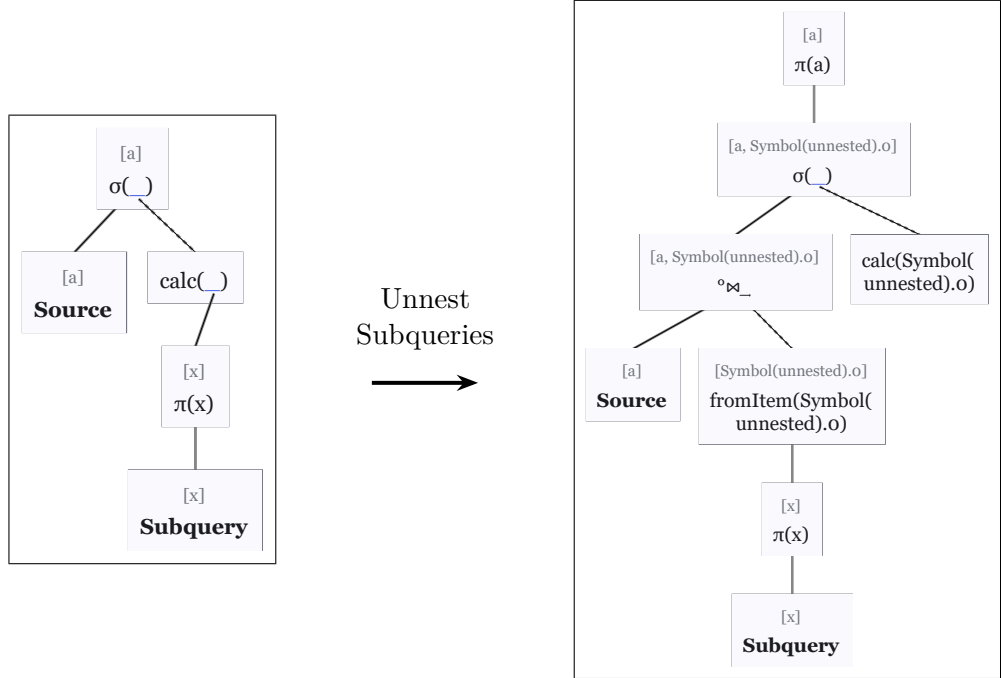


Figure 6.3 An example of an unnested subquery in a **Projection**. The resulting **ProjectionConcat** is outer.

6.3.2 Merge To/FromItems

As can be seen in Figure 6.3, it is possible for **MapFromItem** to have **MapToItem** as its source, or vice versa. This happens most often due to query rewriting rules or language switches. The two rules for handling these situations are rather simple – if **MapFromItem** is the parent and **MapToItem** is its source, they become a **Projection** with a single attribute. Otherwise, both operators are simply removed.

$$\frac{MapToItem(key_1, MapFromItem(key_1, Source))}{Source}$$

Rule 6.2 Merge ToFromItems

$$\frac{\text{MapFromItem}(key_1, \text{MapToItem}(key_2, \text{Source}))}{key_2 \neq *}{\pi(key_2 \rightarrow key_1, \text{Source})}$$

Rule 6.3 Merge FromToItems

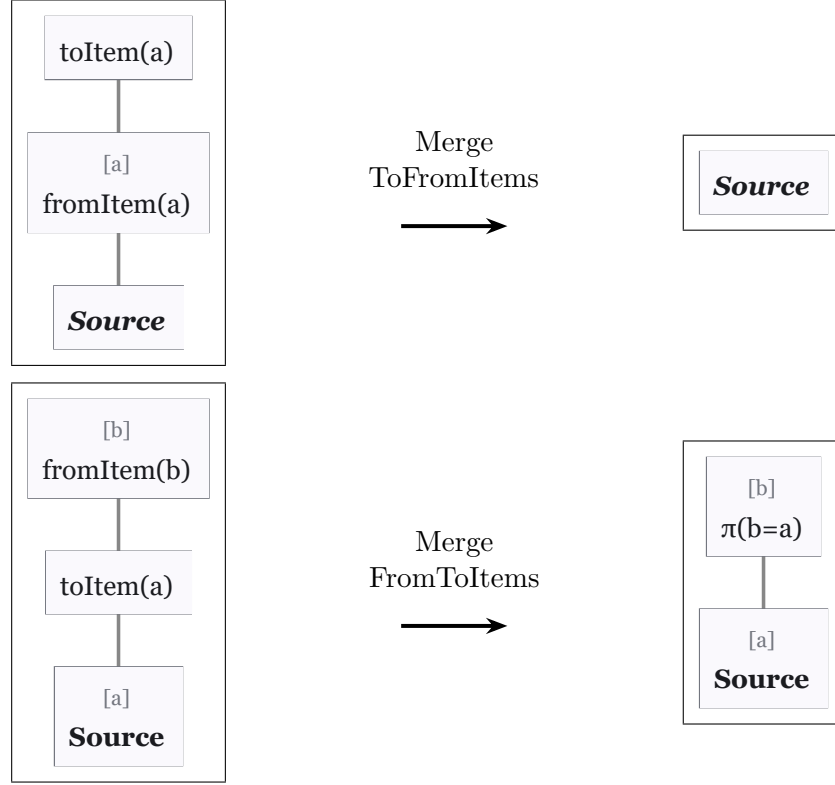


Figure 6.4 An example of the two possible directions of merging `MapFromItem` and `MapToItem` operators.

6.3.3 Pushdown Selections

Selections reduce the cardinality of a tuple stream. It is, therefore, advantageous to process them sooner rather than later. This rule moves **Selections** closer to the leaves of the query tree.

A **Selection** can always be safely pushed below **OrderBy** and **Distinct**. It can also always be pushed down below set operators like **Union** or **Difference**, but it needs to be duplicated and applied to both source branches.

A **Selection** can be pushed below a **Projection** only if two conditions hold. First, the predicate must not depend on attributes created by **Calculations** in the **Projection**; it may only refer to renamed or unchanged attributes. Second, the required renaming must preserve the semantics of the condition, for example, by avoiding variable shadowing. See Listing 18 for more details.

When it comes to **Joins** and **CartesianProducts**, the **Selections** can be pushed into branches that contain all necessary attributes for evaluating the

Program 18 Pseudocode of the `canRename` predicate checking a single renaming.

```

canRename(Op, x → y) =
  when x ∈ schema(Op) and y ∈ schema(Op) then false
  when y ∈ ⋃h ∈ horizontal(Op) freeVars(h) \ ⋃v ∈ vertical(Op) schema(v) then false
  otherwise ⋀o ∈ horizontal(Op) ∪ vertical(Op) canRename(o)

```

conditions, as long as the `Join` is not left outer and it is not its right branch or vice versa.

$$\frac{\sigma(Cond_1, Op_1(Args, Source)) \quad Op_1 \in \{\tau, \delta\}}{Op_1(Args, \sigma(Cond_1, Source))}$$

Rule 6.4 Pushdown Selections 1

$$\frac{\sigma(Cond_1, Op_1(Left, Right)) \quad Op_1 \in \{\cup, \cap, \setminus\}}{Op_1(\sigma(Cond_1, Left), \sigma(Cond_1, Right))}$$

Rule 6.5 Pushdown Selections 2

$$\frac{\sigma(Cond_1, \times(Left, Right)) \quad \text{boundVars}(Cond_1) \subseteq \text{boundVars}(Left)}{\times(\sigma(Cond_1, Left), Right)}$$

Rule 6.6 Pushdown Selections 3

$$\frac{\sigma(Cond_1, \bowtie(Left, Right, leftOuter, \mathbf{false}, Conds)) \quad \text{boundVars}(Cond_1) \subseteq \text{boundVars}(Left)}{\bowtie(\sigma(Cond_1, Left), Right, leftOuter, \mathbf{false}, Conds)}$$

Rule 6.7 Pushdown Selections 4

The two above rules (Pushdown Selections 3 and Pushdown Selections 4) are similarly applied to the right `CartesianProduct` or `Join` branches (or to both branches at once).

$$\frac{\sigma(Cond_1, \pi(Attrs, Source)) \quad \text{canRename}(Cond_1, Attrs)}{\pi(Attrs, \sigma(\text{rename}(Cond_1, Attrs), Source))}$$

Rule 6.8 Pushdown Selections 5

While the above rules work for single **Selections**, chains of **Selections** could result in valid pushdowns not being applied (for example, if there is a **Selection** that cannot be pushed down through a **CartesianProduct**, it would block all chained **Selections** that come after from being considered). If the optimizer were cost-based, we could introduce a rule that simply swaps two selections. Our solution, however, is to consider the entire chain of **Selections** for each pushdown rule and push down only those parts of the chain that are eligible.

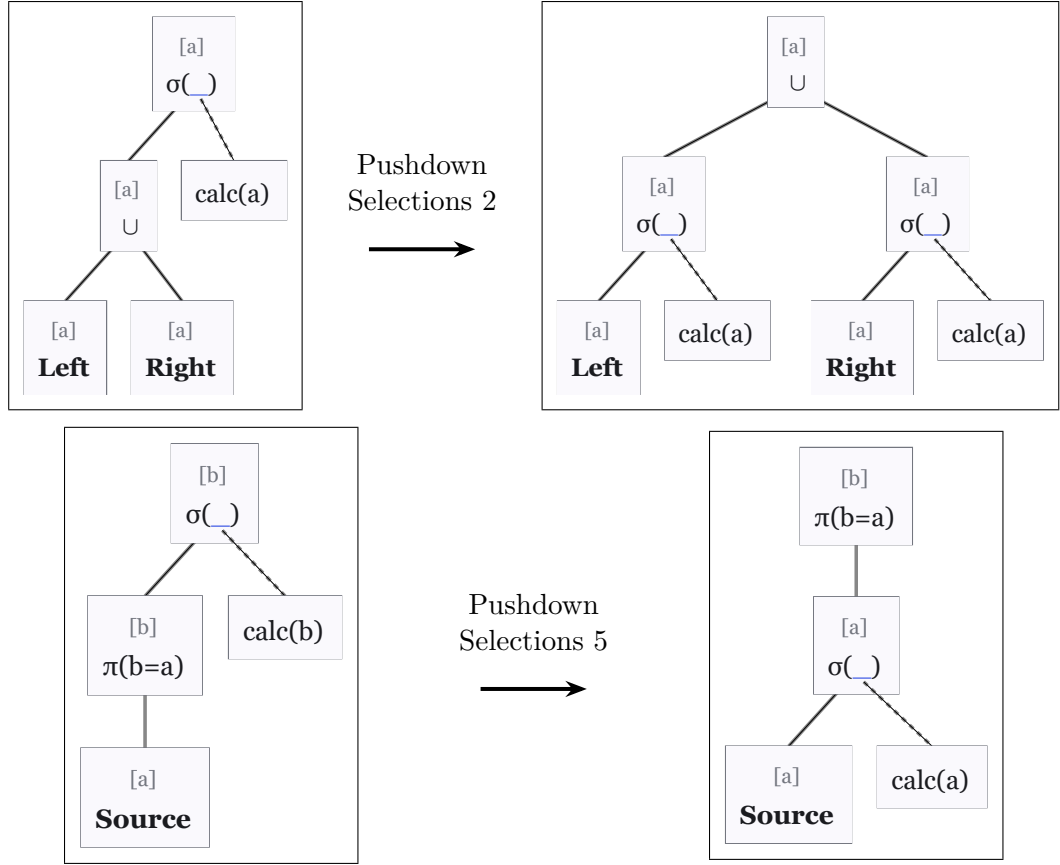


Figure 6.5 Two examples of Selection pushdown.

Program 19 An overly contrived example where it is not possible to fully push the Selection into the t2 subquery due to renaming collisions.

```

SELECT t.attr2
FROM (
  SELECT t.orig1 AS attr1, t.orig2 AS attr2
  FROM t
) AS t
WHERE t.attr1 > (SELECT t.orig1 FROM t2 AS t)

```

6.3.4 ProjectionConcat to Join

ProjectionConcat may be converted to a CartesianProduct when its mapping does not depend on the source context (the subquery is not correlated). If the ProjectionConcat is outer, it is converted to a left outer Join instead.

$$\frac{\begin{array}{c} \rightarrow (Op_1, \text{false}, Op_2) \\ \text{freeVars}(Op_1) \cap \text{boundVars}(Op_2) = \emptyset \end{array}}{\times(Op_1, Op_2)}$$

Rule 6.9 ProjectionConcat to Join 1

$$\frac{\begin{array}{c} \Join_{\rightarrow} (\text{Op}_1, \mathbf{true}, \text{Op}_2) \\ \text{freeVars}(\text{Op}_1) \cap \text{boundVars}(\text{Op}_2) = \emptyset \end{array}}{\Join (\text{Op}_1, \text{Op}_2, \mathbf{true}, \mathbf{false}, [])}$$

Rule 6.10 ProjectionConcat to Join 2

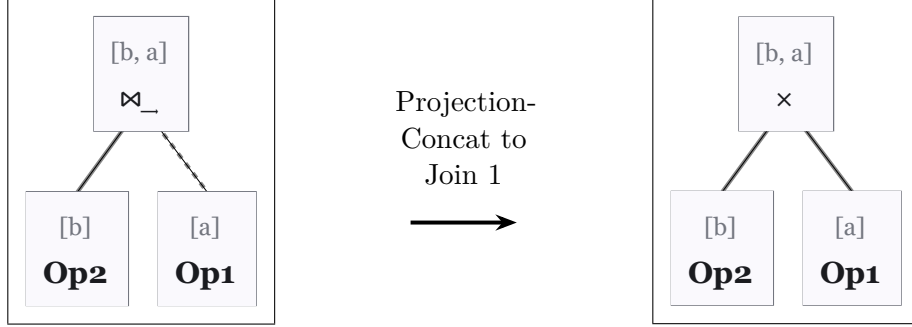


Figure 6.6 A non-correlated non-outer subquery can be converted to a regular CartesianProduct.

6.3.5 Products to Joins

This rule merges **Selections** with subsequent **CartesianProducts** into **Joins**. The resulting **Join** will have multiple conditions rather than a single combined condition.

$$\frac{\sigma(\text{Cond}_1, \times(\text{Op}_1, \text{Op}_2))}{\Join (\text{Op}_1, \text{Op}_2, \mathbf{false}, \mathbf{false}, [\text{Cond}_1])}$$

Rule 6.11 Products to Joins 1

$$\frac{\sigma(\text{Cond}_1, \Join (\text{Op}_1, \text{Op}_2, \mathbf{false}, \mathbf{false}, \text{Conds}))}{\Join (\text{Op}_1, \text{Op}_2, \mathbf{false}, \mathbf{false}, \text{Conds} \cup \text{Cond}_1)}$$

Rule 6.12 Products to Joins 2

6.3.6 Join Indices

This rule is the first part of the two-step handling of secondary indices. It converts applicable **Joins** to **ProjectionConcat**s in a suitable form to be later transformed into **IndexScans**. The specifics of index matching are explained in Section Secondary Indices.

$$\frac{\begin{array}{c} \Join (\text{Op}_1, \text{Op}_2, \text{leftOuter}, \mathbf{false}, \text{Conds}) \\ \text{Op}_1 \in \{\text{TupleSource}(\text{name}), \text{MapFromItem}(\text{key}, \text{ItemSource}(\text{name}))\} \\ \text{MatchedConds} \subseteq \text{Conds} \\ \text{indexExists}(\text{name}, \text{MatchedConds}) \end{array}}{\Join_{\rightarrow} (\sigma(\text{Conds} \setminus \text{MatchedConds}, \sigma(\text{MatchedConds}, \text{Op}_1)), \text{leftOuter}, \text{Op}_2)}$$

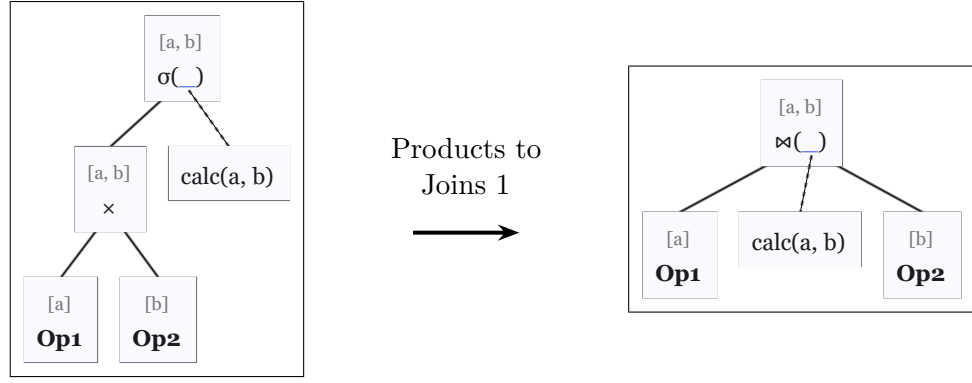


Figure 6.7 The Selection cannot be pushed down, because it depends on attributes from both branches of the CartesianProduct. The CartesianProduct can be, however, converted to a Join operator.

