applied sciences

MDPI

Article

PolyFuseQL: A Unified Middleware for Polyglot Persistence Using the Strategy Design Pattern and Apache Spark Federation

Andres Andrade-Cabrera *, Diana Martinez-Mosquera and Ivan Carrera

Departamento de Ciencias de la Computación, Escuela Politécnica Nacional, Quito 170525, Ecuador; diana.martinez@epn.edu.ec (D.M.-M.); ivan.carrera@epn.edu.ec (I.C.)

* Correspondence: luis.andrade03@epn.edu.ec

Abstract

Modern applications rely on diverse database types to store information efficiently, but interacting with these systems requires developers to master multiple specialized query languages. This fragments development, as engineers must write complex custom code for each database just to retrieve routine data. While existing big-data tools offer a unified query language, they force all data requests through heavy processing engines, degrading the fast performance required for everyday tasks. To address this problem, we introduce PolyFuseQL, a smart routing layer acting as a universal SQL translator for five popular databases (PostgreSQL, Redis, Neo4j, MongoDB, and Cassandra). Our objective is to allow developers to communicate transparently with diverse systems using standard SQL, completely avoiding the need to write specific NoSQL logic. Methodologically, PolyFuseQL dynamically evaluates standard SQL queries to determine the optimal execution path. It translates and sends simple requests directly to the native databases to ensure high speed while seamlessly routing complex analytical queries to a powerful Apache Spark engine. We evaluated the system's performance across varying workloads. Results indicate that PolyFuseQL adds almost no delay to simple queries, preserving sub-millisecond response times. Concurrently, it enables users to execute complex operations, such as eight-table joins, on NoSQL databases that lack native relational support. Although a fixed processing delay during initial data translation slightly limits absolute hardware scaling, the system scales reliably for medium-sized analytical tasks. Ultimately, PolyFuseQL provides a unified, easy-to-use SQL interface to diverse NoSQL databases without sacrificing operational speed.

Keywords: polyglot persistence; unified query interface; SQL-to-NoSQL; middleware; Apache Spark; design patterns; TPC-H; federated database; design science research

Check for updates

Academic Editors: Shi-Jinn Horng, Ania Cravero and Mónica Caniupan

Received: 6 June 2026
Revised: 21 July 2026
Accepted: 22 July 2026
Published: 27 August 2026

Copyright: © 2026 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license.

1. Introduction

Contemporary software applications generate data that grows continuously in both volume and structural diversity. Traditional Relational Database Management Systems (RDBMSs), which enforce rigid schemas and rely on vertical scaling, are increasingly unable to satisfy the elasticity and performance demands imposed by these workloads [1,2]. As a consequence, organizations have progressively adopted specialized storage engines that target specific access patterns, ranging from in-memory caching to graph traversal [3]. This pragmatic, multi-engine strategy is widely known as polyglot persistence and is, in essence, the recognition that no single database technology is optimal for every problem.

Within this paradigm, NoSQL databases have evolved from simple key–value stores into specialized Document, Columnar, and Graph systems, each optimizing for distinct read/write profiles [4,5]. The technical advantages of this heterogeneity are well documented, namely schema flexibility, horizontal scalability, and high throughput on workloads, where strict Atomicity, Consistency, Isolation, Durability (ACID) semantics are not the primary requirement. The trade-off, however, is reduced generalizability. Each engine exposes its own data model, query dialect, and consistency guarantees, which forces developers to maintain niche, case-based application logic and bespoke drivers [6]. Little common ground has been established at the middleware layer, and integrating standard SQL semantics with non-relational APIs increases architectural complexity, isolates data, and prevents unified analytics across diverse datasets [7].

Bridging this gap requires more than a thin wrapper. Enterprise data access solutions must combine high performance with a unified abstraction of storage details, while preserving the specialized advantages of each engine. Recent work explores this idea through canonical data models, mediator/wrapper patterns, and SQL-like dialects extended to cover NoSQL features [8–10]. Nevertheless, two recurring shortcomings limit the practical impact of these proposals. First, transactional consistency across heterogeneous engines is rarely addressed, with most surveyed frameworks either ignoring multi-store transactions or relying on local consistency only [11]. Second, although general-purpose NoSQL benchmarks such as the Yahoo! Cloud Serving Benchmark (YCSB) [12] are well established, they target native Create, Read, Update, Delete throughput on individual stores; the field still lacks a standardized benchmark for *SQL-to-NoSQL query-translation correctness*, in which a relational analytical workload is executed over non-relational engines and validated against a relational ground truth, which leaves cross-engine correctness and overhead claims largely anecdotal [10]. Different design patterns, such as the Data Access Object(DAO) Compound Mediator and the Integrated Polyglot DAO [13], offer alternative architectural blueprints, but each comes with its own trade-off between modularity and execution overhead.

To address this, we introduce PolyFuseQL, a Unified Middleware designed to execute standard queries under the American National Standards Institute (ANSI) for Structured Query Language (SQL) over a federated *pentaglot* environment consisting of PostgreSQL, Redis, Neo4j, MongoDB, and Cassandra. The architecture is grounded in three established design patterns, Strategy, Adapter, and Abstract Factory, and integrates Apache Spark (version 4.0.0) as a hybrid execution engine that resolves complex aggregations and multi-table joins over data drawn from an individual backend through MapReduce-style plans [14]. The development follows the Design Science Research (DSR) methodology [15], with internal Scrum-based iterations to manage technical risk associated with consistency, isolation, and latency across heterogeneous backends.

This work makes three contributions. The first is architectural: the design of an Abstract Syntax Tree (AST)-driven, dual-path routing engine that inspects each parsed SQL statement at runtime and dispatches it either to a native driver for low-latency transactional access or to an Apache Spark execution path for analytical workloads, thereby reconciling sub-millisecond OLTP with relational analytics behind a single SQL-compliant interface spanning five storage paradigms. The second is empirical and methodological: a reproducible polyglot evaluation baseline that validates semantic correctness against a relational ground truth across scale factors from 0.01 to 10.0 GB and two hardware tiers, supplying the measured, replicable comparison that the surveyed literature typically lacks. The third is analytical: the identification and root-cause attribution of the Python–Java Virtual Machine serialization bridge (Py4J version 0.10.9.7), rather than disk or network Input/Output (I/O), as the dominant ceiling on vertical scaling for this class of middleware.

As summarized in Table 1, prior frameworks tend to focus on a subset of NoSQL paradigms, omit Spark-based federation, or do not validate semantic correctness against a relational ground truth; PolyFuseQL addresses these gaps simultaneously.

There are other works that are similar to PolyFuseQL and are referenced in Table 1 that compare the proposed architecture with the existing frameworks for polyglot query interfaces. The columns compares the following dimensions first the paradigms that are covered, after that the nature of the federation engine, and finally the rigor of the empirical benchmarking provided. The table shows that while the previous solutions offer conceptual translations of the queries, they lack to provide reproducible empirical benchmarks against a standardized relational baseline.

Table 1. Comparison between PolyFuseQL and representative prior work on unified query interfaces over polyglot persistence layers.

Approach	Paradigms Covered	Federation Engine	Empirical Benchmark
SQLtoKeyNoSQL [9]	Key–value-derived backends	Canonical key tree	Functional, partial
Query Translation Survey [10]	Cross-model rewriting (40+ systems)	Variable (Datalog, R2RML)	No unified benchmark
Valentine & Kabaso UQP [8]	Key–value, column, document, graph	Modular adapters	Functional only
Holubová's Vision [16]	Multi-model, schema inference	Mediator/wrapper (conceptual)	Not provided
SQL-based Integration [17]	Relational + document + key–value	Extended SQL parser	Throughput, single-store
PolyFuseQL (this work)	Relational, key–value, document, columnar, graph	Apache Spark (hybrid routing)	TPC-H (Q1, Q6, Q8), 0.01–10.0 SF, two hardware tiers

The remainder of this paper is organized as follows. Section 2 introduces the technological background, SQL/NoSQL paradigms, the Consistency, Availability, Partition tolerance (CAP) theorem, design patterns, and Apache Spark, followed by the DSR-based methodology and a detailed description of the proposed architecture. Section 3 reports the experimental results in two phases: OLTP micro-benchmarks measuring direct-mode latency and OLAP macro-benchmarks evaluating the Spark federation engine on TPC-H workloads. Section 4 discusses architectural trade-offs, the shift from I/O-bound to compute-bound bottlenecks, and the operational viability of the system. Section 5 synthesizes the contributions, acknowledges the limitations of the study, and outlines directions for future work.