Rule 6.13 Join Indices

The rule as specified here is actually slightly simplified. In practice, the matched Op_1 branch can also contain arbitrary chains of **Selections**, **OrderBys**, and **Projections**. The index matching is aware of renamed attributes and excludes new attributes computed by **Projections**. Additionally, the Op_2 branch attributes need to be renamed to unique names and back, so that the condition inserted into the Op_1 branch can be pushed down and renamed safely if there are any **Projections** present.

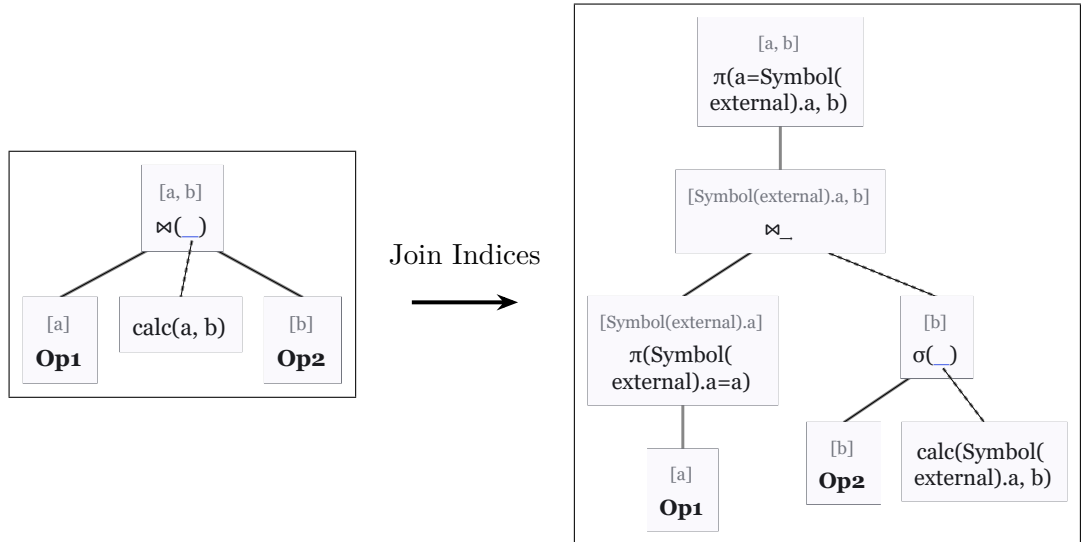


Figure 6.8 The simplest possible case of the join indices rule. There is an index on Op_2 that matches the condition Calculation.

6.3.7 Index Scans

This rule replaces eligible filtered data sources with **IndexScan** operators. Both **TupleSources** and **ItemSources** can be converted.

$$\begin{array}{c}
\sigma(\text{cond}, \text{TupleSource}(\text{name})) \\
\text{indexExists}(\text{name}, \text{cond}) \\
\text{index} = \text{getIndex}(\text{name}, \text{cond}) \\
\hline
\text{IndexScan}(\text{name}, \text{createAccessor}(\text{index}))
\end{array}$$

Rule 6.14 Index Scans 1

$$\begin{array}{c}
\sigma(\text{cond}, \text{MapFromItem}(\text{key}, \text{ItemSource}(\text{name}))) \\
\text{indexExists}(\text{name}, \text{cond}) \\
\text{index} = \text{getIndex}(\text{name}, \text{cond}) \\
\hline
\text{IndexScan}(\text{name}, \text{createAccessor}(\text{index}), \text{key})
\end{array}$$

Rule 6.15 Index Scans 2

Similarly to **Selection** pushdown, this rule actually considers whole chains of **Selections** at once.

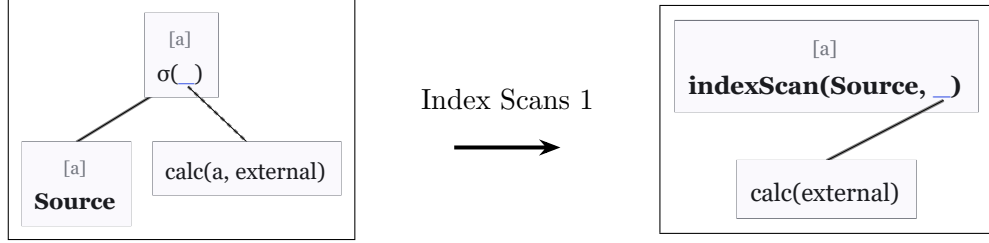


Figure 6.9 Example of a filtered **TupleSource** replaced by a **IndexScan**. The newly created **Calculation** is an index accessor that serves as an input for the underlying index data structure.

6.3.8 Merge Projections

This is likely the most complex rule currently implemented in the DortDB optimizer. It combines subsequent **Projections** into a single **Projection** when possible. Unused attributes are removed. Calculated attributes are merged as long as the **Calculation** is only referenced once, in order to avoid executing potentially expensive computations multiple times.

$$\begin{array}{c}
\pi(\text{Attrs}_1, \pi(\text{Attrs}_2, \text{Source})) \\
\text{Calcs}_2 = \{a \in \text{Attrs}_2; \text{isCalc}(\text{src}(a))\} \\
\neg \text{referencedMultipleTimes}(\text{Calcs}_2, \text{Attrs}_1) \\
\hline
\pi(\text{Attrs}_1 \circ \text{Attrs}_2, \text{Source})
\end{array}$$

Rule 6.16 Merge Projections

As an example of two **Projections** that would be kept unchanged, consider:

$$\pi([x \rightarrow a, x \rightarrow b], \pi([\text{calc}(1 + 1) \rightarrow x], \text{Source}))$$

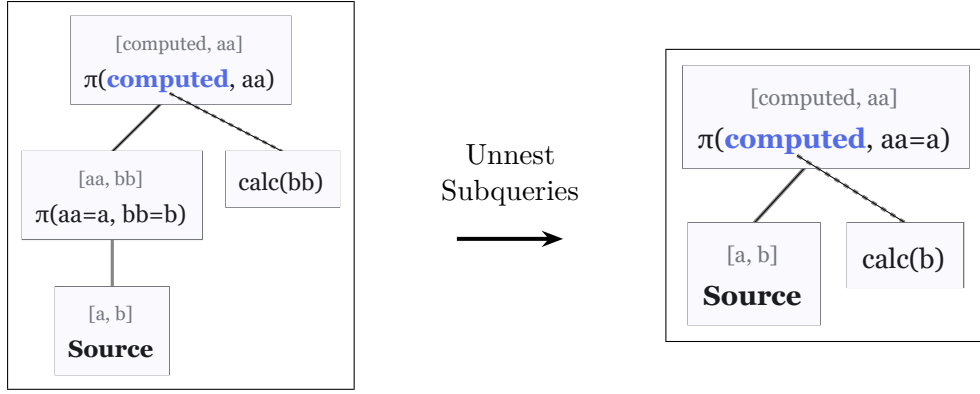


Figure 6.10 Example of two merged Projections.

6.4 Calculation Building

Calculations are usually created from intermediary plan operators such as **FnCalls**. This process aims to create a single final callable. **FnCalls** can be marked as *pure*. If arguments of pure **FnCalls** are constant (represented by **Literal** plan operators), the function is precomputed and replaced by a **Literal** containing the result. This way, redundant computation may be avoided.

6.5 Language-Specific Optimization

Given that all of the languages are translated into the unified algebra, optimizations specific to a single language are uncommon. While it is possible to add optimizer rules that target logical plan nodes based on their source language, it is generally more sensible to apply them universally. All that being said, there are some highly specific optimizations implemented in the language plugins, not in the core library.

SQL includes *quantified queries*. These take the form of a comparison operator combined with the **ANY** or **ALL** keyword and a subquery. When the comparison operator in question is **>**, **>=**, or a similar operator, the quantified subquery can be replaced with a suitable aggregation [24], see Program 20. This happens during the conversion of the query AST to a logical plan.

Program 20 Example of a SQL quantified query.

```
SELECT x FROM a
WHERE x < ALL(
  SELECT y FROM b
)
-- evaluated as the following:
SELECT x FROM a
WHERE x < (SELECT min(y) FROM b)
```

The data accessed via the DortDB Cypher language will likely be stored in a structure that supports efficient neighbor lookups. The graph traversal, however,

is implemented as joins across node and edge sources, failing to leverage the underlying structure. This is corrected using the `ConnectionIndex` secondary index, which detects joins via the Cypher data adapter's `isConnected` condition. The index stores no data on its own; it is merely a wrapper for the Cypher data adapter's `filterNodeEdges` or `getEdgeNode` methods. See Figure 5.6 for an example of indexed Cypher graph traversal.

7 Benchmarks

In the previous chapters, we have introduced the concept of language switching and the unified algebra. We have implemented three query languages and various optimizations. Next, we will evaluate DortDB and compare our results with other established solutions. We will first examine the viability of our approach on multimodel workloads. Then, we will focus on DortDB as a SQL library for JavaScript environments.

All of the experiments were conducted on a machine with a 12-core AMD Ryzen 9 7900 CPU, 32 GB of 6000 MHz RAM, and an SSD.

7.1 UniBench

UniBench [25] is a benchmark designed for evaluating multimodel databases. It provides a data model simulating *social commerce*, a combination of social media and E-commerce. We will use a pre-generated 1 GB dataset, as well as a smaller 5 MB sample created by selecting data from 100 customers. In addition to the data, the benchmark specifies 10 analytical queries that combine relational, document, and graph data models. The queries are described in natural language and need to be rewritten for each tested system, as different multimodel databases use different query languages. The queries used by DortDB are included in Appendix UniBench Queries.

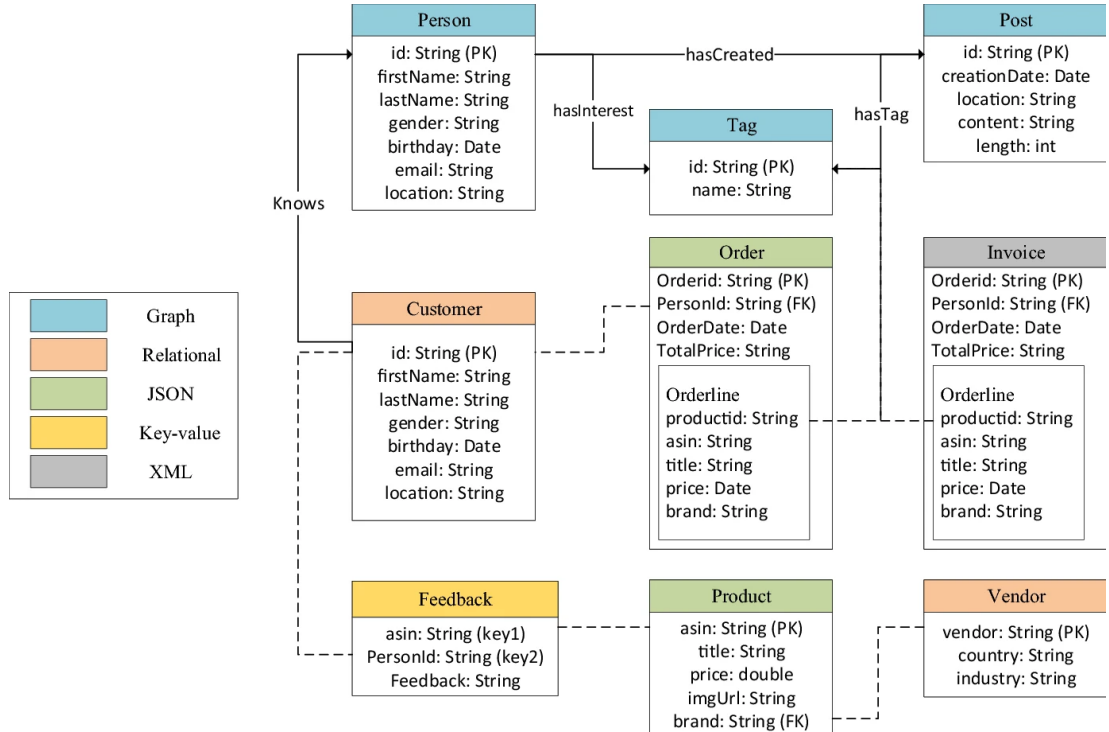


Figure 7.1 Schema of the UniBench data model [26].

Part of the UniBench repository are example queries for ArangoDB¹ and OrientDB,² as well as scripts for importing data into the database systems. We used OrientDB version 2.2.16, the same as in the original paper, because the import mechanisms have changed since then, and the provided scripts could not be used otherwise. The importing scripts had to be slightly modified anyway, and we had to configure several indices and vertex properties in the database schema in order for the OrientDB queries to work. The ArangoDB version used was 3.12.5, the latest at the time of writing. There were no issues with either the ArangoDB data import or the ArangoDB queries.

7.1.1 Data Import

We measured the time required to import data into each system. For DortDB, the UniBench data was decompressed from a zip archive, parsed, loaded into memory, and indexed; this process was repeated multiple times, and the results were averaged. For ArangoDB and OrientDB, import times were taken from the provided import scripts and reflect a single run.

Program 21 Indices created on DortDB UniBench data sources. UniBench only allows explicit indices on primary keys.

```
db.createIndex(['defaultGraph', 'nodes'], [], ConnectionIndex);
db.createIndex(['defaultGraph', 'nodes'], ['x.id'], MapIndex, {
  mainLang: 'cypher',
  fromItemKey: ['x'],
});
db.createIndex(['defaultGraph', 'edges'], [], ConnectionIndex);
db.createIndex(['customers'], ['id'], MapIndex);
db.createIndex(['products'], ['productId'], MapIndex);
db.createIndex(['vendors'], ['id'], MapIndex);
db.createIndex(['posts'], ['id'], MapIndex);
db.createIndex(['orders'], ['OrderId'], MapIndex);
```

The results are displayed in Figure 7.2. Disregarding the sampled dataset, ArangoDB was more than twice as fast as DortDB, even though DortDB did not write to disk. One possible explanation is that DortDB runs in a single thread, while ArangoDB is multithreaded. OrientDB data import took about 7 minutes, which is within expectations. The small dataset import in ArangoDB completed in under 1 second – and since the import scripts report only in whole seconds, we omitted this entry from the graph rather than display a misleading empty bar. As the full dataset import into OrientDB proved to be a very painful and time-consuming process, we opted not to repeat the exercise with the sampled dataset.

¹<https://arangodb.com/>

²<https://orientdb.dev/>

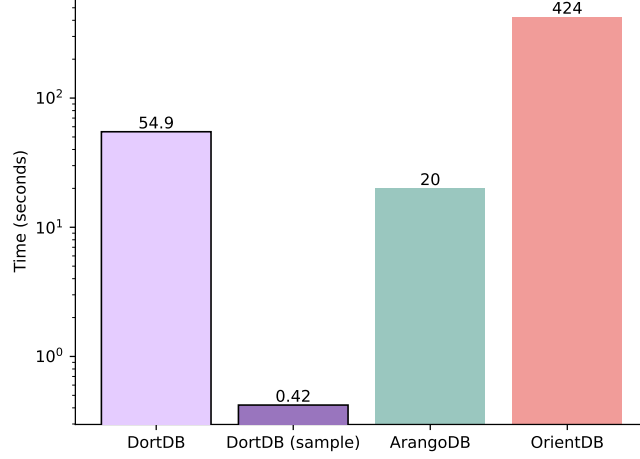


Figure 7.2 Processing time for data import.

7.1.2 Query Performance

Each query was executed multiple times, and the results were averaged. Most queries were run 4 times, unless the cumulative execution time exceeded 4 hours. For parametrized queries, such as Query 1, which retrieves all available information about a specific customer, parameters were drawn from a pool provided with the dataset and randomized on each execution.