2. Materials and Methods

This section establishes the conceptual and technical foundations of PolyFuseQL. It first reviews the relational and non-relational paradigms that coexist within the polyglot persistence model and the design patterns and distributed-computing principles required to bridge them. It then describes the Design Science Research methodology adopted to develop and evaluate the artifact, the iterative engineering strategy decomposed into four sprints, and the experimental protocol used to assess functional correctness and performance.

2.1. Theoretical Background

2.1.1. Relational and Non-Relational Database Paradigms

A Database Management System (DBMS) is a software architecture that controls the organization, storage, retrieval, and integrity of data in a database [18]. Within this broad category, RDBMSs manage data through relationships among tables organized in rows and columns, enforce a rigid schema, and rely on ACID properties (Atomicity, Consistency, Isolation, Durability) to guarantee reliable transactions [19,20]. Structured Query Language (SQL) is the de facto declarative language for these systems, organized hierarchically into Data Definition (DDL), Data Manipulation (DML), Data Query (DQL), Data Control (DCL), and Transaction Control (TCL) sublanguages [21].

Not Only SQL (NoSQL) databases emerged to overcome the limitations of RDBMSs in the context of high-volume, high-velocity, and semi-structured data [22]. Three core characteristics differentiate them from the relational model: schema flexibility, horizontal scalability via sharding, and high performance achieved through simpler data models [5,23]. The CAP theorem constrains the design space of distributed databases by stating that, in the presence of network partitions, a system can guarantee at most two of three properties: Consistency, Availability, and Partition tolerance [22]. Many NoSQL systems consequently adopt the Basically Available, Soft state, Eventual consistency (BASE) model instead of strict ACID semantics [2].

NoSQL is not a single technology but rather a category of purpose-built engines, generally grouped into four main subtypes: Document-based, Key–Value, Columnar (or wide-column), and Graph [24,25]. Table 2 summarizes the representative engines adopted in this study, highlighting their representative technologies, structural strengths, and inherent limitations based on current literature.

Table 2. Comparison of NoSQL engine subtypes adopted in this study.

Type	Representative Engines	Strengths	Limitations
Document-based	MongoDB [26], CouchDB [26]	Flexible JavaScript Object Notation/Binary JSON (JSON/BSON) schema; intuitive for object-oriented domains	Cumbersome multi-document joins
Key–Value	Redis [27], DynamoDB [28]	Sub-millisecond key lookups; simple horizontal distribution	Limited query expressiveness beyond key access
Columnar	Cassandra [29], HBase [30]	High-throughput writes; partition tolerance; analytics on logs	Complex modelling for non-trivial access patterns
Graph	Neo4j [31], Amazon Neptune [32]	Native traversals over connected data; flexible schema-on-read	Suboptimal for high-volume key lookups

Polyglot persistence acknowledges that no single engine is optimal for every workload. As popularized by Fowler, an e-commerce platform may combine Redis for high-velocity sessions, an RDBMS for ACID-compliant financial records, and MongoDB for a flexible product catalogue [3]. The challenge for unified middleware is therefore not to replace these engines, but to abstract their interfaces while preserving their specialized advantages.

Figure 1 illustrates the schematic representation of polyglot persistence, where distinct application modules (e.g., User Sessions, Financial Data) are mapped to the storage engine best suited to their specific access patterns. This demonstrates the architectural modularity proposed by Fowler.



Figure 1. Schematic representation of polyglot persistence.

2.1.2. Software Engineering Foundations

Three classic design patterns underpin the proposed middleware. The *Abstract Factory* is a creational pattern that produces families of related objects without exposing their concrete classes [33]. PolyFuseQL applies it to instantiate the appropriate connector for each backend through a unified interface, decoupling client logic from concrete driver implementations. The *Adapter* (or Wrapper) is a structural pattern that bridges incompatible interfaces by translating calls from one form into another [34]. It is used to integrate pre-existing translation services (for MongoDB and Cassandra) without modifying their source code, thereby honouring the Open–Closed Principle. The *Strategy* is a behavioural pattern that encapsulates a family of interchangeable algorithms, allowing the algorithm to vary independently from the client [35]. PolyFuseQL employs it to dispatch each parsed SQL statement to a specific execution path (DDL, DML, DQL, or aggregation) at runtime.

2.1.3. Spark Federation for Polyglot Unification

In the context of PolyFuseQL, *Spark Federation* refers to the orchestration of Apache Spark as a centralized, distributed computation layer that endows an individual non-relational store with the relational aggregation and multi-table join semantics it lacks natively. For a given analytical query, the federation engine extracts the required raw data from the target backend, loads it into a unified in-memory environment, and evaluates the complete SQL statement, including multi-way joins among the tables hosted within that single store, using standard relational semantics. Each backend is federated indepen-

dently; unifying tables that physically reside in different engines within one query is a generalization that lies beyond the present scope and is revisited as future work.

Apache Spark is uniquely suited for this demanding polyglot unification task. As detailed by IBM [36], Spark is a highly scalable, open-source data processing engine that utilizes in-memory computing, built-in parallelism, and fault tolerance to execute workloads up to 100 times faster than traditional disk-bound MapReduce architectures. By keeping data in memory across a cluster of nodes, it efficiently handles the massive intermediate datasets generated by the multi-table joins it materializes from a single store. The PySpark interface exposes the comprehensive Spark SQL and DataFrame APIs, combining Python's programmatic flexibility with Spark's robust distributed capabilities [37]. These APIs provide a universal relational abstraction over diverse NoSQL data models, enabling PolyFuseQL to ingest serialized native responses and execute complex analytical operations, such as the eight-table joins used in our macro-benchmarks, that the underlying NoSQL engines cannot perform natively.

2.1.4. Distributed Computing Foundations

The polyglot federation requires distributed computing primitives. *Amdahl's law* provides a deterministic model of the achievable speedup $S(n)$ for a workload with parallel fraction f executed on n processors [38]:

$$S(n) = \frac{1}{(1-f) + \frac{f}{n}}. \quad (1)$$

The law indicates that the sequential portion strictly bounds the speedup. As $n \to \infty$, the maximum achievable speedup saturates at $1/(1-f)$, so adding cores yields diminishing returns when a substantial sequential fraction remains.

The *MapReduce* programming model divides large tasks into smaller subtasks via two primary operations [39]: `Map`, which transforms input data into intermediate key/value pairs, and `Reduce`, which aggregates all intermediate values for a given key. *Apache Spark* extends this paradigm by retaining data in memory between stages and orchestrating execution through a Directed Acyclic Graph (DAG) scheduler, achieving 10–100$\times$ speedups over disk-based Hadoop MapReduce on iterative workloads [36]. Its core abstractions, Resilient Distributed Datasets (RDDs), DataFrames, and Datasets, and its libraries (Spark SQL, Spark Streaming, MLlib, GraphX) make it particularly suitable as a federation engine for analytical SQL workloads. PySpark exposes these capabilities to Python via the Py4J bridge [37], which becomes a critical performance factor in the empirical evaluation reported in Section 3. Finally, *Docker* containerization is used throughout the deployment to encapsulate dependencies and guarantee reproducibility across environments [40].

2.1.5. TPC-H Benchmark

The TPC Benchmark H (TPC-H) is the industry-standard workload for evaluating analytical query processing systems [41]. It models a worldwide wholesale supplier through eight tables (Customer, Orders, Lineitem, Part, Supplier, PartSupp, Nation, Region) and a set of 22 ad hoc queries with multiple operators and selectivity constraints. A Scale Factor (SF) parameterises the dataset size; SF = 1 corresponds to approximately 1 GB of raw data. TPC-H was deliberately selected over the Yahoo! Cloud Serving Benchmark (YCSB). YCSB is a natural fit for NoSQL systems and ships native bindings for several of the engines considered here, including MongoDB and Redis; however, it exercises only simple Create, Read, Update, Delete (CRUD) operations on individual records and provides neither multi-table joins nor a relational ground truth. The central claim of this work is precisely that a non-relational store can answer complex relational workloads, namely multi-way joins and

grouped aggregations, that it cannot express natively, and verifying that claim requires a benchmark with interrelated tables and a canonical relational result against which the deep-equality correctness check can be applied. TPC-H provides exactly this: eight interrelated tables, standardized analytical queries, and an unambiguous PostgreSQL reference. YCSB therefore remains the appropriate instrument for native throughput studies and for the multi-store extensions outlined in Section 5, whereas TPC-H is the correct instrument for the correctness-oriented, relational-capability scope of the present evaluation.