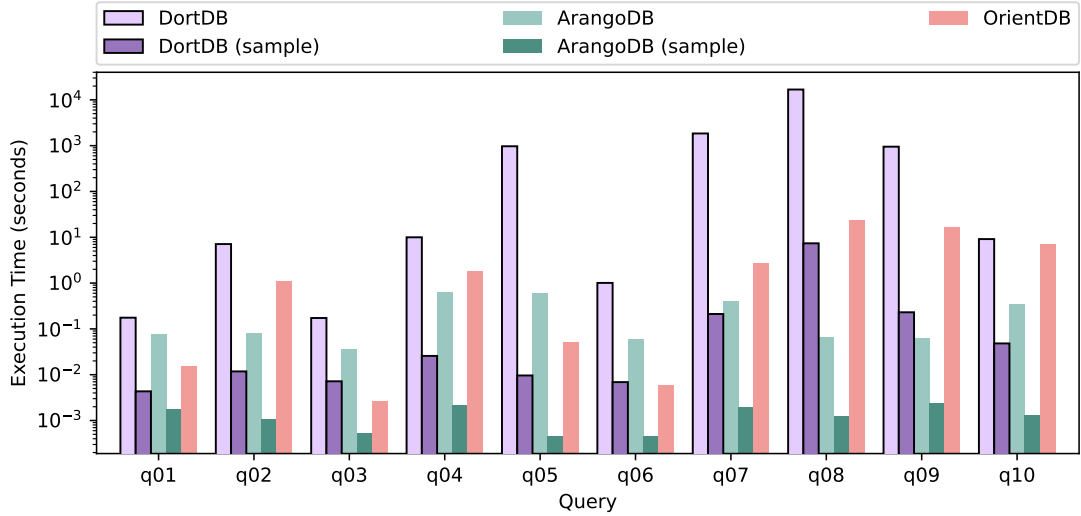


Figure 7.3 Average execution time for each query.

The results are shown in Figure 7.3. Interestingly, OrientDB outperforms ArangoDB on several queries, contrary to the original paper’s findings.

DortDB, on the other hand, is the slowest system overall, in some cases by more than two orders of magnitude. These large execution times generally correspond to XML tree traversal, as the framework currently lacks XML-specific access optimizations beyond the general optimizer rules and does not support XML indices, though the benchmark would not permit those regardless. For Query 4, we initially attempted a more naive and intuitive formulation; however, our optimizer proved insufficiently powerful, and the query ran out of memory. We

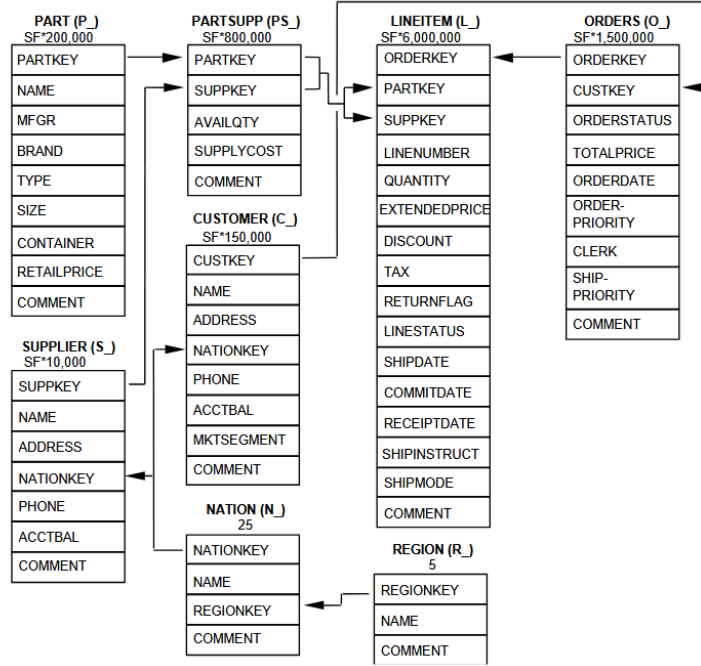


Figure 7.4 TPC-H benchmark schema. Taken from the TPC-H specification.⁵

therefore adopted a better formulation based on the provided ArangoDB query.

In Query 8, the XQuery subquery passes tuple data to SQL as an XML element. We have tested an alternate version of the query without the XML element, and the performance difference was negligible.

DortDB fares better with relational data and aggregation-heavy queries such as Query 10, where it performs comparably to OrientDB, a system not particularly optimized for relational workloads. While DortDB does not outperform either competing system, this should be considered in light of the fact that it runs in an interpreted language on a single thread.

Note that the execution times of Query 4, 5, and 6 on the smaller dataset should be interpreted with caution. The 100 sampled customers have very few KNOWS edges between them, making the results artificially fast.

7.2 SQL

Many benchmarks are available for SQL databases, but the TPC-H³ is perhaps the best known. It features 22 analytical queries with moderate to high complexity. Data and queries were generated using an open-source reimplementation of the benchmark⁴ with a scale factor of 0.1, yielding 100 MB of data. Minor modifications were made to the queries to accommodate the SQL dialect of each tested database. The DortDB versions of queries are provided in Appendix TPC-H Queries.

We have compared DortDB with two other JavaScript in-memory SQL data-

³<https://www.tpc.org/tpch/>

⁴<https://github.com/gregrahn/tpch-kit>

⁵https://www.tpc.org/TPC_Documents_Current_Versions/pdf/TPC-H_v3.0.1.pdf

Program 22 Indices created for the TPC-H benchmark. TPC-H allows explicit indexing of both primary and foreign keys.

```
db.createIndex(['customer'], ['custkey'], MapIndex);
db.createIndex(['customer'], ['nationkey'], MapIndex);
db.createIndex(['lineitem'], ['orderkey'], MapIndex);
db.createIndex(['lineitem'], ['partkey'], MapIndex);
db.createIndex(['lineitem'], ['suppkey'], MapIndex);
db.createIndex(['nation'], ['nationkey'], MapIndex);
db.createIndex(['nation'], ['regionkey'], MapIndex);
db.createIndex(['orders'], ['custkey'], MapIndex);
db.createIndex(['orders'], ['orderkey'], MapIndex);
db.createIndex(['part'], ['partkey'], MapIndex);
db.createIndex(['partsupp'], ['partkey'], MapIndex);
db.createIndex(['partsupp'], ['suppkey'], MapIndex);
db.createIndex(['region'], ['regionkey'], MapIndex);
db.createIndex(['supplier'], ['suppkey'], MapIndex);
db.createIndex(['supplier'], ['nationkey'], MapIndex);
```

bases. SQL.js⁶ is SQLite⁷ compiled into WebAssembly, while AlaSQL⁸ is a rich implementation of SQL with many features. We used SQL.js version 1.14.1 and AlaSQL version 4.17.2. Each query was run up to 4 times, with a cumulative timeout of 1 hour. The averaged results are in Figure 7.5.

⁶<https://sql.js.org/>

⁷<https://sqlite.org/>

⁸<https://github.com/AlaSQL/alasql>

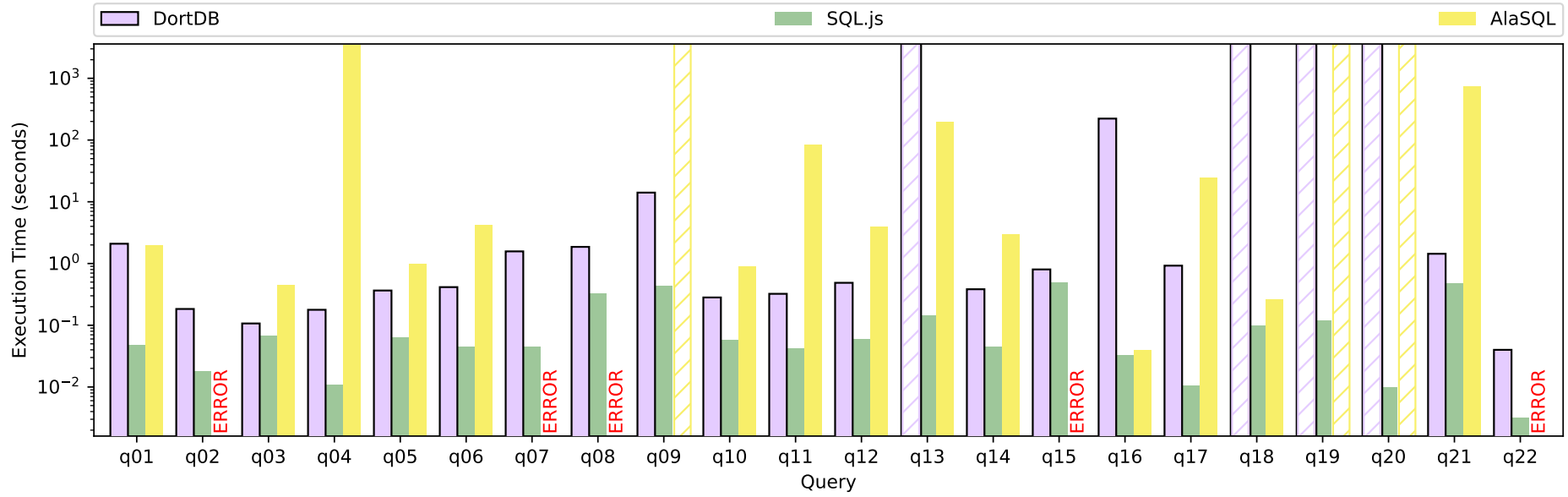


Figure 7.5 Average execution time for each query.

It comes as no surprise that SQL.js achieved the best execution times. It runs in native code and has a very powerful query optimizer, owing to more than 20 years of development. Compared to that, AlaSQL timed out on three of the queries and crashed on five.⁹ According to its GitHub wiki, it does not have a query optimizer per se, but it preemptively indexes tables before joining, and it tries to push down **WHERE** conditions before joins. It is, therefore, quite surprising how fast AlaSQL runs. AlaSQL is faster than DortDB on three queries, at one point coming very close to SQL.js.

DortDB managed most of the queries quite well, timing out on four. Query 13 contains a **LIKE** comparison with three % wildcards in a join condition. Queries 16, 18, and 20 contain complex **IN** operators, which are currently not optimized. Finally, Query 19 tests the optimization of **OR** conditions, which is also not implemented.

⁹We also ran this benchmark approximately a year prior to writing. At that time, AlaSQL crashed on only three queries but timed out on 15. Since then, the project has seen considerable development activity, with AI agents contributing to a large portion of the progress. We sincerely wish the maintainers well in their continued efforts.

8 Implementation

DortDB is a collection of TypeScript libraries. Its source code is available as an appendix to this thesis and online at <https://github.com/filipjezek/dortdb>. All the libraries are part of a single monorepository managed with Nx.¹ The `@dortdb/core` is the main library containing the core of the framework. There is a separate library for each of the currently implemented languages (`@dortdb/lang-cypher`, `@dortdb/lang-sql`, and `@dortdb/lang-xquery`). Additionally, there are packages for the GUI and benchmarks (`@dortdb/showcase` and `@dortdb/benchmarks`). Finally, `@dortdb/dataloaders` contains utilities for importing data from various formats.

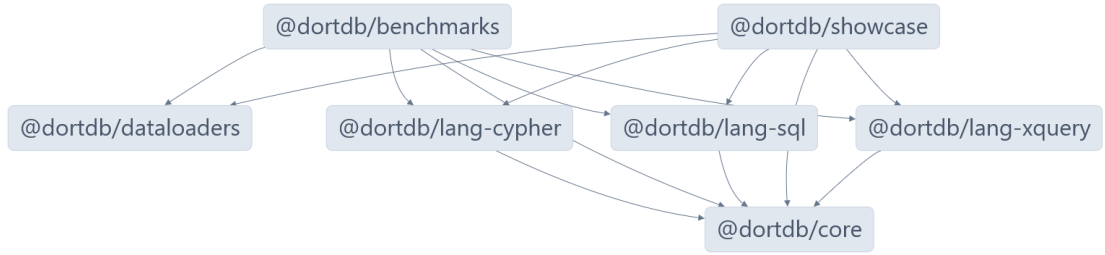


Figure 8.1 Dependency graph of the monorepository.

8.1 Core Library

The core library serves as a common framework for registering data sources, querying, and optimization. Each language is responsible for converting a text query into a logical plan. The plan operator tree implements an extended version of the visitor design pattern [27]. In order for each language to be able to expand the plan algebra with its own operators, or to customize the behavior defined for a common operator, the design pattern is modified to take into account the language that instantiates each plan operator, as seen in Figure 23. The `accept` method receives a dictionary of visitors instead of a single visitor. The core library provides a default visitor implementation for each phase of the query process. Individual languages may provide their own implementations, typically by extending the original class. The `accept` and `visit` methods also optionally take a second argument, which may be necessary to propagate information during the downward pass of the plan tree, for example, the existing context during plan execution.

8.1.1 Language Manager

Each language can define its own visitors, callables, or aggregates. Callables and aggregates may also be provided as an extension, not scoped to any specific language. For example, a datetime extension might define operators and functions for manipulating datetime values. The `LanguageManager` class then handles instantiating visitor dictionaries or finding aggregates or callables by name.

¹<https://nx.dev/>

Program 23 The extended visitor pattern used in the logical plan.

```
export interface PlanOperator {
  lang: Lowercase<string>;

  accept<Ret, Arg>(
    visitors: Record<string, PlanVisitor<Ret, Arg>>,
    arg?: Arg,
  ): Ret;

  // ...
}

export interface PlanVisitor<Ret, Arg = never> {
  visitRecursion(operator: Recursion, arg?: Arg): Ret;
  visitItemSource(operator: ItemSource, arg?: Arg): Ret;
  visitProjection(operator: Projection, arg?: Arg): Ret;
  // ...
}

export class ItemSource implements PlanOperator {
  constructor(
    public lang: Lowercase<string>,
    public name: ASTIdentifier | Aliased<ASTIdentifier>,
  ) {}

  accept<Ret, Arg>(
    visitors: Record<string, PlanVisitor<Ret, Arg>>,
    arg?: Arg,
  ): Ret {
    return visitors[this.lang].visitItemSource(this, arg);
  }

  // ...
}
```

8.2 Query Process

The rest of the library's key components will be outlined in the order they appear during query execution. Different languages may handle some steps differently. The key deviations will be noted when appropriate.

8.2.1 Lexer and Parser

First of all, the text query needs to be parsed into an *abstract syntax tree (AST)*. In order to do this, *parser generators* are used. Modern projects use solutions such as ANTLR,² where the code processing the parsed text is separated from the grammar description. This, while promoting cleaner code, unfortunately, results

²<https://www.antlr-ng.org/>

in large source files. We have initially used `ts-jison`³ instead, which is based on GNU Bison.⁴ Bison employs an older approach less compatible with modern software engineering practices. The resulting parser and its runtime, however, are significantly smaller (approximately 10 times). `ts-jison` produces LALR(1) grammars, which have turned out not to be powerful enough to fully describe the openCypher grammar. The grammar for `@dortdb/lang-cypher` therefore had to compromise in several areas.

Later, we migrated to Peggy.⁵ Peggy is based on *parsing expression grammar* (PEG) formalism, which enabled us to properly implement the Cypher parser. Peggy also produces slightly smaller source code, and, according to benchmarks,⁶ is about twice as fast as both `jison` and `ANTLR4`.

The parser is generated from a formal grammar file. Often, parser generators separate the parsing process into two modules. The *lexer* performs lexical analysis, converting the text query into tokens, such as specific keywords or strings. The actual parser then performs syntactic analysis and combines the tokens into grammar constructs such as a function call or a `SELECT` statement. Peggy does not differentiate between lexical and syntactic analysis and keeps everything in a single grammar file. The resulting AST nodes implement the visitor pattern, similarly to the logical plan.

For more detail on AST, see Appendix AST in More Detail.

8.2.2 Plan Builder

The AST is converted into a logical plan using a `PlanBuilder` visitor. It is possible to apply some optimizations while creating the plan, but the resulting plan is mostly a literal representation of the original query. Intermediary operators are converted into `Calculations` right at this stage, in a process more closely described in Section Calculation Building.

The SQL language generates the plan in two passes because the full schema of referenced data sources is known only after the query has been fully processed. The first pass builds the plan from the data sources downstream. The second pass then propagates the known operator schema back upstream.

The plan builder keeps track of the current context to determine which identifiers refer to variables (or columns, etc.) and which refer to data sources. Some languages (such as SQL) cannot determine the full context until the entire query is processed, including subqueries. If the subqueries belong to another language, the parent language needs a way to receive the information contained in the subquery. The context passed to the subquery, therefore, also includes information about data sources whose schemas are currently being inferred. The nested plan builder should, in addition to creating the plan, also generate a set of newly identified identifiers. See Program 24 for an example.