2.2. Design Science Research Methodology

Engineering a Unified Middleware for a pentaglot environment is a non-trivial task that justifies a systematic scientific approach. This work adopts the Design Science Research (DSR) methodology proposed by Peffers et al. [15], which is widely used in Information Systems and Computer Science to guide the construction and evaluation of artifacts. The artifact developed here is PolyFuseQL, a middleware that transpiles generic SQL into the vendor-specific NoSQL languages of multiple backends, including Redis and Neo4j, while delegating Document and Columnar translation to validated external modules.

To balance scientific rigour with engineering flexibility, the DSR life cycle is hybridized with internal Scrum-based sprints. Integrating Apache Spark with Redis and Neo4j alongside external adapters for MongoDB and Cassandra introduces a high technical risk in terms of consistency, isolation, and latency. Embedding Scrum within DSR enables iterative validation, adaptive scoping, and continuous integration of internal and external connectors. As illustrated in Figure 2, the framework is composed of six DSR phases, applied to PolyFuseQL, which are summarized as follows:

1. *Problem Identification and Motivation:* the absence of unified interoperability mechanisms in polyglot persistence environments increases development complexity and promotes vendor lock-in.
2. *Definition of Objectives:* the middleware must parse SQL, route queries to the correct backend, aggregate distributed results, and impose minimal performance overhead.
3. *Design and Development:* the artifact is built iteratively in four sprints, each focused on a specific architectural layer.
4. *Demonstration:* the middleware is deployed in a containerized testbed and exercised with TPC-H workloads.
5. *Evaluation:* OLTP and OLAP benchmarks compare middleware latency and overhead against native PostgreSQL execution.
6. *Communication:* the results are disseminated through this paper as a Academic Dissemination.

As depicted in the development cycle of Figure 2, the artifact was built over multiple iterations, designated as *Sprint I* through *Sprint IV*. Each sprint traversed the complete four-stage cycle, culminating in a functional software increment. Finally, the process terminates at the Communication stage, represented by formal *Academic Dissemination*. This denotes the documentation and peer-reviewed distribution of the architectural artifact and empirical findings.

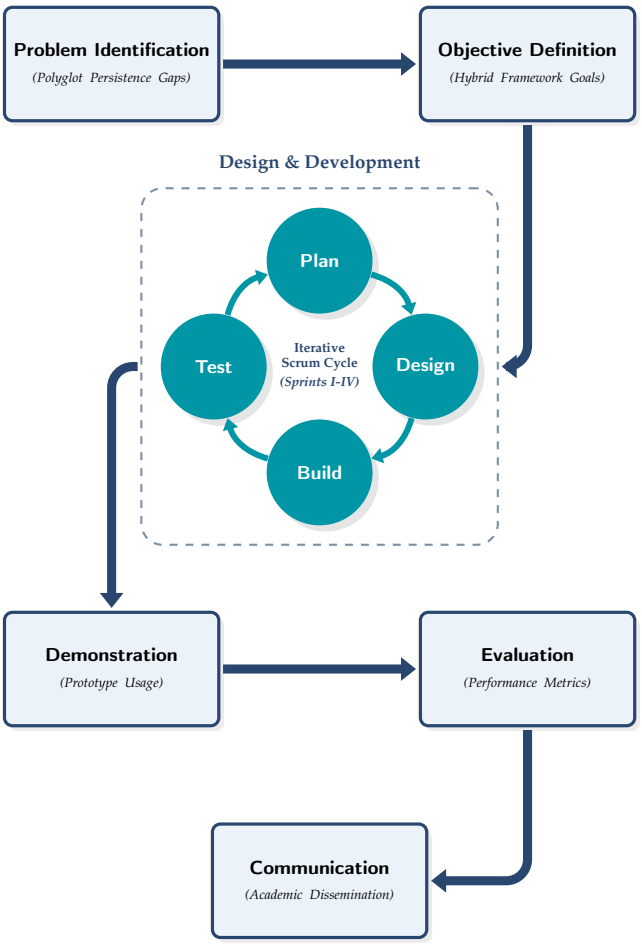


Figure 2. Methodological framework combining DSR and iterative Scrum software development.

2.3. Technological Baseline and Reused Components

A federated middleware architecture rewards strict adherence to component reuse and interface standardization. Strategic integration of validated artifacts, grounded in established software design patterns, mitigates structural redundancy and concentrates development effort on the novel challenges of distributed aggregation and pentaglot federation. The Adapter Pattern enables PolyFuseQL to integrate two pre-existing translation engines as encapsulated black-box services.

Document Store Translation Module. The document-oriented persistence layer integrates the SQL-to-MongoDB Query Language (MQL) translator developed by Chalacama-Mueses [42]. This module decomposes SQL statements into Abstract Syntax Trees (ASTs) and translates conditional WHERE clauses into JSON-based MongoDB filters. The federation engine delegates SQL queries destined for the document layer to this service, which returns data in a standardized JSON format, ensuring that the core middleware remains agnostic to MQL syntax generation.

Columnar Store Translation Module. The columnar layer integrates the middleware developed by Castro Paredes et al. [43], which addresses the structural gap between relational tables and Cassandra wide-column families. It resolves the translation of DDL and DML operations to the Cassandra Query Language (CQL), enabling standard SQL commands to interact with Cassandra's keyspaces and column families.

The reuse of these two modules, complemented by a standard PostgreSQL driver as the relational baseline, allows the present research to focus on (i) the design of native SQL connectors for Redis and Neo4j and (ii) the orchestration logic of the Apache Spark federation engine.

2.4. Proposed Architecture: PolyFuseQL

The implementation of PolyFuseQL was decomposed into four functional increments, each managed as a Scrum sprint. This modular approach decouples service orchestration from query translation logic.

2.4.1. Sprint I: Infrastructure Orchestration

As shown in Figure 3, the first sprint instantiates the container orchestration layer through Docker Compose. A five-node cluster hosts the heterogeneous persistence systems (PostgreSQL, Redis, Neo4j, MongoDB, Cassandra) alongside the middleware and Spark master. All databases are exposed over an internal Docker bridge (`internal_net`), eliminating the manual overhead of configuring five distinct engines on a host machine and guaranteeing reproducibility through a single command. The Middleware API and the Spark Master also retain external network egress, allowing the system to reach databases outside the Docker mesh for extended testing.

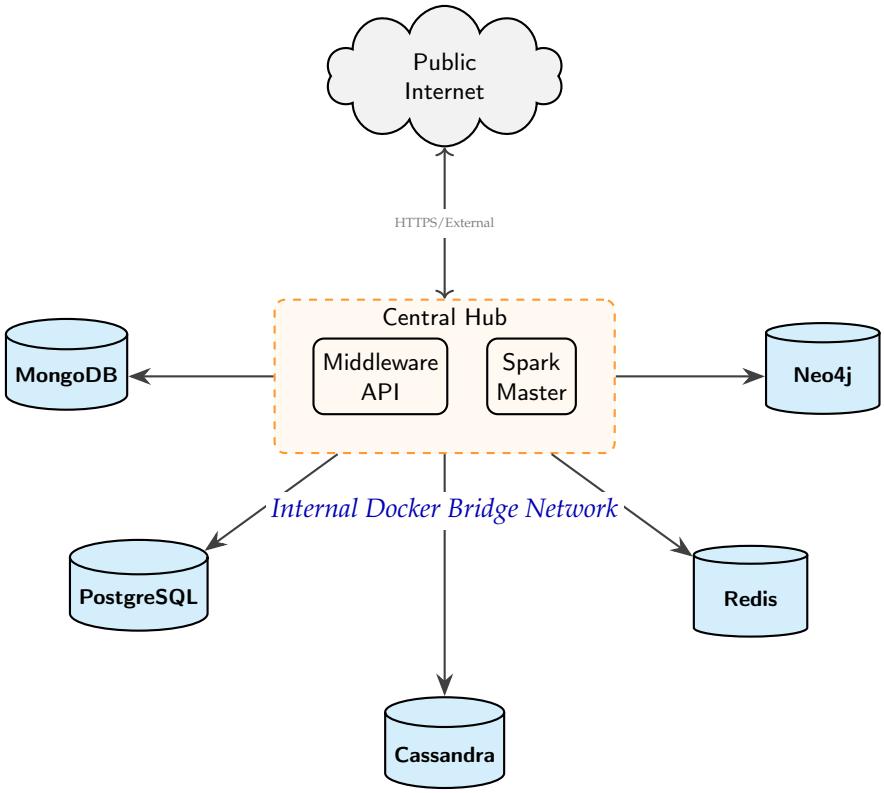


Figure 3. Container network topology of the PolyFuseQL deployment.

2.4.2. Sprint II: Polymorphic Connectivity

The second sprint addresses fragmented interface management through the Strategy design pattern. Figure 4 illustrates a UML with an abstract `Connector`; base class establishes a unified contract for all data interactions, mandating uniform protocols for connection establishment, query execution, and session termination. Five concrete classes inherit from this base, encapsulating engine-specific libraries (`asyncpg`, `aioredis`, the Neo4j driver, and HTTP clients for the MongoDB and Cassandra translator services). The dispatcher holds references only to the `Connector` superclass, allowing backend implementations to be swapped at runtime without altering the orchestration code.

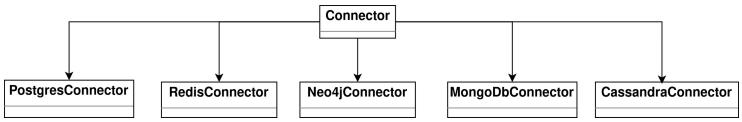


Figure 4. UML class diagram for the Connector hierarchy.

The orchestration logic is summarized in Algorithm 1. The `PolyClient` parses the input SQL string with `sqlglot` to obtain an AST. Inspection of the root expression type identifies the SQL category (insert, select, update), and a `ConnectorFactory` (Abstract Factory) instantiates the appropriate backend. The matching strategy is then resolved from a registry and invoked.

Algorithm 1 Polyglot Query Orchestration

Require: SQL string SQL, PolyClient state $Client$
Ensure: Result set returned from the backend
Procedure ORCHESTRATE(SQL, $Client$)
  $AST \leftarrow$ SQLGLOT.PARSE(SQL)
  $TableName \leftarrow$ EXTRACTTABLE(AST)
  $Schema \leftarrow Client.Catalogue$.GETSCHEMA($TableName$)
  $Backend \leftarrow Schema.backend$
  $Connector \leftarrow$ FACTORY.CREATE($Backend$)
  $Strategy \leftarrow Client$.GETSTRATEGY($AST.Type$)
  if $Strategy$ is NULL **then**
    return $Connector$.RAWQUERY(SQL)
  else
    return $Strategy$.EXECUTE($Client$, AST, $Backend$)
  end if
End Procedure

The middleware decides between two adapter categories. *Internal driver adapters* for Redis and Neo4j wrap standard Python libraries and translate the unified query into engine-specific syntax (Redis `HGETALL` or Cypher `MATCH`). *Proxy adapters* for MongoDB and Cassandra do not interact directly with database drivers; instead, they wrap an HTTP client (`aiohttp`) and forward the SQL string to the external translation microservices described in Section 2.3. Algorithms 2 and 3 formalize these two execution paths.

Algorithm 2 Relational-to-Key/Value Adaptation (Direct Pipe)

Require: SQL AST AST
Ensure: List of typed rows $[TypedRow]$
Procedure DIRECTEXECUTEREDIS(AST)
  $KeyColumn \leftarrow$ EXTRACTKEYCOLUMN(AST)
  $KeyValue \leftarrow$ EXTRACTLITERALVALUE(AST)
  if $KeyValue$ is NULL **then**
    return ERROR("Full Table Scan not supported in Direct Mode")
  end if
  $RedisKey \leftarrow$ CONCAT($TableName$, ":", $KeyValue$)
  $RawData \leftarrow$ REDISCLIENT.HGETALL($RedisKey$)
  if $RawData$ is EMPTY **then**
    return $[\,]$
  end if
  $TypedRow \leftarrow$ TYPECAST($RawData$, $Schema$)
  return $[TypedRow]$
End Procedure

Algorithm 3 External Service Delegation (Proxy Path)

Require: SQL string *SQL*, Translation service URL *TranslatorURL*
Ensure: Normalized JSON rows *NormalizedRows*
Procedure PROXYEXECUTE(*SQL*, *TranslatorURL*)
  Payload ← {`sql:`*SQL*, `format:json`}
  Headers ← {`Authorization:`GETTOKEN()}
  Response ← HTTPCLIENT.POST(*TranslatorURL*, *Payload*, *Headers*)
  if *Response.Status* ≠ 200 **then**
    return ERROR("External Translation Failed")
  end if
  RawJSON ← PARSEJSON(*Response.Body*)
  NormalizedRows ← NORMALIZETYPES(*RawJSON*)
  return *NormalizedRows*
End Procedure

2.4.3. Sprint III: Distributed Processing Integration

The third sprint integrates Apache Spark to resolve the functional limitations of the internal engines. While the proxy strategies for MongoDB and Cassandra leverage external APIs for execution, Redis and Neo4j lack native support for the full SQL-92 aggregation standard. To this end, the middleware implements a *dual-path execution model*. The *direct pipe* handles simple queries by translating the SQL into the native dialect of the target engine and forwarding it to the corresponding driver, optimizing for low latency. The *compiler path* compiles complex SQL into a distributed Spark job, transforming aggregations such as `SUM` or `GROUP BY` into MapReduce-style transformations on a Spark DataFrame.

The decision logic is captured in Algorithm 4. The system inspects the AST for heavy keywords (`JOIN`, `GROUP BY`, aggregates). If they are present and the connector supports Spark translation, the query is compiled to a Spark plan; otherwise, it is transpiled to the native dialect and executed directly.

Algorithm 4 Dual-Path Query Routing

Require: SQL AST *AST*, Database connector *Connector*
Ensure: Query result set
Procedure EXECUTESELECT(*AST*, *Connector*)
if CONTAINS(*AST*, [`JOIN`, `GROUP_BY`, `AGGREGATE`]) **then**
  if *Connector* supports SPARKTRANSLATION **then**
    SparkJob ← COMPILETOSPARK(*AST*)
    return *Connector*.EXECUTESPARK(*SparkJob*)
  else
    return ERROR("Operation not supported on this backend")
  end if
else
  NativeQuery ← TRANSPILE(*AST*, *Connector.Dialect*)
  return *Connector*.DIRECTEXECUTE(*NativeQuery*)
end if
End Procedure

The compilation procedure (Algorithm 5) maps the relational schema to a Spark `StructType`, lazily applies filters and grouping operations to a DataFrame, and finally collects the results. Bifurcating the execution logic prevents the Spark startup cost (the so-called "Spark tax") from impeding simple, high-frequency transactions while still allowing complex analytics on stores that lack native SQL support.

Algorithm 5 SQL-to-Spark Compilation (Compiler Path)

Require: SQL AST *AST*, Spark context *Context*
Ensure: Spark DataFrame collected results
Procedure CompileToSpark(*AST*, *Context*)
TableName ← ExtractTable(*AST*)
Schema ← *Context*.*Catalogue*.GetSchema(*TableName*)
StructType ← MapToSparkTypes(*Schema*)
DataFrame ← *Context*.*Spark*.Read().Format(*Context*.*Source*).Schema(*StructType*).Load()
if *AST* has WhereClause **then**
    DataFrame ← *DataFrame*.Filter(TranslateToSparkExpr(*AST*.*Where*))
end if
if *AST* has GroupBy **then**
    Groups ← ExtractGroups(*AST*)
    Aggregates ← ExtractAggregations(*AST*)
    DataFrame ← *DataFrame*.GroupBy(*Groups*).Agg(*Aggregates*)
end if
return *DataFrame*.Collect()
End Procedure

For the polyglot validation, the relational TPC-H schema in PostgreSQL serves as the semantic ground truth, while the same data is replicated into Redis (using three serialization formats: plain JSON strings, Redis Hashes, and the RedisJSON module) and Neo4j (with each tuple modelled as a node and foreign keys as edges). MongoDB and Cassandra are populated only with the `Orders` collection and the `Logs` table respectively, since the proxy adapters are exercised only in Phase 1 OLTP benchmarks. Each Spark-enabled backend hosts a complete, independent replica of all eight TPC-H tables, so that every analytical query, including the eight-way Q8 join, is resolved entirely within one store rather than across multiple engines simultaneously.

2.4.4. Sprint IV: Quality Assurance and Continuous Integration/Continuous Deployment (CI/CD) Integration

The final sprint provides a DevOps pipeline that runs the full regression suite on every commit. A synchronization script polls the health endpoints of resource-intensive containers (Cassandra and Neo4j) and blocks integration tests until the cluster is fully responsive, ensuring reproducible validation results. The integration of MongoDB and Cassandra additionally demonstrates that the middleware successfully routes queries to legacy APIs and normalizes the resulting JSON and wide-column responses. The complete validation therefore spans three dimensions: *accuracy* against the PostgreSQL baseline, *depth* via Spark on Redis and Neo4j, and *breadth* via proxies for MongoDB and Cassandra.