³<https://github.com/ericprud/ts-jison>

⁴<https://www.gnu.org/software/bison/>

⁵<https://peggyjs.org/>

⁶<https://chevrotain.io/performance/>

Program 24 The plan builder for XQuery receives context with `t1.attr1` and with `t1` set as a data source with not yet identified schema. It then correctly interprets `$t1:attr2` as a variable and `$Invoices` as a data source. The SQL plan builder then receives back information on `t1.attr2`.

```
SELECT attr1 FROM t1
WHERE EXISTS (
  LANG xquery
  $Invoices//[OrderId = $t1:attr2]
)
```

8.2.3 Optimizer et al.

The optimizer itself and its rules are covered in Section Implemented Optimizer Rules. They make use of multiple other plan visitors.

EqualityChecker A deep equality checker. It is possible to check the equality of plan operator trees regardless of their language or with specific renamings applied. This is helpful, for example, when checking possibly indexable expressions in a `Join` operator, whose input `TupleSources` are renamed with a `Projection`.

AttributeRenamer This visitor renames attributes in an operator tree. Necessary when, e.g., pushing down `Selections` below `Projections`.

AttributeRenameChecker It is not always possible to rename attributes without changing the meaning of a query. This visitor verifies that a specific renaming can be done. See Figure 19 for an example.

TransitiveDependencies This visitor extracts the dependencies of an operator tree, helping to identify correlated subqueries. As an example,

```
SELECT foo.x FROM foo WHERE bar.y > foo.z
```

depends on an external attribute `bar.y`. Used in many rules, among others, in `Selection` pushdown or `Projection` merging.

8.2.4 Variable Mapper

Attributes used in queries and their plans can generally consist of multiple components, for example `schema.table.column` or `$qualified:variable`. During query planning and optimization, the attributes are represented as lists of tokens. Sets or maps of attributes are then represented using tries made of nested hash tables. This is fine during planning, but if that were the case during execution and tuples were represented using the same tries, attribute lookups would become prohibitively expensive. To optimize this, the `VariableMapper` visitor renames attributes to numeric indices. During execution, tuples are then represented using simple arrays. The numeric attributes are reused when possible to keep the array sizes minimal.

8.2.5 Data Adapters

Different data models and different languages may make different assumptions about how the data is stored or represented. To decouple logical plans from data, languages can be configured with data adapters. Data adapters specify how data sources should be accessed. For example, the default XQuery data adapter allows `TreeStep` operators to traverse XML elements. It may, however, be desirable to use XQuery to query JSON data instead. Cypher queries target graphs, which can also be stored in multiple ways. Nodes might be JavaScript objects, and edges might be their properties. The graph might also be stored as a collection of relations, one for each node or edge type. The default Cypher data adapter uses Graphology [28] as the data backend. The SQL data adapter can be used to override how SQL accesses tuple attributes in relations.

Program 25 The XQuery data adapter interface

```
export interface XQueryDataAdapter<NodeType = any> {
  isNode(node: unknown): node is NodeType;
  treeStep(
    test: ASTItemType,
    axis: AxisType
  ): (node: NodeType) => NodeType[];
  createElement(
    ns: string,
    qname: string,
    content: unknown[]
  ): NodeType;
  createAttribute(ns: string, qname: string, content: string): NodeType;
  createComment(content: string): NodeType;
  createDocument(
    ns: string,
    qname: string,
    content: unknown[]
  ): NodeType;
  createNS(name: string, content: string): NodeType;
  createText(content: string): NodeType;
  createProcInstr(name: string, content: string): NodeType;
  addAttribute(el: NodeType, attr: NodeType): void;
  lookupPrefix(el: NodeType, ns: string): string;
  lookupNSUri(el: NodeType, prefix: string): string;
  atomize(value: unknown): unknown;
}
```

8.2.6 Executor

Finally, the logical plan is executed. Unlike other visitors, the `Executor` provided by the core library is abstract and not immediately usable. As discussed in previous sections, the tuple representations during execution will differ from the actual tuples in the source data. Each language should therefore extend the

Executor and provide its own implementation of the `visitTupleSource` method. That generally means interfacing with the language data adapter.

The default executor implementation heavily relies on iterators. This has a performance impact on smaller datasets, but on larger datasets, the difference becomes negligible. This is compensated for by memory efficiency. Unless an operator needs to load all of its inputs fully into memory (for example, if it is an **OrderBy** or a **GroupBy** operator), the operator produces values on demand lazily. The default executor implementation is fully synchronous and does not support user-defined functions that return promises (e.g., a **TupleFnSource** that would load a remote JSON document). The visitor pattern, however, makes it easy to implement another executor that adds this functionality.

The executor can be configured with index types usable in hash joins. In a process similar to detecting opportunities for index scans (see Section Secondary Indices), the index types themselves decide whether a **Join** condition is suitable for a hash join. If no index type can be used, the executor falls back to nested loop join.

After the query results are produced, tuple values must be serialized into a suitable representation. To this end, each language may specify a serializer function. The default serializer function joins attribute parts with periods and creates a JavaScript object for each tuple.

9 Usage

In the previous chapters, we have delved into the design decisions behind DortDB, its implementation, and its theoretical background. However, we have not yet described how to actually use it or extend it. This chapter will serve as a walkthrough through user interaction and as a concise manual for developers. It does not aim to be a full-fledged documentation of DortDB.

9.1 Showcase

DortDB Showcase is a GUI intended to demonstrate DortDB's capabilities. While it is possible to run the demo locally, the recommended way is to use an instance deployed to GitHub Pages at <https://filipjezek.github.io/dortdb>. The Showcase is a purely frontend single-page application written in Angular. The user can write queries and inspect their logical plans, experimenting with various optimizer rules. It is also possible to execute the queries on the UniBench sample dataset used in benchmarks in Section UniBench. The rest of this section will consist of a sequence of figures describing various parts of the user interface.

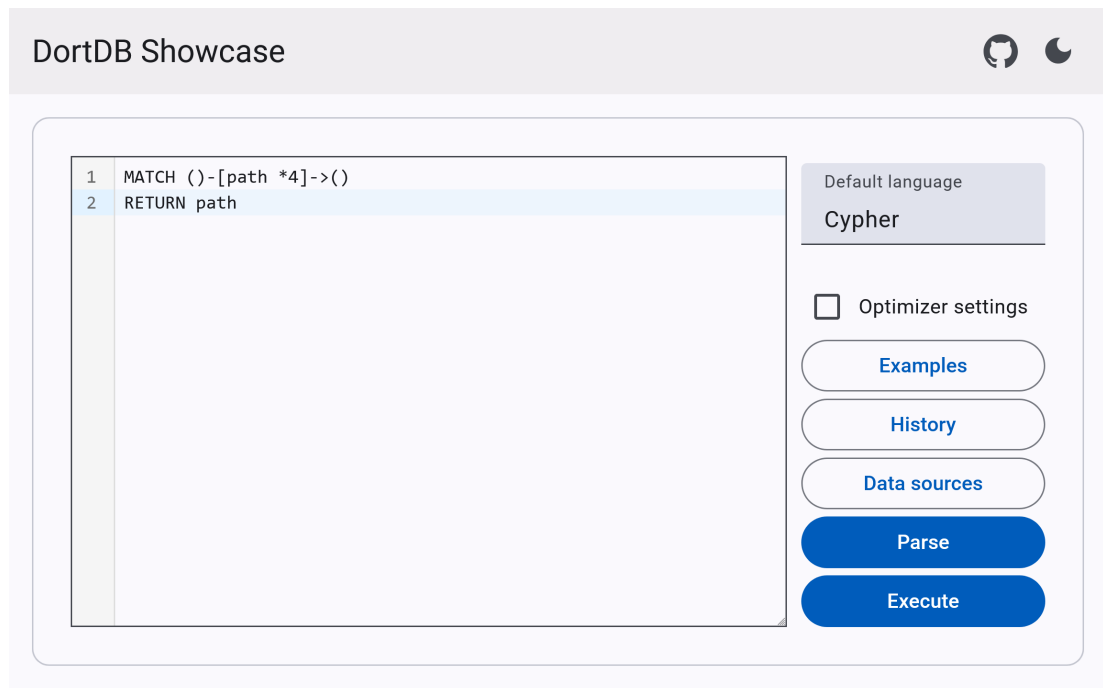


Figure 9.1 Showcase when loaded in the web browser. The application remembers all settings, including the current query, the optimizer settings, the default language, and query history.

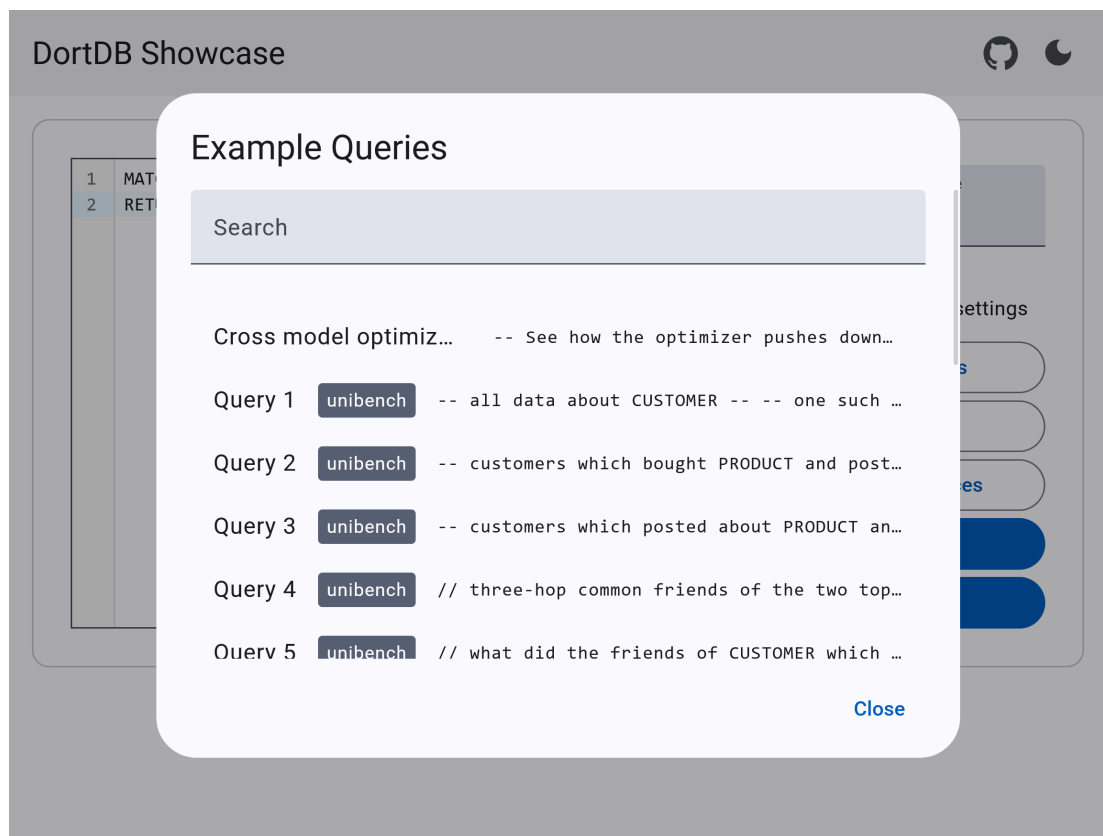


Figure 9.2 Showcase example queries. To help the user try out DortDB effectively, this dialog provides a quick way to find interesting examples.

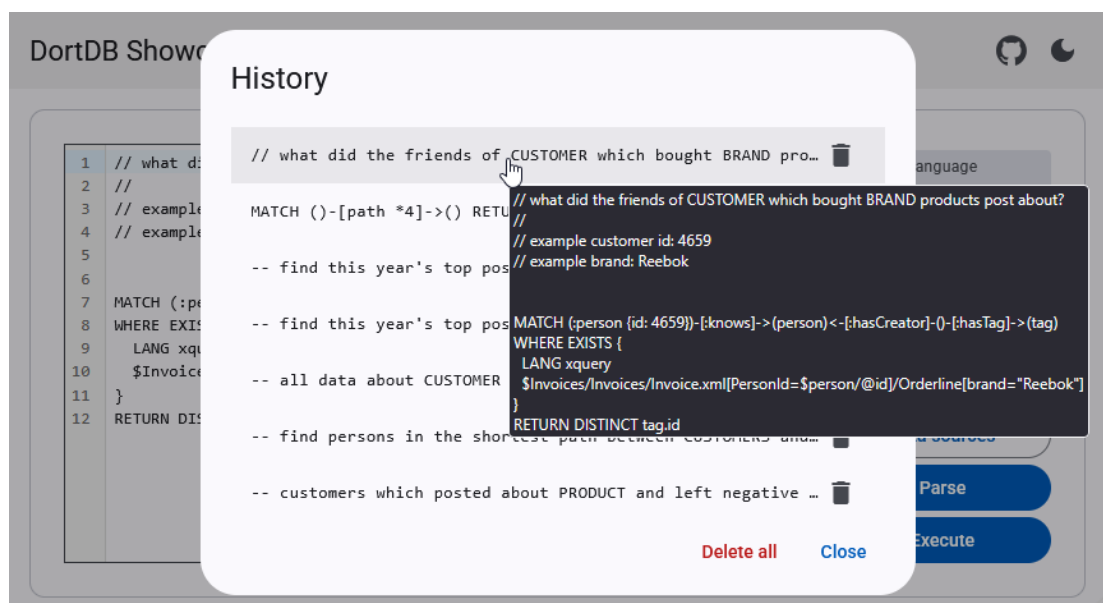


Figure 9.3 Showcase query history. History stores the last 20 parsed or executed queries.

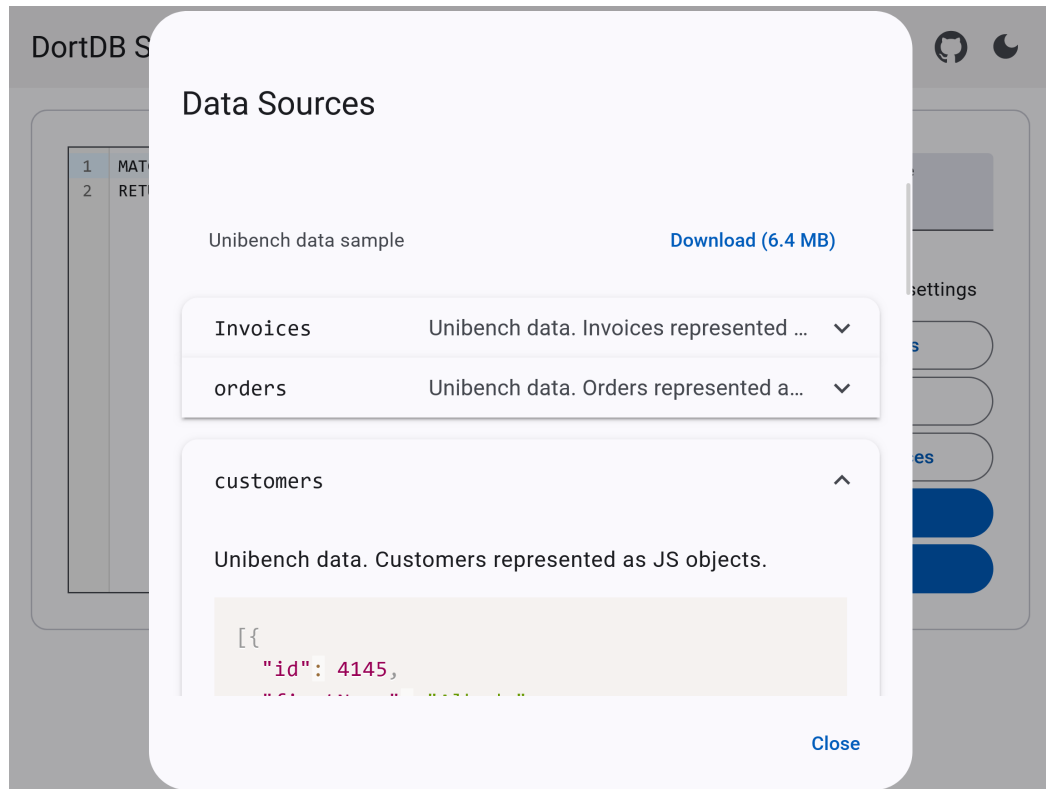


Figure 9.4 Showcase data sources. Showcase allows users to download a sample of the UniBench data and register it as data sources. The individual data sources are described in collapsible expansion panels, including information on any secondary indices. Once the user downloads the data, the dialog offers to save the parsed data into the web browser to avoid redownloading it when the page is refreshed.