2.4.5. Graphical User Interface and Artifact Demonstration

To demonstrate the operational viability of the proposed architecture, the backend middleware is exposed through a web-based client in which an SQL editor accepts only standard ANSI SQL, irrespective of the target engine. Only the two views that most directly substantiate the central contribution are retained here; routine interface elements (authentication, runtime connection configuration, and result-format toggling) are described in text and omitted as figures for brevity.

The substantive demonstration is the invariance of the front end across heterogeneous engines. Figure 5 shows a standard `SELECT` executed against the relational PostgreSQL baseline, which returns the expected tabular result and fixes the reference behaviour. Figure 6 shows the *same* ANSI SQL statement routed to the MongoDB document store: the developer writes no MongoDB Query Language (MQL), yet the middleware transparently translates the request and normalizes the document response into the identical tabular shape. The point of interest is the contrast between the two figures, an unchanged SQL front end producing equivalent, normalized output over two structurally different engines, rather than the specific records displayed.

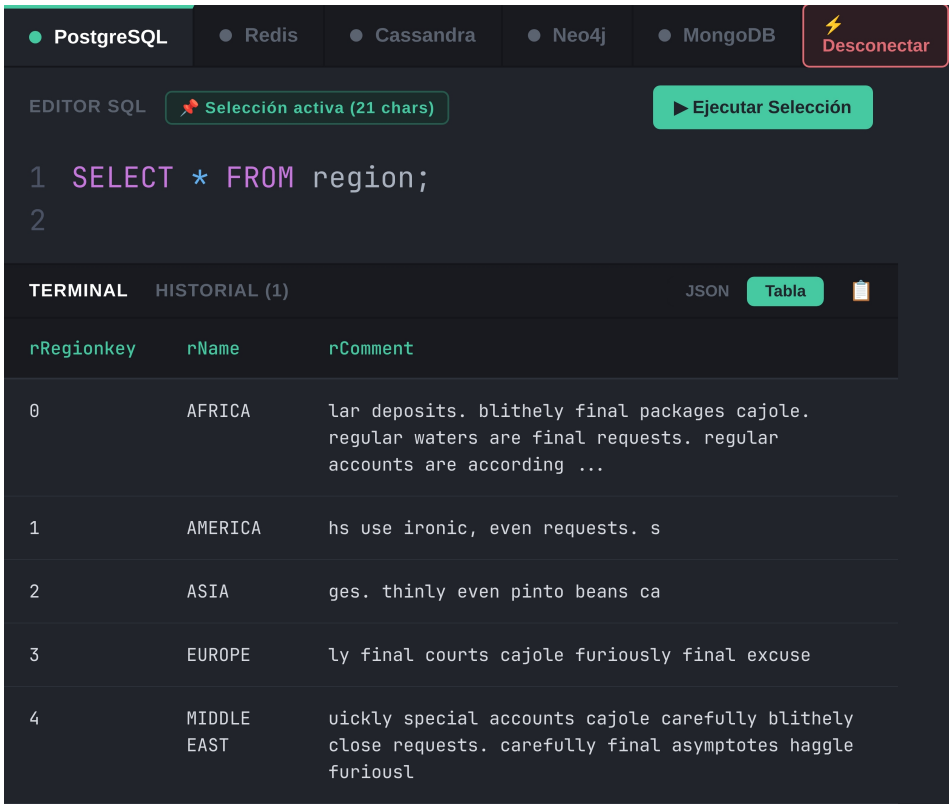


Figure 5. Execution of a standard SQL query against the PostgreSQL baseline, yielding a tabular result.

Beyond these two views, the client also normalizes wide-column responses (for example, from Cassandra) and lets the user toggle the presentation between a structured relational table and a raw normalized JSON payload; as this behaviour adds no information beyond the abstraction already established in Figures 5 and 6, it is reported in text rather than as an additional screenshot.

2.5. Experimental Evaluation Protocol

2.5.1. Hardware Environments

To assess adaptability across infrastructure tiers, the middleware was deployed on two configurations: Environment A (a developer workstation) for a functional baseline and Environment B (a high-performance computing server) for testing vertical scalability and saturation limits. Table 3 details the specifications.

Table 3. Hardware specifications for the two evaluation environments.

Component	Environment A (Workstation)	Environment B (High-Performance Computing (HPC) Server)
Role	Development/Edge Node	Production Analytics Node
CPU	6 Cores/12 Threads	32 Cores/64 Threads
RAM	32 GB DDR4	377 GB ECC DDR4
Storage	Standard SSD	NVMe RAID 0
OS	Arch Linux	Linux (Enterprise/CentOS)

The orchestration node runs Python 3.12 and hosts PolyFuseQL as a FastAPI application. The relational backend is PostgreSQL 16 (Alpine image). The non-relational partitions are Redis Stack and Neo4j 5, while MongoDB and Cassandra are integrated via the third-party translation middlewares described in Section 2.3.

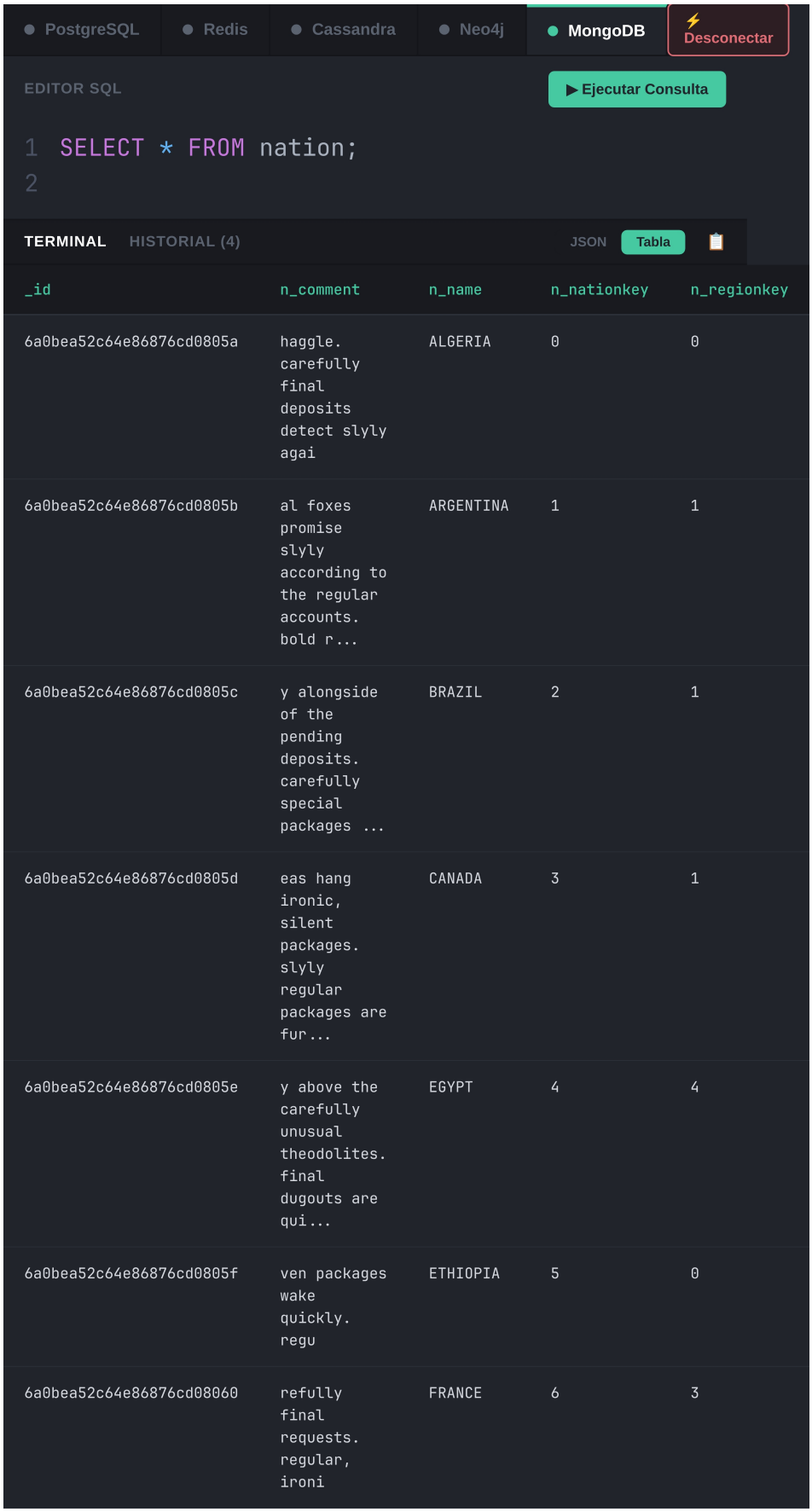


Figure 6. Successful abstraction demonstrating the same standard SQL statement querying the MongoDB document store, with the document response normalized into the identical tabular output produced for the PostgreSQL baseline.