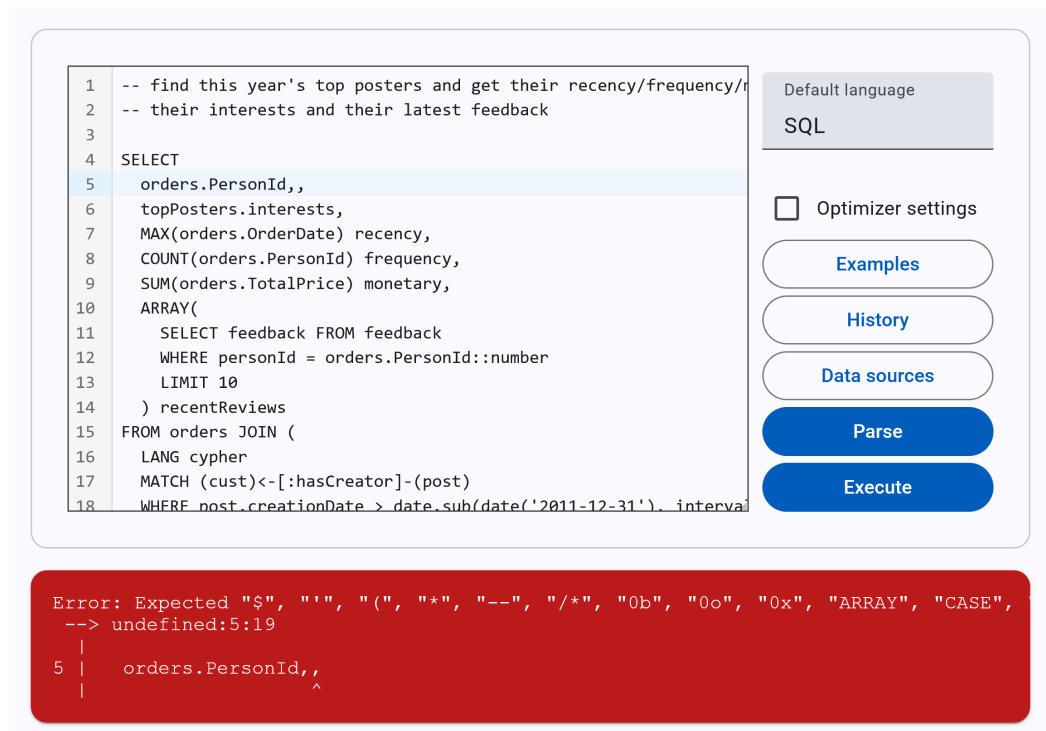


Figure 9.5 Query errors are displayed to the user.

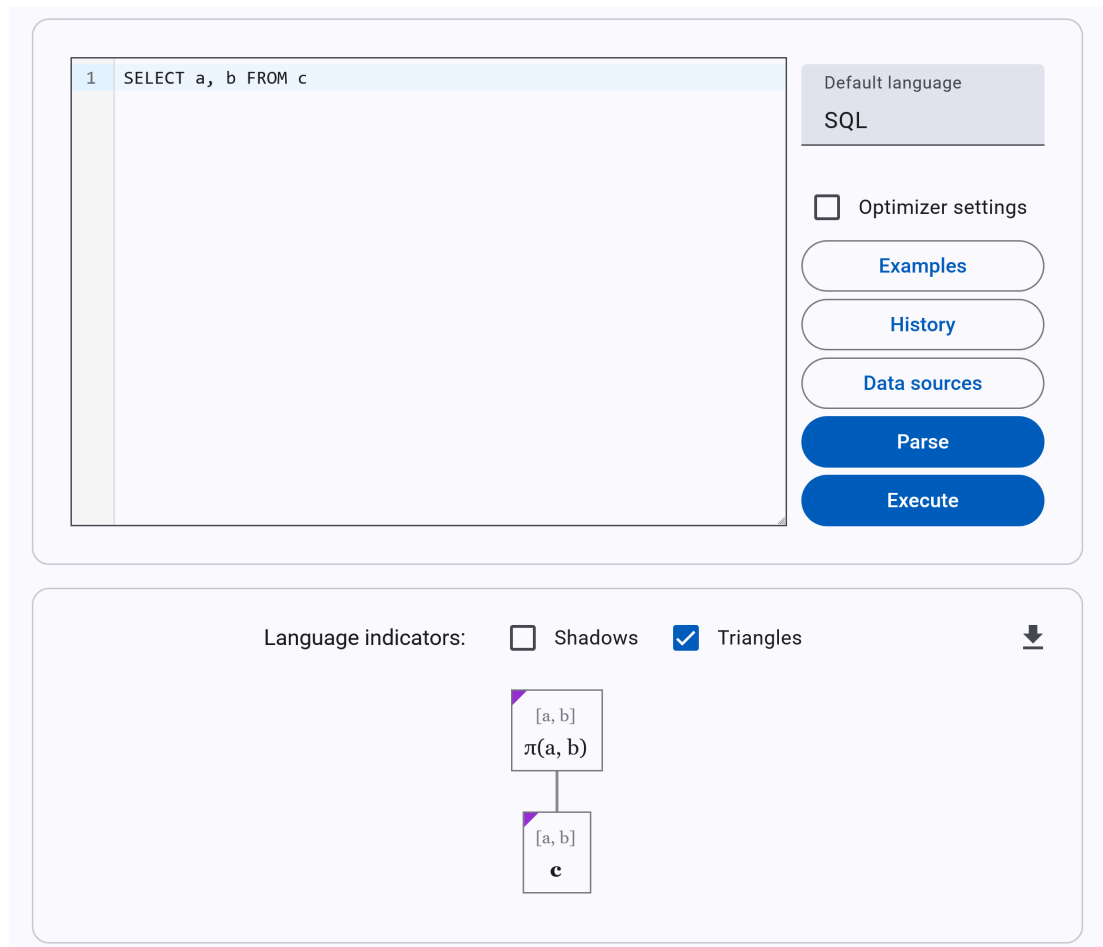


Figure 9.6 When the query is parsed, the logical plan is displayed as an interactive tree, similar to previous figures in this thesis. The tree can be zoomed and panned. It is possible to download the tree as an SVG image.

```

1 -- find this year's top posters and get their recency/frequency/r
2 -- their interests and their latest feedback
3
4 SELECT
5   orders.PersonId,
6   topPosters.interests,
7   MAX(orders.OrderDate) recency,
8   COUNT(orders.PersonId) frequency,
9   SUM(orders.TotalPrice) monetary,
10  ARRAY(
11    SELECT feedback FROM feedback
12    WHERE personId = orders.PersonId::number
13    LIMIT 10
14  ) recentReviews
15 FROM orders JOIN (
16   LANG cypher
17   MATCH (cust)<-[:hasCreator]-(post)
18   WHERE post.creationDate > date.sub(date('2011-12-31'), interval

```

Default language

SQL

☐ Optimizer settings

Examples

History

Data sources

Parse

Execute

☒ Fixed height rows

orders.PersonId	topPosters.inte	recency	frequency	monetary	recentReviews
10664	163 697 810 065	2024-07-10	11	17729.77	'1.0,I would have chosen
15393162793476	39 163 216 414 1007	2024-08-24	11	15693.72000000	'5.0,This is a very sturdy,
10027	163 201 005 016	2024-07-10	24	33031.72999999	'1.0,This might be ok

Figure 9.7 The results of the query execution are displayed in a paginated table. The table columns have adjustable widths.

9.2 Programmatic Usage

DortDB can be installed from NPM:¹

Program 26 Installation of the JavaScript packages.

```
npm i @dortdb/core
# installation of language packages
npm i @dortdb/lang-cypher @dortdb/lang-sql @dortdb/lang-xquery
```

The main class of the DortDB framework is simply called `DortDB`. It serves as the container for languages and extensions and exposes methods for registering data sources or querying. When creating a `DortDB` instance, it is necessary to provide at least one language. To facilitate better tree shaking, the optimizer receives no rules by default. However, all currently available rules are also exported as a convenient single array, already ordered for best performance. The individual languages can also be configured, for example, to use a different data adapter. The DortDB configuration can also specify additional user-defined functions, operators, or aggregates. It is also possible to bundle functions, operators, and aggregates into a single importable object called an extension. DortDB already comes with a `datetime` extension for manipulating dates and times. Extensions can be scoped to specific languages only. Only some languages support user-defined operators, for example, SQL.

Once a `DortDB` instance is created, data sources can be registered. The registration simply pairs a data structure with an identifier, an $\mathcal{O}(1)$ operation. It is then possible to index the structure with secondary indices. The developer needs to specify what type of index is used.

After the database is fully configured, we can parse queries into an AST, build and optimize logical plans, and execute them. Both the AST and the logical plan implement the visitor design pattern, so additional preprocessing can be added. All of the logical plan operators also expose a `getChildren()` method, so it is possible to quickly and simply iterate through the whole operator tree.

9.3 Extensibility

One of DortDB's main selling points is its extensibility. It is possible (and encouraged) to provide more languages, index types, optimizer rules, or extensions. This section aims to outline, in broad steps, how to do all of the above.

9.3.1 Authoring a New Language

First of all, it is necessary to implement a parser that parses language queries into an AST. The parser must recognize language switches, switch to a different parser, and then resume parsing afterward. To this end, each parser must recognize when to end even if there is yet remaining input, be it because of encountered `LANG EXIT` keywords or because the parser would exit its initial scope. The parser

¹www.npmjs.com/

Program 27 DortDB initial setup.

```
import { datetime, DortDB, MapIndex } from '@dortdb/core';
import { defaultRules } from '@dortdb/core/optimizer';
import { SQL } from '@dortdb/lang-sql';
import { DomDataAdapter, XQuery } from '@dortdb/lang-xquery';
import { Cypher } from '@dortdb/lang-cypher';

const db = new DortDB({
  mainLang: SQL(),
  additionalLangs: [
    XQuery({
      // in browser environment
      adapter: new DomDataAdapter(document)
    }),
    Cypher({ defaultGraph: 'defaultGraph' })
  ],
  optimizer: {
    // it is possible to use only specific optimizer rules
    rules: defaultRules,
  },
  executor: {
    // these indices will be available for hash joins
    hashJoinIndices: [MapIndex]
  },
  extensions: [
    datetime,
    {
      scope: ['cypher'],
      aggregates: [
        // ... aggregates only for Cypher
      ],
    },
  ],
});
```

should return both the parsed AST and any remaining unparsed input, so that an outer parser could continue, as in Program 31.

After the AST is available, the language should build an initial logical plan. If the unified algebra does not suffice, the language can define its own plan operators. It must, however, extend all core visitors to take these new operators into account. When building the plan, the language will receive an `IdSet` with identifiers defined in the outer context. Some of the identifiers in the `IdSet` may have the `Symbol(toInfer)` as their last token. This means that the identifier belongs to a data source whose schema is still being inferred. If the language encounters any matching identifiers, it should record them and return them in addition to the built logical plan.

Program 28 DortDB data sources registration.

```
import { MapIndex } from '@dortdb/core';
import { MultiDirectedGraph } from 'graphology';

// db is already initialized
const arr = [1, 2, 3];
const obj = { prop: 'value' };
const objArr = [obj];

// the identifiers can have multiple parts
db.registerSource(['testing', 'arr'], arr);
db.registerSource(['testing', 'objArr'], objArr);
// anything can be registered, it is up to
// the language data adapters to use it
db.registerSource(['testing', 'obj'], obj);
db.registerSource(['defaultGraph'], new MultiDirectedGraph());

// a simple hash index
// some indices may support indexing multiple expressions
db.createIndex(['testing', 'objArr'], ['prop'], MapIndex);
// the indexed expression is parsed with the default language
// but this can be changed by passing in a config parameter
db.createIndex(['testing', 'objArr'], ['prop->2'], MapIndex);

// for data sources that should be interpreted as item sources
// a specific identifier is configured to be treated as the item
// while parsing the expression
db.createIndex(['defaultGraph', 'nodes'], ['x.id'], MapIndex, {
  mainLang: 'cypher',
  fromItemKey: ['x'],
});
```

Program 29 The query optimizer can be reconfigured at any time.

```
import { UnnestSubqueries } from '@dortdb/core/optimizer';

db.optimizer.reconfigure({
  rules: [UnnestSubqueries]
});
```

9.3.2 Authoring a New Index Type

DortDB features secondary indices. The index must implement the following three methods: **match**, which receives an array of expressions and decides whether any match the index. **CreateAccessor**, which takes an array of expressions that match the index, and should return a **Calculation** that accesses the index and finds matching items for specific values of the expressions. Finally, there is the **reindex** method, which receives items from the connected data source and values

Program 30 Querying the database.

```
import { ASTNode } from '@dortdb/core';
import * as plan from '@dortdb/core/plan';

const query = 'SELECT a FROM t';

// some languages allow multiple statements
// per one query, so the result is an array
const ast: ASTNode[] = db.parse(query);

const plan = db.buildPlan(ast.at(-1));

// if we wanted to invert `orderBy` directions, we
// could write a visitor, or do it the quick way:
const stack = [plan];
while (stack.length) {
  const current = stack.pop();
  stack.push(...current.getChildren());
  if (current instanceof plan.OrderBy) {
    for (const o of current.orders) {
      o.ascending = !o.ascending;
    }
  }
}

const results = db.executePlan(plan);

// during the relevant phases, the default language
// can be changed, and bound parameters can be provided

// we can also just query without caring about the plan or AST
const newResults = db.query(`
  MATCH (a)-[]->(b)
  WHERE a.id > $id
  RETURN a, b
`, {
  mainLang: 'cypher',
  boundParams: { id: 13 }
});
```

of the indexed expressions, and should fill the index data structure.

The index does not always need to store data in any data structure. For example, the Cypher `ConnectionIndex` detects join conditions between nodes and edges, and then selects connected edges or nodes using the Cypher data adapter.

If the index type is to be used in hash joins during query execution, it must additionally provide access to all stored values via the method `allValues`. This is necessary for evaluating full outer joins. Additionally, it must implement the `canIndex` static method, which acts similarly to the `match` method.

Program 31 The nested language should recognize the closing parenthesis as a scope exit and terminate. It should return both the parsed AST and the remaining input string `) AS t1 WHERE attr1 > 3 .`.

```
SELECT t1.attr1 FROM (  
  LANG newlang  
  -- new language here...  
) AS t1 WHERE attr1 > 3
```

Program 32 When extending the unified algebra, all core visitors must be extended as well. This example is taken from `@dortdb/lang-xquery`.

```
import { AttributeRenamer, PlanVisitor } from '@dortdb/core';  
import { TreeJoin, XQueryPlanVisitor } from '../plan/index.js';  
import { RenameMap } from '@dortdb/core/plan';  
  
export class XQueryAttributeRenamer  
  extends AttributeRenamer  
  implements XQueryPlanVisitor<void, RenameMap>  
{  
  // ...  
  visitTreeJoin(operator: TreeJoin, renames: RenameMap): void {  
    operator.source.accept(this.vmap, renames);  
    this.processItem(operator, 'step', operator.dependencies, renames);  
  }  
}  
  
// in language.ts:  
  
export function XQuery(config?: XQueryConfig): XQueryLanguage {  
  return {  
    name: 'xquery',  
    // ...  
    visitors: {  
      attributeRenamer: XQueryAttributeRenamer,  
      // ...  
    },  
  };  
}
```

9.3.3 Authoring Other Components

Besides entire languages or specialized secondary indices, DortDB can also be extended with new optimizer rules. This may be especially relevant for custom logical plan operators. The rules may be as simple as removing neighboring `MapFromItem` and `MapToItem` operators, or complex algorithms for merging `Projections`. In all cases, they can be reduced to two methods: `match`, which recognizes patterns in the logical plan, and `transform`, which replaces those patterns with others.

Finally, DortDB users may define their own functions or aggregates. These may be grouped into and distributed as extensions, for example, for spatial data manipulation or for the creation of dynamic data sources.

10 Future Work

The developed software is a solid foundation for multimodel queries in the web browser. However, there are many areas for improvement. The current implementation is the result of approximately two years of development, but there are only so many features we can implement in the timeframe. The following are key directions for further efforts.