2.5.2. Test Suites and Workloads

The evaluation strategy uses two test suites to isolate distinct performance characteristics. *Phase 1* focuses on OLTP micro-benchmarks executed via direct connectors that bypass Spark, in order to measure raw driver overhead and concurrency handling. The workload (Table 4) consists of low-complexity point lookups, range scans, and single-row inserts executed asynchronously. *Phase 2* assesses OLAP macro-benchmarks based on TPC-H standard. This standard benchmark was selected because it provides a good representation of the industry complexity and also helps to test the distributed processing capabilities of the middleware.

Table 4. OLTP micro-benchmark workload (Phase 1). Queries use direct connectors and bypass the Spark engine.

Query Type	SQL Logic Example	Complexity & Objective
Point Lookup (Order)	`SELECT * FROM orders WHERE o_orderkey = {id}`	Low: minimum driver latency on primary-key access.
Point Lookup (Customer)	`SELECT c_name, c_phone FROM customer WHERE c_custkey = {id}`	Low: attribute-level access and deserialization speed.
Range Scan (Lineitems)	`SELECT * FROM lineitem WHERE l_orderkey = {id}`	Medium: one-to-many retrieval; non-unique index scans.
Simple Filter (Status)	`SELECT id FROM orders WHERE status = 'F' LIMIT 20`	Medium: WHERE/LIMIT pushdown to the source.
Write (Insert)	`INSERT INTO orders VALUES ({id}, ...)`	Medium: ACID-aware write latency.

The TPC-H benchmark contains 22 queries, but for this research and to validate the architectural capabilities of the middleware, only queries Q1, Q6, and Q8 were selected (Table 5). Q1 tests raw throughput via MapReduce-style aggregations without joins; Q6 isolates the efficiency of filter-heavy predicate pushdown; and Q8 stresses the federation engine to its absolute limits with an eight-table join evaluated entirely within a single non-relational store, exercised independently over the graph and the key–value backend, a relational capability that neither engine offers natively. These three queries are not an arbitrary subset: they bracket the operator spectrum at its extremes, from pure aggregation (Q1), through selective filtering (Q6), to maximal multi-way joining (Q8). The remaining nineteen TPC-H queries recombine these same operator classes at intermediate points of that spectrum. Because this research evaluates a functional proof of concept designed to validate the dual-path architecture, exhaustive execution of the entire benchmark suite falls outside the current scope. Instead, isolating these specific operator extremes sufficiently proves the central claim: that the middleware can seamlessly orchestrate both low-latency transactional (OLTP) requests and highly complex analytical (OLAP) workloads without requiring native relational support from the underlying engines. Datasets were generated with the TPC-H `dbgen` utility at scale factors SF $\in \{0.01, 0.02, 0.1, 0.5, 1.0, 1.5, 10.0\}$, ranging from approximately 10 MB to 10 GB. The highest scale factor (SF 10.0) evaluates the middleware's performance on a high-end server environment (64-core, 377 GB RAM).

Table 5. OLAP macro-benchmark workload (Phase 2). Queries use the Spark-based federation engine.

Query	Logic (Excerpt)	Complexity
Q1	`SELECT l_returnflag, l_linestatus, sum(l_quantity), sum(l_extendedprice * (1 - l_discount)), count(*) FROM lineitem WHERE l_shipdate <= date '1998-09-02' GROUP BY l_returnflag, l_linestatus`	High: massive aggregation/grouping; stress test for the Spark MapReduce capability over NoSQL data.
Q6	`SELECT sum(l_extendedprice * l_discount) AS revenue FROM lineitem WHERE l_shipdate ...AND l_discount BETWEEN 0.05 AND 0.07 AND l_quantity < 24`	Medium–High: filter-heavy query, validates predicate pushdown.
Q8	Eight-table join over Part, Supplier, Lineitem, Orders, Customer, Nation (twice), Region; computes Brazilian market share per year.	Very High: an eight-way relational join executed within a single non-relational store (Redis or Neo4j), supplying join capability the store lacks natively.

2.5.3. Validation Metrics

The protocol distinguishes *semantic correctness* from *performance overhead*. For semantic correctness, an automated suite enforces a deep-equality check between the middleware result set R_{mw} and the native PostgreSQL result R_{pg}. Two invariants are required: (i) value equivalence, with a floating-point tolerance of 10^{-5} for numerical data, and (ii) type consistency across the JSON serialization boundary. Performance is measured through end-to-end latency (the time between the client query submission and the receipt of the result) and the *Overhead Factor*:

$$\text{Overhead} = \frac{T_{\text{middleware}}}{T_{\text{postgres}}}. \tag{2}$$

A downward trend in this ratio over increasing scale factors indicates that the middleware amortizes its fixed initialization costs over progressively larger payloads.

3. Results

This section reports the empirical evaluation of PolyFuseQL. The analysis follows the two-phase protocol introduced in Section 2.5. Section 3.1 establishes that the middleware preserves data integrity across the entire scale-factor range, before any performance claim is made. Section 3.2 reports OLTP micro-benchmark results that exercise the direct-pipe path and quantify the cost of the routing layer. Section 3.3 presents OLAP macro-benchmark results that exercise the Spark-based federation engine, while Section 3.4 examines vertical scalability and identifies the dominant bottleneck.

3.1. Functional Correctness

Semantic correctness was verified before any latency analysis. For each query and scale factor, an automated suite executed the equivalent statement on PostgreSQL and on the federated middleware, then enforced the value-equivalence and type-consistency invariants of Section 2.5. The outcome is summarized in the functional correctness matrix of Table 6. Within the matrix, the per-cell verdict *Pass* denotes that the individual (query, scale-factor) execution satisfied both invariants for that run, whereas the row-level *Status* aggregates those verdicts across the entire scale-factor range: *Stable* indicates that standard SQL operations (such as the Q1 aggregation or the Q6 filter) passed at every scale factor with no type or value degradation as the data volume increased, while *Robust* additionally confirms sustained success on high-complexity logic. *Status* is therefore an aggregate qualifier over the per-run *Pass* verdicts, not a competing or lower standard. The *Robust* status assigned to Q8 specifically attests to the engine's capability to execute an eight-table

relational join within a single non-relational store without semantic degradation, including the correct handling of `NULL` values during the market-share calculation. This indicates that the system is resilient to data anomalies beyond simple scaling.

Table 6. Functional validation results and execution times across TPC-H scale factors. Note: SF denotes the Scale Factor, where SF 1.0 is equivalent to 1 GB of raw dataset volume. Each cell reports the per-run verdict and execution time (in seconds, for Redis-based federation). 'Pass' denotes that the execution satisfied both the value-equivalence and type-consistency invariants against the PostgreSQL ground truth. The 'Status' column summarizes the row: 'Stable' indicates that all standard operations passed across the entire scale-factor range, while 'Robust' additionally denotes sustained success on the high-complexity eight-table join.

Query	SF 0.5	SF 1.0	SF 1.5	SF 10.0	Status
Q1 (Aggregation)	Pass (22 s)	Pass (63 s)	Pass (133 s)	Pass (580 s)	Stable
Q6 (Filtering)	Pass (19 s)	Pass (53 s)	Pass (120 s)	Pass (490 s)	Stable
Q8 (Multi-Join)	Pass (27 s)	Pass (76 s)	Pass (155 s)	Pass (680 s)	Robust

3.2. Phase 1: OLTP Micro-Benchmark Results

This phase isolates the performance of polymorphic connectors operating in *Direct Mode*, in which the middleware bypasses the Spark execution engine and forwards queries to native database drivers (or to the external translator services for MongoDB and Cassandra). The aim is to quantify the raw overhead introduced by the abstraction layer when no analytics is required. To evaluate this, the micro-benchmark categorizes the operations into four distinct groups displayed on the X-axis of Figure 7: Point Lookups, Range Scans, Filter Operations, and Writes. Note that the two distinct lookup operations evaluated in the raw data (Lookup Customer and Lookup Order) were grouped together under the single "Point Lookups" category to consolidate the assessment of single-record retrieval performance.

Figure 7 reports the absolute end-to-end latency of each engine, accessed through the middleware, on a logarithmic scale; PostgreSQL is included as a familiar relational reference point rather than as a target that the non-relational engines are expected to match. Two distinct questions must be separated. The first is whether the routing layer itself is cheap: because PostgreSQL is also reached through the PolyFuseQL dispatcher, its low latency on point lookups, filters, and writes confirms that AST inspection and strategy dispatch add only marginal delay over a direct driver call, which is the precise sense in which the routing logic introduces negligible overhead. The second question is how the underlying engines differ once a query reaches them, a behaviour governed by their native architectures rather than by the middleware. Redis attains the lowest absolute latency on point lookups ($\sim 10^{-3}$ s), consistent with its in-memory key–value design, while PostgreSQL follows in the low-millisecond range ($\sim 2 \times 10^{-3}$ s). Neo4j incurs the highest latency on point lookups and simple filters (above 10^{-1} s), reflecting the cost of traversing pointers between nodes and edges under ACID guarantees, whereas MongoDB exhibits the most pronounced range-scan penalty (above 1 s), where scanning JSON collections is far more expensive than PostgreSQL's optimized B-tree indices. Cassandra, although accessed through an external translator, avoids this extreme range-scan degradation and stays within roughly one order of magnitude across lookups, filters, and writes ($\sim 10^{-2}$ s), exhibiting an elevated but bounded range-scan cost; this relative consistency reflects its LSM-tree storage, which is optimized for sequential reads and high-throughput ingestion. The architectural conclusion is therefore twofold: the routing layer is effectively transparent, and the order-of-magnitude spread across engines is an intrinsic property of each storage model, not an artefact introduced by PolyFuseQL.

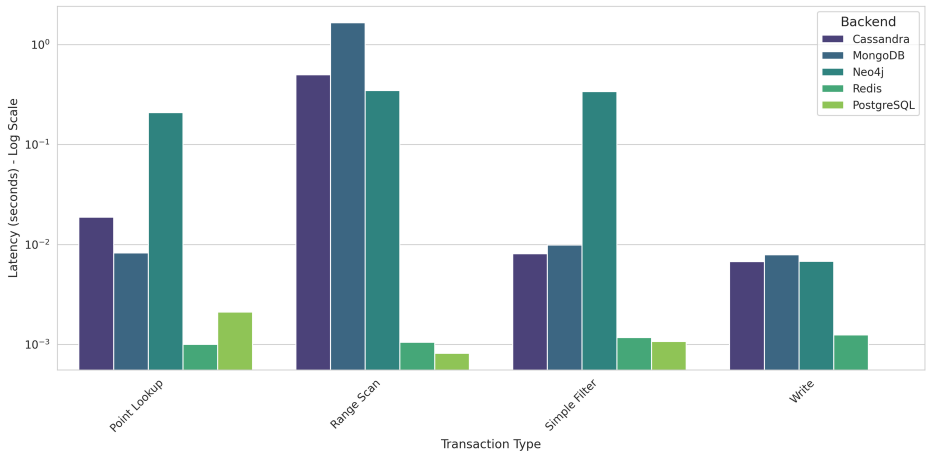


Figure 7. OLTP latency comparison across the five engines (log scale).

These observations validate the architectural Strategy Pattern in the routing layer. By inspecting the AST during parsing, the middleware avoids initializing the Spark Context for low-complexity queries and ensures that the heavy startup overhead of the analytical engine does not impact OLTP throughput.

3.3. Phase 2: OLAP Macro-Benchmark Results

Phase 2 evaluates the federation engine on TPC-H Q1, Q6, and Q8. As discussed in Section 2.4, MongoDB and Cassandra are excluded from these benchmarks because their adapters are lightweight HTTP proxies that lack the distributed ingestion capabilities required to execute Spark-based joins.

3.3.1. Q1: The Spark Tax

Query Q1, the Pricing Summary Report, isolates the cost of bootstrapping the Spark engine. As shown in Figure 8, on small datasets (SF 0.01–0.1, i.e., 10–100 MB) the middleware exhibits higher latency than the native PostgreSQL planner: the latter completes the aggregation in approximately 0.07 s at SF 0.01, whereas the middleware spends roughly 1.34 s preparing the execution environment (Spark Context initialization and Java Virtual Machine (JVM) bridging). This fixed cost, commonly referred to as the "Spark tax", makes PolyFuseQL inefficient for micro-analytics on small datasets but is amortized at higher scales.

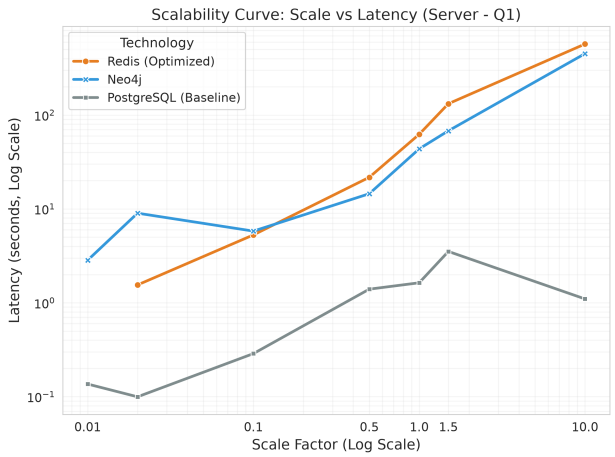


Figure 8. TPC-H Q1 latency on a log–log scale.

3.3.2. Q8: Single-Store Relational Federation

Query Q8 (National Market Share) is the most demanding workload of the suite, requiring a join across eight tables that all reside within the same non-relational backend. While simple aggregation queries inevitably suffer from the Spark startup cost, Q8 demonstrates the primary utility of the middleware: enabling complex relational analytics on stores that natively lack such capability. As shown in Figure 9, latency scales approximately linearly between SF 0.1 and SF 1.5, indicating that the Spark aggregation engine handles increasing data volumes gracefully. At SF 1.5 (approximately 1.5 GB), the HPC server completes the join in roughly 155 s.

The PostgreSQL line in the figure should be interpreted as a *capability gap* rather than a head-to-head performance comparison: native engines such as Redis or Neo4j do not support eight-way SQL joins over their own data, and developers typically simulate such operations through application-side logic. By ingesting each store's data into Spark and operating with standard SQL semantics, PolyFuseQL supplies this join capability uniformly on each non-relational backend, without requiring data to be unified across engines.

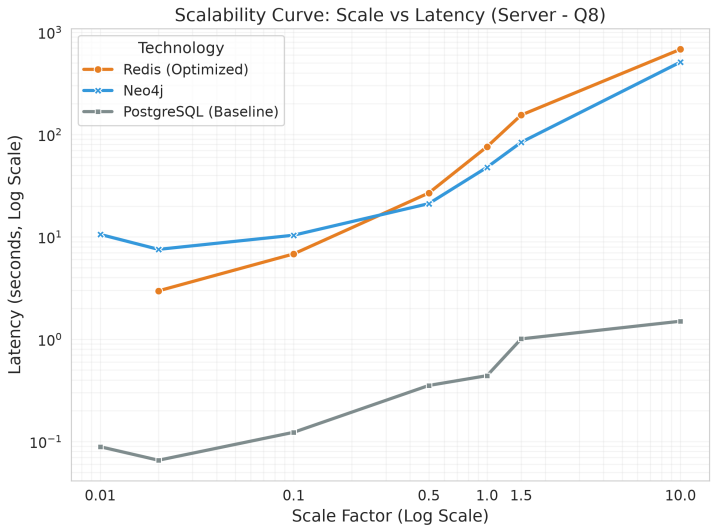


Figure 9. TPC-H Q8 latency on a log–log scale.

3.4. Scalability and Hardware Saturation

3.4.1. Global Scalability Trend

Aggregating Phase 2 results across all scale factors yields the global trend shown in Figure 10, with the federated backends measured from SF 0.02 upward. On a log–log scale, two regimes are visible. At low scale factors the federated curves flatten toward a fixed floor: the Neo4j curve, for instance, is almost horizontal between SF 0.02 and SF 0.1 (remaining near 7 s), because the end-to-end latency there is dominated by the scale-invariant Spark Context startup rather than by data processing. Beyond SF $\approx$ 0.1 the curves enter a positively sloped, approximately linear regime in which latency grows with data volume and processing time overtakes the fixed initialization cost. The middleware processes the 1.5 GB dataset without deviating from this linear regime, confirming predictable scaling for medium-sized analytical workloads.