10.1 Language Improvements

The currently implemented languages are not fully complete; the missing features are detailed in the relevant sections of Chapter Implemented Languages. We consider the most pressing addition to be support for wildcard attribute selection using `SELECT *` in SQL. As DortDB aims to be approachable for both users and developers, the ability to quickly inspect all columns of a table without needing to recall the schema would be a meaningful quality-of-life improvement. The main drawback is that wherever the asterisk appears in a **Projection**, the exact schema of the resulting tuples cannot be statically determined, likely requiring certain optimizer rules to be disabled in those cases. Beyond this, shortest-path matching currently lacks adequate support, with only a very basic approach in place.

Besides the languages we already have, DortDB would definitely benefit from more. There is currently no language specifically designed for JSON data, though it is possible to use XQuery with a suitable data adapter or SQL with JSON operators. Furthermore, it would be interesting to implement and test academia-driven languages such as MMQL [13], which aims to universally query any data model.

10.2 Additional Optimization

While we have provided several optimizations as a starting point, the benchmarks show that there are areas where they fall short. The graph model optimizations are currently focused primarily on faster graph joins and do not modify the direction or patterns of the graph traversal itself. As an example, in UniBench Query 4.1, there is a recursive graph traversal pattern:

```
MATCH (:person {id: toptwo[0]})-[:knows *..3]->(foaf)
<-[:knows *..3]-({id: toptwo[1]})}
RETURN foaf
```

DortDB executes this query as a node lookup, followed by two recursions. Even though filters are applied throughout, the number of nodes in memory grows exponentially through up to six steps. The same result could be achieved much faster if we split the graph traversal into two, beginning at the two ends of the original pattern, and intersect the results, like in the optimized version of the same query.

Moreover, DortDB currently cannot index XQuery expressions such as this:

```
$Invoices/invoice.xml[OrderId eq 13]
```

There is also no `Join` reordering optimizer rule, although it is debatable whether it would be viable, given that DortDB collects no statistics on the registered data sources.

10.3 New Features

Currently, DortDB suffices for general data queries. Nevertheless, it could be extended with entirely new features. One example is an asynchronous query executor. That would enable users to add user-defined functions that return promises, e.g., to fetch remote resources or query databases over the network. DortDB could also be extended with data manipulation capabilities, allowing the creation, update, or deletion of data. The combination of asynchronicity and data manipulation could be a use case for a transaction manager, even though JavaScript makes it hard to encounter race conditions.

Conclusion

In this thesis, we have designed and implemented DortDB, a TypeScript in-memory database. We have also introduced the unified algebra, which can describe relational, document, and graph queries. The unified algebra enabled us to implement the concept of multilanguage queries, in which the most suitable or intuitive query language is used for different parts of the query. The unified algebra is extensible, as we have demonstrated on the XQuery language. This opens the door to incorporating additional data models in the future, if they cannot already be described.

The database itself serves as a framework providing common building blocks and a query environment for separately developed query languages. We have implemented three languages: SQL for the relational model, XQuery for the document model, and Cypher for the graph model. The languages can be further extended with user-defined functions, operators, or aggregates. We have exposed both the abstract syntax trees made of parsed queries and the logical plans as a public API for developers using DortDB. The DortDB is entirely modular and has a small footprint on webpage size when used on the web. The core framework in combination with the SQL language, with all current features included, weighs 202 KB with no compression.¹

We have also developed a rule-based query optimizer. Thanks to the unified algebra, the optimizer can optimize across language boundaries. It is configurable and extensible by providing further optimization rules. The implemented rules include subquery unnesting rules, selection pushdown, and projection merging. Among other DortDB optimizations is a common interface for implementing secondary indices. The secondary indices can be specialized for various scenarios, such as range, hash, or spatial indexes. The indices themselves decide when they can be used. As proof of concept, we have provided a simple hash table index.

We then demonstrated DortDB’s capabilities by matching it against two well-established JavaScript in-memory SQL databases and two well-established multimodel databases. DortDB proved the power of its optimizer by significantly outperforming AlaSQL on the TPC-H benchmark, while coming reasonably close to SQL.js. The comparison with ArangoDB and OrientDB did not go over so well, highlighting gaps in optimizing graph and document queries. Even then, the concepts behind DortDB seem like a viable approach, and the benchmarks helped us identify directions for further development.

DortDB functionality can be explored through the Showcase, a GUI developed alongside the system for experimentation with logical plans. Users can query data, inspect logical plans, and interact with the query optimizer. A demo paper [18] presenting DortDB and the DortDB Showcase, coauthored with Pavel Koupil, Michal Kopecký, Jáchym Bártík, and Irena Holubová, was presented at the 51st VLDB conference. A more substantial journal article is currently in preparation.

Looking back at the goals defined at the outset of this thesis: the modular architecture and tree-shaking-friendly decomposition of DortDB into decoupled

¹In contrast, AlaSQL weighs 499KB, while SQL.js weighs 684KB. Gzipped versions of DortDB with SQL, AlaSQL, and SQL.js weigh 61KB, 109KB, and 333KB, respectively. Note that the gzipped size varies based on other JavaScript content in the final bundle.

units addresses G1; the implementation of SQL, XQuery, and Cypher, each suited to a different data model, as well as the extensibility of the algebra itself fulfills G2; the support for user-defined functions, operators, aggregates, and custom index types satisfies G3; the exposure of both abstract syntax trees and logical plans as a public API meets G4; and finally, the rule-based query optimizer together with the secondary index infrastructure realizes G5.

Bibliography

1. STATCOUNTER. *Platform Market Share Worldwide*. 2025. Available also from: <https://gs.statcounter.com/platform-market-share/desktop-mobile-tablet>. Accessed: 2026-04-03.
2. HTTP ARCHIVE. *Page Weight Report – March 2026*. 2026. Available also from: https://httparchive.org/reports/page-weight?start=2026_03_01. Accessed: 2026-04-03.
3. DUGGAN, Jennie; ELMORE, Aaron J; STONEBRAKER, Michael; BALAZINSKA, Magda; HOWE, Bill; KEPNER, Jeremy; MADDEN, Sam; MAIER, David; MATTSON, Tim; ZDONIK, Stan. The BigDAWG polystore system. *ACM Sigmod Record*. 2015, vol. 44, no. 2, pp. 11–16.
4. LU, Jiaheng; HOLUBOVÁ, Irena. Multi-model databases: a new journey to handle the variety of data. *ACM Computing Surveys (CSUR)*. 2019, vol. 52, no. 3, pp. 1–38.
5. CODD, Edgar F. A relational model of data for large shared data banks. *Communications of the ACM*. 1970, vol. 13, no. 6, pp. 377–387.
6. GUO, Qingsong; ZHANG, Chao; ZHANG, Shuxun; LU, Jiaheng. Multi-Model Query Languages: Taming the Variety of Big Data. *J. of Distributed and Parallel Databases*. 2023, vol. 42, pp. 1–41.
7. ISO. *ISO/IEC 9075-14:2011 Information technology – Database languages – SQL – Part 14: XML-Related Specifications (SQL/XML)*. 2011.
8. ISO. *ISO/IEC TR 19075-6:2017 Information technology — Database languages — SQL — Part 6: SQL support for JavaScript Object Notation (JSON)*. 2017.
9. ISO. *ISO/IEC TR 9075-16:2023 Information technology — Database languages — SQL — Part 16: Property Graph Queries (SQL/PGQ)*. 2023.
10. JINGJING WANG ET AL. The Myria Big Data Management and Analytics System and Cloud Services. In: *Proc. of CIDR 2017*. www.cidrdb.org, 2017.
11. HAUSENBLAS, Michael; NADEAU, Jacques. Apache Drill: Interactive Ad-Hoc Analysis at Scale. *J. of Big Data*. 2013, vol. 1, no. 2, pp. 100–104.
12. SANFILIPPO, S.; NOORDHUIS, P. *Redis* [online]. 2009. [visited on 2025-03-15]. Available from: <http://web.archive.org/web/20080207010024/http://www.808multimedia.com/winnt/kernel.htm>.
13. KOUPIL, Pavel; CRHA, Daniel; HOLUBOVÁ, Irena. A universal approach for simplified redundancy-aware cross-model querying. *Inf. Syst.* 2025, vol. 127, p. 102456. Available from DOI: 10.1016/J.IS.2024.102456.
14. OPENLINK VIRTUOSO. *SPARQL Inline in SQL*. 2025. Available also from: <https://docs.openlinksw.com/virtuoso/rdfsparqlinline/>. Accessed: 2025-06-19.
15. KOLEV, Boyan; BONDIOMBOUY, Carlyna; VALDURIEZ, Patrick; JIMÉNEZ-PERIS, Ricardo; PAU, Raquel; PEREIRA, José. The CloudMdsQL Multistore System. *Proc. of SIGMOD 2016*. 2016, pp. 2113–2116.

16. CRHA, Daniel. Unified Querying of Multi-Model Data. 2023.
17. DEUTSCH, Alin; FRANCIS, Nadime; GREEN, Alastair; HARE, Keith; LI, Bei; LIBKIN, Leonid; LINDAAKER, Tobias; MARSAULT, Victor; MARTENS, Wim; MICHELS, Jan; MURLAK, Filip; PLANTIKOW, Stefan; SELMER, Petra; REST, Oskar van; VOIGT, Hannes; VRGOČ, Domagoj; WU, Mingxi; ZEMKE, Fred. Graph Pattern Matching in GQL and SQL/PGQ. In: *Proceedings of the 2022 International Conference on Management of Data*. Philadelphia, PA, USA: Association for Computing Machinery, 2022, pp. 2246–2258. SIGMOD '22. ISBN 9781450392495. Available from DOI: 10.1145/3514221.3526057.
18. JEŽEK, Filip; KOUPIL, Pavel; KOPECKÝ, Michal; BÁRTÍK, Jáchym; HOLUBOVÁ, Irena. DortDB: Bridging Query Languages for Multi-Model Data Ponds. *Proceedings of the VLDB Endowment*. 2025, vol. 18, no. 12, pp. 5335–5338.
19. ISO. *ISO/IEC 9075-1:2023*. 2023. Available also from: <https://www.iso.org/standard/76583.html>. Accessed: 2025-07-15.
20. SCHEK, H-J; SCHOLL, Marc H. The relational model with relation-valued attributes. *Information systems*. 1986, vol. 11, no. 2, pp. 137–147.
21. RE, C.; SIMEON, J.; FERNANDEZ, M. A Complete and Efficient Algebraic Compiler for XQuery. In: *22nd International Conference on Data Engineering (ICDE'06)*. 2006, pp. 14–14. Available from DOI: 10.1109/ICDE.2006.6.
22. ANGLES, Renzo; BONIFATI, Angela; GARCÍA, Roberto; VRGOČ, Domagoj. Path-based Algebraic Foundations of Graph Query Languages. *arXiv preprint arXiv:2407.04823*. 2024.
23. MIHÁL, Filip; BÁRTÍK, Jáchym; KOUPIL, Pavel. Refining storage strategy through index selection methods in multi-model database systems: A survey. *Data Knowl. Eng.* 2026, vol. 164, p. 102582. Available from DOI: 10.1016/J.DATAK.2026.102582.
24. HÖLSCH, Jürgen; GROSSNIKLAUS, Michael; SCHOLL, Marc H. Optimization of nested queries using the nf2 algebra. In: *Proceedings of the 2016 International Conference on Management of Data*. 2016, pp. 1765–1780.
25. ZHANG, Chao; LU, Jiaheng; XU, Pengfei; CHEN, Yuxing. UniBench: a benchmark for multi-model database management systems. In: *Technology conference on performance evaluation and benchmarking*. Springer, 2018, pp. 7–23.
26. ZHANG, Chao; LU, Jiaheng. Holistic evaluation in multi-model databases benchmarking. *Distributed and Parallel Databases*. 2021, vol. 39, no. 1, pp. 1–33.
27. JOHNSON, Ralph; VLISSIDES, John. Design patterns. *Elements of Reusable Object-Oriented Software Addison-Wesley, Reading*. 1995, vol. 20.
28. PLIQUE, Guillaume. *Graphology, a robust and multipurpose Graph object for JavaScript*. Zenodo, 2025. Version 0.26.0. Available from DOI: 10.5281/zenodo.14835805.

List of Figures

1.1	Event loop diagram. <i>Ilya Kantor</i> . ²	11
1.2	Relational model.	13
1.3	Document model.	13
1.4	Graph model.	14
1.5	Key-value model.	14
3.1	Data import and query evaluation workflow in DortDB [18].	23
4.1	Labeled property graph	29
5.1	SQL schema inference example.	33
5.2	Operator context example.	35
5.3	The different edge types: TupleSource table1 is created once while the Calculation is reevaluated for each source row. The Calculation edge points directly to its representative attribute xplusthree shown in blue color.	42
5.4	A subquery can be expressed as a Calculation with a Projection as an argument. Alternatively, the optimizer could convert the nested Projection to an outer ProjectionConcat . Note that this is not yet fully optimized. Because the nested Projection does not depend on table1 context, the ProjectionConcat can be further converted to a left outer Join .	43
5.5	Example of a complex SQL aggregation. The FILTER clause is defined in SQL:2003 as an optional feature. It is implemented in PostgreSQL and SQLite.	45
5.6	This Cypher query first selects all nodes from a graph. It then selects all of their outgoing and incoming edges using IndexScans . These edges are recursively expanded and connected into four-segment paths from both directions with the BidirectionalRecursion operator. The final Selection removes paths with duplicate edges created by joining the forward and backward subpaths.	46
5.7	One possible application for the NullSource operator.	46
6.1	Example of cross-model optimization.	50
6.2	The registered index recognizes the equality check on the expression it was registered with. Plan before and after the optimization. For more details, see optimizer rules Join Indices and Index Scans.	53
6.3	An example of an unnested subquery in a Projection . The resulting ProjectionConcat is outer.	54
6.4	An example of the two possible directions of merging MapFromItem and MapToItem operators.	55
6.5	Two examples of Selection pushdown.	57
6.6	A non-correlated non-outer subquery can be converted to a regular CartesianProduct .	58

6.7	The Selection cannot be pushed down, because it depends on attributes from both branches of the CartesianProduct . The CartesianProduct can be, however, converted to a Join operator.	59
6.8	The simplest possible case of the join indices rule. There is an index on Op2 that matches the condition Calculation	59
6.9	Example of a filtered TupleSource replaced by a IndexScan . The newly created Calculation is an index accessor that serves as an input for the underlying index data structure.	60
6.10	Example of two merged Projections	61
7.1	Schema of the UniBench data model [26].	63
7.2	Processing time for data import.	65
7.3	Average execution time for each query.	65
7.4	TPC-H benchmark schema. Taken from the TPC-H specification. ³	66
7.5	Average execution time for each query.	68
8.1	Dependency graph of the monorepository.	70
8.2	Data flow through the query process	76
9.1	Showcase when loaded in the web browser. The application remembers all settings, including the current query, the optimizer settings, the default language, and query history.	77
9.2	Showcase example queries. To help the user try out DortDB effectively, this dialog provides a quick way to find interesting examples.	78
9.3	Showcase query history. History stores the last 20 parsed or executed queries.	78
9.4	Showcase data sources. Showcase allows users to download a sample of the UniBench data and register it as data sources. The individual data sources are described in collapsible expansion panels, including information on any secondary indices. Once the user downloads the data, the dialog offers to save the parsed data into the web browser to avoid redownloading it when the page is refreshed. . .	79
9.5	Query errors are displayed to the user.	79
9.6	When the query is parsed, the logical plan is displayed as an interactive tree, similar to previous figures in this thesis. The tree can be zoomed and panned. It is possible to download the tree as an SVG image.	80
9.7	The results of the query execution are displayed in a paginated table. The table columns have adjustable widths.	81
9.8	Showcase optimizer settings. The optimizer can be completely turned off. It is also possible to disable individual rules or to reorder them using drag & drop.	82

A UniBench Queries

The following are the DortDB versions of the UniBench [25] queries.

Query 1 For a given **CUSTOMER**, find their profile, orders, feedback, and posts.

```
SELECT
  ROW(
    customers.id AS id,
    customers.firstName AS firstName,
    customers.lastName AS lastName
  ) profile,
  ARRAY(SELECT ROW(
    orders.OrderId as orderId,
    orders.Orderline AS orderline,
    orders.TotalPrice AS totalPrice
  ) FROM orders WHERE PersonId = customers.id) orders,
  ARRAY(SELECT ROW(
    feedback.productAsin AS asin,
    feedback.feedback AS feedback
  ) FROM feedback WHERE personId = customers.id) feedback,
  ARRAY(
    LANG cypher
    MATCH ({id: customers.id})<-[:hasCreator]-(post)
    RETURN post
  ) posts
FROM customers
WHERE id = :customerId
```

Query 2 For a given **PRODUCT**, find the persons who had bought it and posted on it.

```
SELECT id, firstName FROM customers
WHERE EXISTS (
  LANG cypher
  MATCH (:person {id: customers.id})<-[:hasCreator]-(post)-[:hasTag]->({id: $productId})
  RETURN 1
) AND EXISTS (
  SELECT 1 FROM orders
  WHERE PersonId = customers.id AND EXISTS (
    SELECT 1 FROM unwind(orders.Orderline) orderline
    WHERE productId = :productId
  )
)
```

Query 3 For a given **PRODUCT**, find persons who have commented and posted on it, and detect negative sentiments from them.

```
SELECT customers.id, feedback.feedback, products.productId
FROM customers
JOIN feedback ON customers.id = feedback.personId
JOIN products ON feedback.productId = products.asin
WHERE products.productId = :productId
AND (feedback.feedback[1]::number < 3
AND EXISTS (
  LANG cypher
  MATCH ({id: customers.id})<-[:hasCreator]-(post)-[:hasTag]->
    ({id: products.productId})
  RETURN 1
)
```

Query 4.1 Find the top-2 persons who spend the highest amount of money in orders. Then for each person, traverse their knows-graph with 3-hop to find the friends, and finally return the common friends of these two persons. This is a naive version with subpar performance.

```
UNWIND (
  LANG sql
  SELECT PersonId::number
  FROM orders
  GROUP BY PersonId
  ORDER BY sum(TotalPrice) DESC
  LIMIT 2
) AS toptwo
WITH collect(toptwo) AS toptwo
MATCH (:person {id: toptwo[0]})-[:knows *..3]->(foaf)
  <-[:knows *..3]-({id: toptwo[1]})
RETURN foaf
```

Query 4.2 A more optimized version of Query 4.1.

```
WITH toptwo(results) AS MATERIALIZED (
  SELECT ARRAY(
    SELECT PersonId::number
    FROM orders
    GROUP BY PersonId
    ORDER BY sum(TotalPrice) DESC
    LIMIT 2
  )
)
(
  LANG cypher
  WITH (LANG sql SELECT results[0] FROM toptwo) AS topfirst
  MATCH (:person {id: topfirst})-[:knows *..3]->(foaf)
  RETURN foaf
)
INTERSECT
(
  LANG cypher
  WITH (LANG sql SELECT results[1] FROM toptwo) AS topsnd
  MATCH (:person {id: topsnd})-[:knows *..3]->(foaf)
  RETURN foaf
)
```

Query 5 The query description given in the original paper is completely different from example implementations for ArangoDB, OrientDB, and AgensGraph that are part of the UniBench repository. The actual queries can be described as "what did the friends of **CUSTOMER** who bought **BRAND** products post about?"

```
MATCH (:person {id: $personId})-[:knows]->(person)<-[:hasCreator]-
()-[:hasTag]->(tag)
WHERE EXISTS {
  LANG xquery
  $Invoices/Invoices/Invoice.xml[
    PersonId=$person/@id
  ]/Orderline[brand=$param:brand]
}
RETURN DISTINCT tag.id
```

Query 6 Given **CUSTOMER 1** and **CUSTOMER 2**, find persons in the shortest path between them in the subgraph, and return the TOP 3 best sellers from all these persons' purchases.

```

SELECT x.value AS productId FROM (
  LANG xquery
  for $interPerson in (
    LANG cypher
    MATCH (:person {id: $customerId1})-[edges:knows*]-
      ({id: $customerId2})
    WITH [e in edges[1..-1] | [startNode(e), endNode(e)]] AS edges
    LIMIT 1 // recursion is BFS, so this is the shortest path
    UNWIND edges AS edge
    UNWIND edge AS person
    RETURN DISTINCT person.id
  ), $productId in $Invoices/Invoices/Invoice.xml[
    PersonId=$interPerson
  ]//productId
  group by $num := number($productId)
  order by fn:count($productId) descending
  return $num
) x
LIMIT 3

```

Query 7 For the products of a given **VENDOR** with declining sales, analyze the reviews for these items to see if there are any negative sentiments.

```

SELECT feedback.feedback FROM feedback
JOIN brandProducts
ON brandProducts.productAsin = feedback.productAsin
WHERE brandProducts.brandName = :brand
AND feedback.feedback[1]:number < 4
AND (
  LANG xquery
  for $interPerson in (
    let $now := date('2024-12-31') (: the data is static :)
    let $recent := $Invoices/Invoices/Invoice.xml[
      date(OrderByDate) gt date:sub($now, interval('6 months'))
    ][OrderByDate/asin = $brandProducts:productAsin]
    let $old := $Invoices/Invoices/Invoice.xml[
      date(OrderByDate) le date:sub($now, interval('6 months')) and
      date(OrderByDate) gt date:sub($now, interval('12 months'))
    ][OrderByDate/asin = $brandProducts:productAsin]
    return fn:count($recent) lt fn:count($old)
  )
)

```

Query 8 For all the products of a given **CATEGORY** during a given year, compute its total sales amount, and measure its popularity in the social media. The alternative no element version simply removes the outer SQL layer and instead of an XML element returns the sales amount.

```

SELECT x.value->'id' AS productId, x.value->'sales' AS sales,
x.value->'popularity' || '%' AS popularity
FROM (
  LANG xquery
  let $categoryProducts := (
    LANG sql
    SELECT products.productId
    FROM products
    JOIN brandProducts ON products.asin = brandProducts.productAsin
    JOIN vendors ON brandProducts.brandName = vendors.id
    WHERE vendors.Industry = :industry
  ),
  $yrAgo := date:sub(date('2024-12-31'), interval('1y')),
  (: there are no posts newer than 2012 in the dataset :)
  $postsYrAgo := date('2011-12-31')
  let $totalPosts := (
    LANG cypher
    UNWIND categoryProducts AS pid
    MATCH ({id: pid})<-[hasTag]-(p)
    WHERE p.creationDate > postsYrAgo
    RETURN count(p)
  )

  for $pid in $categoryProducts
  let $soldProducts := $Invoices/Invoices/Invoice.xml[
    date(OrderDate) gt $yrAgo
  ]/Orderline[productId eq $pid],
  $relatedPosts := (
    LANG cypher
    MATCH ({id: pid})<-[hasTag]-(p)
    WHERE p.creationDate > postsYrAgo
    RETURN count(p)
  )
  return <product
    id="{ $pid }"
    sales="{ sum($soldProducts/price/number()) }"
    popularity="{ $relatedPosts div $totalPosts * 100 }"
  />
) x

```

Query 9 Find top-3 companies who have the largest amount of sales at one **COUNTRY**, for each company, compare the number of the male and female customers, and return the most recent posts of them.

```

SELECT
  topVendors.id,
  (
    SELECT
      count(*) FILTER (WHERE gender = 'male') /
      count(*) FILTER (WHERE gender = 'female')
    FROM customers
    WHERE EXISTS (
      SELECT 1 FROM orders WHERE PersonId = customers.id
      AND Orderline @> ARRAY[ROW(topVendors.id AS brand)]
    )
  ) mfRatio,
  ARRAY(
    LANG cypher
    UNWIND (
      LANG sql
      SELECT products.productId
      FROM products
      JOIN brandProducts
      ON products.asin = brandProducts.productAsin
      WHERE brandProducts.brandName = topVendors.id
    ) AS productId
    MATCH ({id: productId})<-[:hasTag]-(post)
    RETURN post
    ORDER BY post.creationDate DESC LIMIT 5
  ) latestPosts
FROM (
  SELECT
    vendors.id,
    (
      LANG xquery
      let $sales := $Invoices/Invoices/Invoice.xml
        /Orderline[brand = $vendors:id]
      return fn:count($sales)
    ) sales
  FROM vendors
  WHERE Country = :country
  ORDER BY sales DESC
  LIMIT 3
) topVendors

```

Query 10 Find the top-10 most active persons by aggregating the posts during the last year, then calculate their RFM (Recency, Frequency, Monetary) value in the same period, and return their recent reviews and tags of interest

```
SELECT
  orders.PersonId,
  topPosters.interests,
  MAX(orders.OrderDate) recency,
  COUNT(orders.PersonId) frequency,
  SUM(orders.TotalPrice) monetary,
  ARRAY(
    SELECT feedback FROM feedback
    WHERE personId = orders.PersonId
    LIMIT 10
  ) recentReviews
FROM orders JOIN (
  LANG cypher
  MATCH (cust)<-[:hasCreator]-(post)
  WHERE post.creationDate >
    date.sub(date("2011-12-31"), interval("1 year"))
  WITH cust, count(post) AS postCount
  ORDER BY postCount DESC LIMIT 10
  MATCH (cust)-[:hasInterest]->(tag)
  RETURN cust.id AS custId, collect(tag.id) AS interests
) topPosters
ON orders.PersonId = topPosters.custId
GROUP BY orders.PersonId, topPosters.interests
```

B TPC-H Queries

The following are the DortDB versions of the TPC-H queries.

Query 1 Pricing Summary Report Query

```
select
  l.returnflag,
  l.linestatus,
  sum(l.quantity) as sum_qty,
  sum(l.extendedprice) as sum_base_price,
  sum(l.extendedprice * (1 - l.discount)) as sum_disc_price,
  sum(l.extendedprice * (1 - l.discount) * (1 + l.tax))
    as sum_charge,
  avg(l.quantity) as avg_qty,
  avg(l.extendedprice) as avg_price,
  avg(l.discount) as avg_disc,
  count(*) as count_order
from
  lineitem l
where
  l.shipdate <= date.sub('1998-12-01'::date, interval('70 days'))
group by
  l.returnflag,
  l.linestatus
order by
  l.returnflag,
  l.linestatus;
```

Query 2 Minimum Cost Supplier Query

```
select
    s.acctbal,
    s.name,
    n.name,
    p.partkey,
    p.mfgr,
    s.address,
    s.phone,
    s.comment
from
    part p,
    supplier s,
    partsupp ps,
    nation n,
    region r
where
    p.partkey = ps.partkey
    and s.supkey = ps.supkey
    and p.size = 31
    and p.type like '%NICKEL'
    and s.nationkey = n.nationkey
    and n.regionkey = r.regionkey
    and r.name = 'AMERICA'
    and ps.supplycost = (
        select
            min(ps.supplycost)
        from
            partsupp ps,
            supplier s,
            nation n,
            region r
        where
            p.partkey = ps.partkey
            and s.supkey = ps.supkey
            and s.nationkey = n.nationkey
            and n.regionkey = r.regionkey
            and r.name = 'AMERICA'
    )
order by
    s.acctbal desc,
    n.name,
    s.name,
    p.partkey
limit 100;
```

Query 3 Shipping Priority Query

```
select
    l.orderkey,
    sum(l.extendedprice * (1 - l.discount)) as revenue,
    o.orderdate,
    o.shippriority
from
    customer c,
    orders o,
    lineitem l
where
    c.mktsegment = 'HOUSEHOLD'
    and c.custkey = o.custkey
    and l.orderkey = o.orderkey
    and o.orderdate < '1995-03-09'::date
    and l.shipdate > '1995-03-09'::date
group by
    l.orderkey,
    o.orderdate,
    o.shippriority
order by
    revenue desc,
    o.orderdate
limit 10;
```

Query 4 Order Priority Checking Query

```
select
    o.orderpriority,
    count(*) as order_count
from
    orders o
where
    o.orderdate >= '1994-05-01'::date
    and o.orderdate < date.add('1994-05-01'::date, interval('3 months'))
    and exists (
        select
            1
        from
            lineitem l
        where
            l.orderkey = o.orderkey
            and l.commitdate < l.receiptdate
    )
group by
    o.orderpriority
order by
    o.orderpriority;
```

Query 5 Local Supplier Volume Query

```
select
    n.name,
    sum(l.extendedprice * (1 - l.discount)) as revenue
from
    customer c,
    orders o,
    lineitem l,
    supplier s,
    nation n,
    region r
where
    c.custkey = o.custkey
    and l.orderkey = o.orderkey
    and l.suppkey = s.suppkey
    and c.nationkey = s.nationkey
    and s.nationkey = n.nationkey
    and n.regionkey = r.regionkey
    and r.name = 'ASIA'
    and o.orderdate >= '1997-01-01'::date
    and o.orderdate <
        date.add('1997-01-01'::date, interval('1 year'))
group by
    n.name
order by
    revenue desc;
```

Query 6 Forecasting Revenue Change Query

```
select
    sum(l.extendedprice * l.discount) as revenue
from
    lineitem l
where
    l.shipdate >= '1997-01-01'::date
    and l.shipdate <
        date.add('1997-01-01'::date, interval('1 year'))
    and l.discount between 0.08 - 0.01 and 0.08 + 0.01
    and l.quantity < 24;
```

Query 7 Volume Shipping Query

```
select
    shipping.supp_nation,
    shipping.cust_nation,
    shipping.year,
    sum(shipping.volume) as revenue
from
    (
        select
            n1.name as supp_nation,
            n2.name as cust_nation,
            date.extract(1.shipdate, 'year') as year,
            1.extendedprice * (1 - 1.discount) as volume
        from
            supplier s,
            lineitem l,
            orders o,
            customer c,
            nation n1,
            nation n2
        where
            s.supkey = l.supkey
            and o.orderkey = l.orderkey
            and c.custkey = o.custkey
            and s.nationkey = n1.nationkey
            and c.nationkey = n2.nationkey
            and (
                (
                    n1.name = 'ROMANIA' and
                    n2.name = 'BRAZIL'
                ) or (
                    n1.name = 'BRAZIL' and
                    n2.name = 'ROMANIA'
                )
            )
            and 1.shipdate between '1995-01-01'::date
                and '1996-12-31'::date
        ) as shipping
    group by
        shipping.supp_nation,
        shipping.cust_nation,
        shipping.year
    order by
        shipping.supp_nation,
        shipping.cust_nation,
        shipping.year;
```

Query 8 National Market Share Query

```
select
    all_nations.year,
    sum(case
        when all_nations.nation = 'BRAZIL' then
            all_nations.volume
        else 0
    end) / sum(all_nations.volume) as mkt_share
from
    (
        select
            date.extract(o.orderdate, 'year') as year,
            l.extendedprice * (1 - l.discount) as volume,
            n2.name as nation
        from
            part p,
            supplier s,
            lineitem l,
            orders o,
            customer c,
            nation n1,
            nation n2,
            region r
        where
            p.partkey = l.partkey
            and s.supkey = l.supkey
            and l.orderkey = o.orderkey
            and o.custkey = c.custkey
            and c.nationkey = n1.nationkey
            and n1.regionkey = r.regionkey
            and r.name = 'AMERICA'
            and s.nationkey = n2.nationkey
            and o.orderdate between '1995-01-01'::date
                and '1996-12-31'::date
            and p.type = 'LARGE POLISHED NICKEL'
        ) as all_nations
group by
    all_nations.year
order by
    all_nations.year;
```

Query 9 Product Type Profit Measure Query

```
select
    profit.nation,
    profit.year,
    sum(profit.amount) as sum_profit
from
    (
        select
            n.name as nation,
            date.extract(o.orderdate, 'year') as year,
            l.extendedprice * (1 - l.discount) -
            ps.supplycost * l.quantity as amount
        from
            part p,
            supplier s,
            lineitem l,
            partsupp ps,
            orders o,
            nation n
        where
            s.suppkey = l.suppkey
            and ps.suppkey = l.suppkey
            and ps.partkey = l.partkey
            and p.partkey = l.partkey
            and o.orderkey = l.orderkey
            and s.nationkey = n.nationkey
            and p.name like '%navy%'
        ) as profit
group by
    profit.nation,
    profit.year
order by
    profit.nation,
    profit.year desc;
```

Query 10 Returned Item Reporting Query

```
select
    c.custkey,
    c.name,
    sum(l.extendedprice * (1 - l.discount)) as revenue,
    c.acctbal,
    n.name,
    c.address,
    c.phone,
    c.comment
from
    customer c,
    orders o,
    lineitem l,
    nation n
where
    c.custkey = o.custkey
    and l.orderkey = o.orderkey
    and o.orderdate >= '1994-11-01'::date
    and o.orderdate <
        date.add('1994-11-01'::date, interval('3 month'))
    and l.returnflag = 'R'
    and c.nationkey = n.nationkey
group by
    c.custkey,
    c.name,
    c.acctbal,
    c.phone,
    n.name,
    c.address,
    c.comment
order by
    revenue desc
limit 20;
```

Query 11 Important Stock Identification Query

```
select
    ps.partkey,
    sum(ps.supplycost * ps.availqty) as value
from
    partsupp ps,
    supplier s,
    nation n
where
    ps.suppkey = s.suppkey
    and s.nationkey = n.nationkey
    and n.name = 'UNITED KINGDOM'
group by
    ps.partkey having
        sum(ps.supplycost * ps.availqty) > (
            select
                sum(ps.supplycost * ps.availqty) *
                0.0010000000
            from
                partsupp ps,
                supplier s,
                nation n
            where
                ps.suppkey = s.suppkey
                and s.nationkey = n.nationkey
                and n.name = 'UNITED KINGDOM'
        )
order by
    value desc;
```

Query 12 Shipping Modes and Order Priority Query

```
select
    l.shipmode,
    sum(case
        when o.orderpriority = '1-URGENT'
            or o.orderpriority = '2-HIGH'
        then 1
        else 0
    end) as high_line_count,
    sum(case
        when o.orderpriority <> '1-URGENT'
            and o.orderpriority <> '2-HIGH'
        then 1
        else 0
    end) as low_line_count
from
    orders o,
    lineitem l
where
    o.orderkey = l.orderkey
    and l.shipmode in ('FOB', 'TRUCK')
    and l.commitdate < l.receiveptdate
    and l.shipdate < l.commitdate
    and l.receiveptdate >= '1997-01-01'::date
    and l.receiveptdate <
        date.add('1997-01-01'::date, interval('1 year'))
group by
    l.shipmode
order by
    l.shipmode;
```

Query 13 Customer Distribution Query

```
select
  orders.count,
  count(*) as custdist
from
  (
    select
      c.custkey,
      count(o.orderkey) as count
    from
      customer c left outer join orders o on
        c.custkey = o.custkey
        and o.comment not like
          '%unusual%deposits%'
    group by
      c.custkey
  ) as orders
group by
  orders.count
order by
  custdist desc,
  orders.count desc;
```

Query 14 Promotion Effect Query

```
select
  100.00 * sum(case
    when p.type like 'PROMO%'
      then l.extendedprice * (1 - l.discount)
    else 0
  end) / sum(l.extendedprice * (1 - l.discount)) as promo_revenue
from
  lineitem l,
  part p
where
  l.partkey = p.partkey
  and l.shipdate >= '1997-07-01'::date
  and l.shipdate <
    date.add('1997-07-01'::date, interval('1 month'));
```

Query 15 Top Supplier Query

```
with revenue0 (supplier_no, total_revenue) as materialized (  
    select  
        l.supkey,  
        sum(l.extendedprice * (1 - l.discount))  
    from  
        lineitem l  
    where  
        l.shipdate >= '1993-07-01'::date  
        and l.shipdate <  
            date.add('1993-07-01'::date, interval('3 month'))  
    group by  
        l.supkey  
)  
select  
    s.supkey,  
    s.name,  
    s.address,  
    s.phone,  
    r.total_revenue  
from  
    supplier s,  
    revenue0 r  
where  
    s.supkey = r.supplier_no  
    and r.total_revenue = (  
        select  
            max(total_revenue)  
        from  
            revenue0  
    )  
order by  
    s.supkey;
```

Query 16 Parts/Supplier Relationship Query

```
select
    p.brand,
    p.type,
    p.size,
    count(distinct ps.supkey) as supplier_cnt
from
    partsupp ps,
    part p
where
    p.partkey = ps.partkey
    and p.brand <> 'Brand#42'
    and p.type not like 'MEDIUM ANODIZED%'
    and p.size in (15, 30, 20, 2, 3, 36, 37, 38)
    and ps.supkey not in (
        select
            s.supkey
        from
            supplier s
        where
            s.comment like '%Customer%Complaints%'
    )
group by
    p.brand,
    p.type,
    p.size
order by
    supplier_cnt desc,
    p.brand,
    p.type,
    p.size;
```

Query 17 Small-Quantity-Order Revenue Query

```
select
    sum(l.extendedprice) / 7.0 as avg_yearly
from
    lineitem l,
    part p
where
    p.partkey = l.partkey
    and p.brand = 'Brand#22'
    and p.container = 'LG JAR'
    and l.quantity < (
        select
            0.2 * avg(l.quantity)
        from
            lineitem l
        where
            l.partkey = p.partkey
    );
```

Query 18 Large Volume Customer Query

```
select
    c.name,
    c.custkey,
    o.orderkey,
    o.orderdate,
    o.totalprice,
    sum(l.quantity)
from
    customer c,
    orders o,
    lineitem l
where
    o.orderkey in (
        select
            l.orderkey
        from
            lineitem l
        group by
            l.orderkey having
                sum(l.quantity) > 313
    )
    and c.custkey = o.custkey
    and o.orderkey = l.orderkey
group by
    c.name,
    c.custkey,
    o.orderkey,
    o.orderdate,
    o.totalprice
order by
    o.totalprice desc,
    o.orderdate
limit 100;
```

Query 19 Discounted Revenue Query

```
select
    sum(l.extendedprice* (1 - l.discount)) as revenue
from
    lineitem l,
    part p
where
    (
        p.partkey = l.partkey
        and p.brand = 'Brand#23'
        and p.container in (
            'SM CASE', 'SM BOX', 'SM PACK', 'SM PKG'
        )
        and l.quantity >= 8 and l.quantity <= 8 + 10
        and p.size between 1 and 5
        and l.shipmode in ('AIR', 'AIR REG')
        and l.shipinstruct = 'DELIVER IN PERSON'
    )
    or
    (
        p.partkey = l.partkey
        and p.brand = 'Brand#23'
        and p.container in (
            'MED BAG', 'MED BOX', 'MED PKG', 'MED PACK'
        )
        and l.quantity >= 17 and l.quantity <= 17 + 10
        and p.size between 1 and 10
        and l.shipmode in ('AIR', 'AIR REG')
        and l.shipinstruct = 'DELIVER IN PERSON'
    )
    or
    (
        p.partkey = l.partkey
        and p.brand = 'Brand#24'
        and p.container in (
            'LG CASE', 'LG BOX', 'LG PACK', 'LG PKG'
        )
        and l.quantity >= 24 and l.quantity <= 24 + 10
        and p.size between 1 and 15
        and l.shipmode in ('AIR', 'AIR REG')
        and l.shipinstruct = 'DELIVER IN PERSON'
    );
```

Query 20 Potential Part Promotion Query

```
select
    s.name,
    s.address
from
    supplier s,
    nation n
where
    s.suppkey in (
        select
            ps.suppkey
        from
            partsupp ps
        where
            ps.partkey in (
                select
                    p.partkey
                from
                    part p
                where
                    p.name like 'mint%'
            )
        and ps.availqty > (
            select
                0.5 * sum(l.quantity)
            from
                lineitem l
            where
                l.partkey = ps.partkey
                and l.suppkey = ps.suppkey
                and l.shipdate >=
                    '1994-01-01'::date
                and l.shipdate <
                    date.add('1994-01-01'::date, interval('1 year'))
        )
    )
    and s.nationkey = n.nationkey
    and n.name = 'KENYA'
order by
    s.name;
```

Query 21 Suppliers Who Kept Orders Waiting Query

```
select
    s.name,
    count(*) as numwait
from
    supplier s,
    lineitem l1,
    orders o,
    nation n
where
    s.supkey = l1.supkey
    and o.orderkey = l1.orderkey
    and o.orderstatus = 'F'
    and l1.receiptdate > l1.commitdate
    and exists (
        select
            1
        from
            lineitem l2
        where
            l2.orderkey = l1.orderkey
            and l2.supkey <> l1.supkey
    )
    and not exists (
        select
            1
        from
            lineitem l3
        where
            l3.orderkey = l1.orderkey
            and l3.supkey <> l1.supkey
            and l3.receiptdate > l3.commitdate
    )
    and s.nationkey = n.nationkey
    and n.name = 'INDONESIA'
group by
    s.name
order by
    numwait desc,
    s.name
limit 100;
```

Query 22 Global Sales Opportunity Query

```
select
    custsale.cnttrycode,
    count(*) as numcust,
    sum(custsale.acctbal) as totacctbal
from
    (
        select
            substr(c.phone, 0, 2) as cnttrycode,
            c.acctbal
        from
            customer c
        where
            substr(c.phone, 0, 2) in
                ('24', '18', '27', '28', '29', '10', '17')
            and c.acctbal > (
                select
                    avg(c.acctbal)
                from
                    customer c
                where
                    c.acctbal > 0.00
                    and substr(c.phone, 0, 2) in
                        ('24', '18', '27', '28', '29', '10', '17')
            )
        and not exists (
            select
                1
            from
                orders o
            where
                o.custkey = c.custkey
        )
    ) as custsale
group by
    custsale.cnttrycode
order by
    custsale.cnttrycode;
```

C AST in More Detail

The AST classes for each language can be categorized by the language concepts they embody. In the following table, we can see how the languages themselves compare in terms of features. The class names are colored according to the package they belong to.

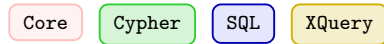


Table C.1 AST node types

AST Node Types			
Identifiers			
The identifiers are composed of individual tokens, such as schema names, table names, and so on. They are the only AST nodes used in both logical and physical plans. The individual languages specialize the base <code>ASTIdentifier</code> to include parsing logic and store the original representation. XQuery provides two subclasses because the language grammar distinguishes between variables and general identifiers.			
ASTIdentifier CypherIdentifier SQLIdentifier XQueryIdentifier ASTVariable			
AST Node Types (continued)			
Simple Expressions			
These are used for literals, function calls, and similar. They are usually simply wrappers for the operation name and its arguments. All of the nodes are later converted into calculation intermediaries.			
ASTLiteral ASTFunction ASTOperator			
Concept	Cypher	SQL	XQuery
String literal	ASTStringLiteral	ASTStringLiteral	ASTStringLiteral
Number literal	ASTNumberLiteral	ASTNumberLiteral	ASTNumberLiteral
Boolean literal	ASTBooleanLiteral	Uses the base <i>ASTLiteral</i>	No boolean literals, instead there are the <code>true()</code> or <code>false()</code> functions
List literal	ASTListLiteral	ASTArray	SequenceConstructor
Tuple literal	Only lists	ASTTuple	Only sequences
Dictionary literal	ASTMapLiteral	ASTRow	XML elements
Function calls	FnCallWrapper	Uses the base <i>ASTFunction</i>	DynamicFunctionCall also the base <i>ASTFunction</i>

AST Node Types (continued)			
Simple Expressions (continued)			
Function calls producing streams	<i>Specified by properties of the <code>FnCallWrapper</code></i>	<code>TableFn</code>	<i>No special syntax</i>
Aggregates	<i>No special syntax, same as standard functions</i>	<code>ASTAggregate</code> <code>ASTWindowFn</code> <i>Some aggregates are parsed as standard functions</i>	<i>No special syntax, same as standard functions</i>
Partial function application	<i>No such concept</i>	<i>No such concept</i>	<code>BoundFunction</code>
List indexing	<code>SubscriptExpr</code>	<code>ASTSubscript</code>	<i>Filter expressions and path predicates</i>
Dictionary indexing	<code>PropLookup</code>	<code>ASTSubscript</code>	<i>Paths</i>
Type casting	<i>Standard function syntax</i>	<code>ASTCast</code>	<code>CastExpr</code>
Type checking	<i>Not directly present, <code>labels(node)</code> and <code>type(edge)</code> functions</i> <code>LabelFilterExpr</code>	<i>Not directly present</i>	<code>InstanceOfExpr</code> <code>ASTSequenceType</code> <i>DortDB recognizes the expression, but does not support its execution</i>
Conditionals	<code>CaseExpr</code>	<code>ASTCase</code>	<code>SwitchExpr</code> <code>IfExpr</code>
Model-specific			<code>OrderedExpr</code>
Complex Expressions			
These are still only expressions and not statements, but they cannot be classified as simple. They are either grammatically very complex (like XQuery direct constructors) or result in one or more plan operators that are not calculation intermediaries (for example, the Cypher list comprehension).			
Concept	Cypher	SQL	XQuery
Function declaration	<i>No such concept</i>	<i>No such concept</i>	<code>InlineFunction</code>
Comprehension	<code>Pattern Comprehension</code> <code>List Comprehension</code>	<i>No such concept</i>	<code>SimpleMapExpr</code> <code>FilterExpr</code> <i>Also paths</i>

AST Node Types (continued)			
Complex Expressions (continued)			
Model-specific (graph or tree traversal or creation)	PatternElChain NodePattern RelPattern		DirectElement Constructor DirectComment Constructor DirectPIConstructor ComputedConstructor PathExpr PathPredicate
Subquery Expressions			
Expressions that always include a subquery.			
LangSwitch			
Concept	Cypher	SQL	XQuery
Quantified query	QuantifiedExpr	ASTQuantifier	QuantifiedExpr
Exists subquery	ExistsSubquery	ASTExists	<i>Only exists(sequence) function</i>
Statements			
Statements and other nodes that make up the skeleton of a query. Usually, there is a separate class for each statement clause, but SQL enforces a strict statement structure, so most of the clauses are included in the main SelectStatement and SelectSet classes.			
Concept	Cypher	SQL	XQuery
Full statement	Query	SelectStatement	Module FLWORExpr
Projection	ReturnClause WithClause	ValuesClause SelectSet	FLWORLet FLWORReturn
Selection	MatchClause WithClause	SelectSet	FLWORWhere
Order by	ReturnClause WithClause	SelectStatement	FLWOROrderBy
Group by	<i>Groups by projected attributes that are not aggregates</i>	GroupByClause	FLWORGroupBy
Join	MatchClause	JoinClause RowsFrom	FLWORFor

AST Node Types (continued)			
Statements (continued)			
Depend join	MatchClause UnwindClause	JoinClause <i>if specified to be a lateral join, also subqueries</i>	FLWORFor
Set operation	SetOp	SelectSetOp	union, intersect and except operators
Distinct values	ReturnClause WithClause	SelectSet	distinct-values(seq) function
Limit, offset	ReturnClause WithClause	SelectStatement	subsequence(sequence, from, to) function or path predicates or filter expressions
Ordinal position	<i>Not directly, can be done with combination of UNWIND and WITH clauses</i>	<i>Either window functions or the WITH ORDINALITY specifier for table functions</i>	FLWORCount also implicitly in path context variables
Window definitions for window functions	<i>Cypher does not have window functions. Workarounds are possible with more or less complex combinations of UNWIND and WITH clauses</i>	SelectStatement	FLWORWindow
Recursion	<i>Through pattern matching</i>	WithQuery	<i>Through path operators, also by declaring custom recursive functions</i>
Data modification	CreateClause MergeClause SetClause RemoveClause DeleteClause	<i>Not implemented in DortDB</i>	<i>Not part of the language</i>

AST Node Types (continued)			
Statements (continued)			
Pragmas	<i>Not implemented in DortDB</i>	<i>Not implemented in DortDB</i>	Prolog NSDeclaration DefaultNS Declaration BaseURIDeclaration OrderingDeclaration EmptyOrder Declaration
Statement Helpers			
Strictly structural nodes with no meaning on their own. For example, the SQL ORDER BY clause contains a list of subexpressions that specify the sort key, the sort direction, and the behavior for sorting empty fields. One such subexpression would be described with the <code>OrderByItem</code> class. MergeAction SetItem RemoveItem ProjectionBody OrderItem OrderByItem WindowSpec ASTTableAlias ASTExpressionAlias FLWORForBinding WindowBoundary OrderByItem DirConstrContent PathAxis			

Query 23 Example of a basic XQuery query AST

```
for $x in $books/bookstore/book
where $x/price > 30
return $x/title
```

```
// results in the following
Module({
  prolog: Prolog([]),
  body: [
    FLWORExpr([
      FLWORFor([
        {variable: ASTVariable('x'), expr: PathExpr([/**/])}
      ]),
      FLWORWhere(ASTOperator(
        XQueryIdentifier('>'),
        [/*path*/, ASTNumberLiteral(30)]
      )),
      FLWORReturn(PathExpr([
        ASTVariable('x'),
        PathAxis(
          'child',
          ASTItemType('element', XQueryIdentifier('title'))
        )
      ])),
    ])
  ]
})
```

D Source Code

A part of this thesis is the source code for the implemented software. The source code is also available at <https://github.com/filipjezek/dortdb>.