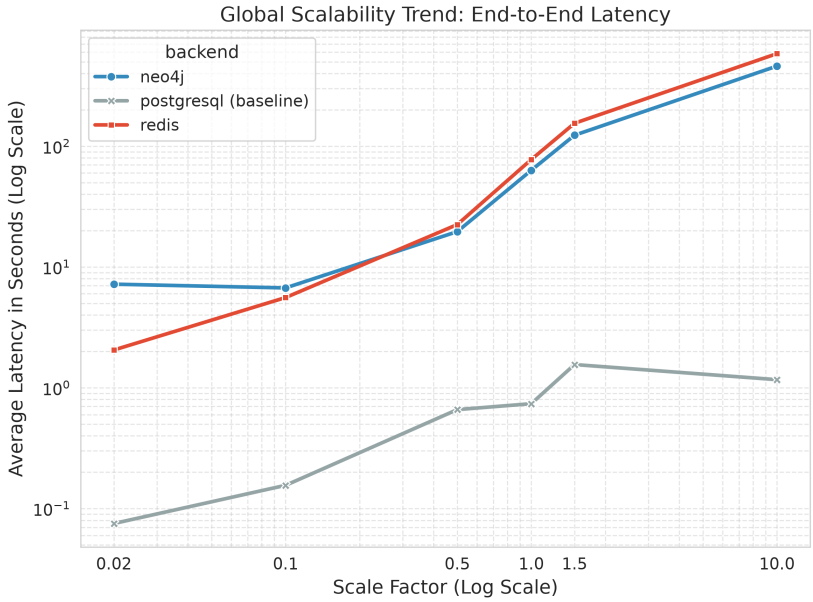


Figure 10. Global scalability trend of end-to-end latency.

3.4.2. Overhead Factor

Figure 11 plots the overhead ratio defined in Equation (2) against the data scale, exclusively for the Spark-enabled backends. Two qualitatively different regimes appear. Neo4j (orange) shows a generally downward trend, dropping from an initial overhead of approximately 160× at SF 0.01 to ∼60× at SF 1.5. This conforms to a "Law of Diminishing Overheads": as the query processing time begins to dominate the fixed startup time, the relative cost of abstraction shrinks. Cypher's ability to filter relationships before returning data to Spark mitigates the data-transfer cost. The single exception to this monotonic decline is the point at SF 1.0, where the Neo4j ratio rises sharply to ∼130× before returning to ∼60× at SF 1.5. Because this isolated excursion breaks an otherwise consistent downward trend and is not reproduced by the neighbouring scale factors, we treat it as a measurement outlier attributable to an uncontrolled runtime perturbation (for example, a JVM garbage-collection pause or transient container scheduling) rather than a structural property of the backend. In contrast, Redis (green) displays an upward trend, escalating from ∼40× at small SF to over 190× at SF 1.5. Because Redis lacks a native query engine for relational logic pushdown, the middleware is forced to fetch essentially all raw data into Spark, and the linear growth of serialization costs (transforming JSON/Hash objects into Spark `Row` objects through Py4J) prevents amortization. As the dataset grows, the time spent on serialization ($\mathcal{O}(N)$) overwhelms the parallel-processing gains, causing the overhead ratio to diverge from the C-optimized PostgreSQL execution. More broadly, the ratios reported here derive from single-run measurements and are therefore subject to such perturbations; quantifying this dispersion through repeated trials and confidence intervals is a methodological refinement we identify as future work.

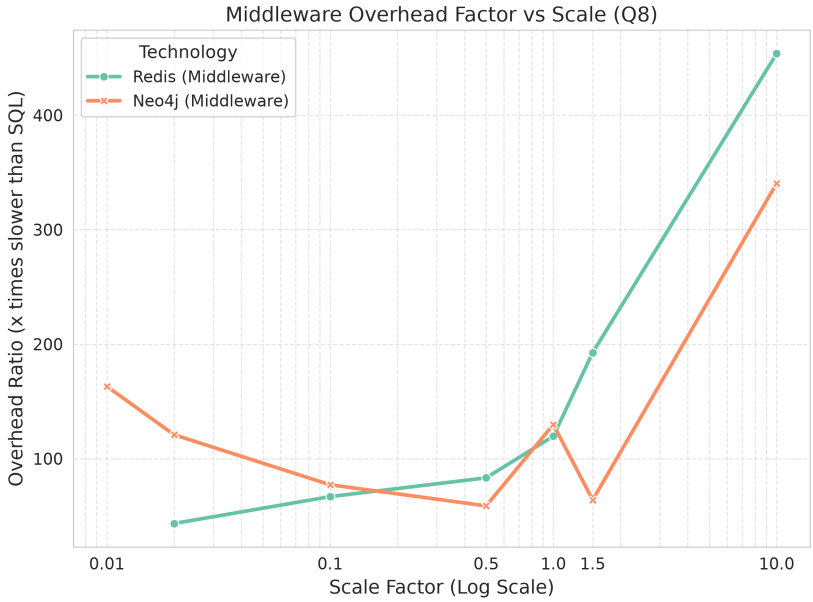


Figure 11. Middleware overhead factor versus TPC-H scale factor.

3.4.3. Vertical Scaling: Workstation Versus HPC Server

To delineate the limits of vertical scaling, the system was benchmarked on Environment A (12 cores) and Environment B (64 cores). The HPC server represents an upgrade of approximately 5.3× in core count and 11× in RAM. However, as shown in Figure 12, performance curves converge at higher scales rather than diverging linearly. For Q8 at SF 1.5, the workstation requires 206.40 s while the server completes the task in 155.01 s, yielding an overall speedup of only ∼1.33×.

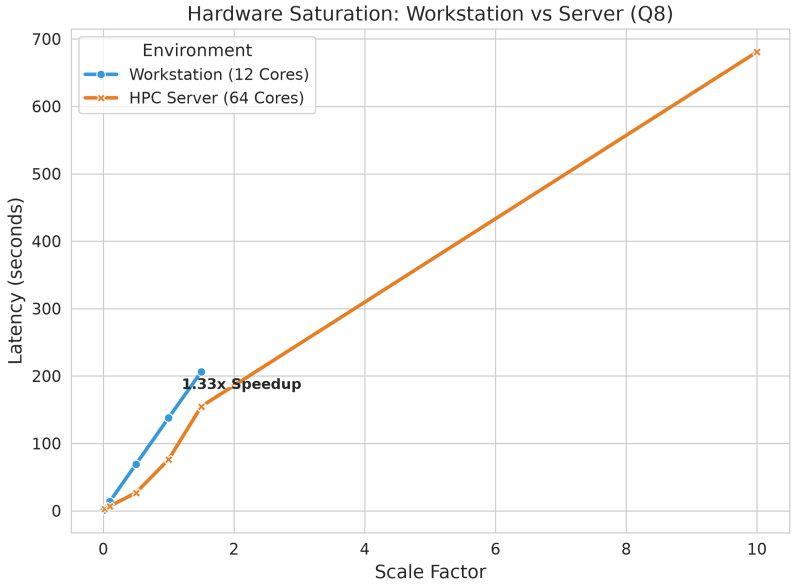


Figure 12. Latency comparison between the workstation and the HPC server for Q8.

This non-linear scaling is a direct consequence of Amdahl's law (Equation (1)). Mapping the theoretical variables to the PolyFuseQL pipeline explains the observed diminishing returns. The parallel fraction f corresponds to the Spark executors operating on worker nodes; the map, shuffle, and reduce stages on the 1.5 GB dataset benefit from the additional cores, and this portion is accelerated by nearly 5.3×. The serial fraction $(1 - f)$ corresponds

to the Python driver and the Py4J gateway: receiving the SQL request, parsing the AST, and serializing the final aggregated JSON result are strictly single-threaded and run on one core regardless of the cores available. The substantial weight of the serial fraction acts as a hard ceiling on the total speedup, explaining why an 11× increase in RAM and a 5.3× increase in cores produce only a 1.33× overall speedup.

3.4.4. Root-Cause Analysis: The Serialization Bottleneck

To pinpoint the bottleneck, Redis was benchmarked with three serialization formats: plain JSON Strings, Redis Hashes, and the RedisJSON module. The aim was to determine whether the overhead originates in the Redis storage layer or in the Python-to-JVM ingestion path. Figure 13 shows that all three formats produce statistically identical latencies on Q8 at SF 1.5 (Redis String ≈ 155.0 s, Redis JSON ≈ 156.6 s). This parity indicates that the bottleneck does *not* lie in the Redis retrieval mechanism but in the serialization bridge between the Python driver and the JVM.

A closer look at the data-packet life cycle explains the result. The Python middleware retrieves results from Redis as native objects (e.g., dictionaries). Aggregation in Spark requires transferring this data to a JVM-resident execution engine. The Py4J gateway pickles each object into a byte stream, sends it through a local socket, and unpickles it on the JVM side as a `Row`. The entire sequence runs on a single thread within the driver, processing one packet at a time regardless of whether the host has 12 or 64 cores. The massive parallelism of the Spark workers therefore remains idle while waiting for the driver to serialize the initial dataset. This architectural constraint is the root cause of the marginal 1.33× speedup observed in Section 3.4.

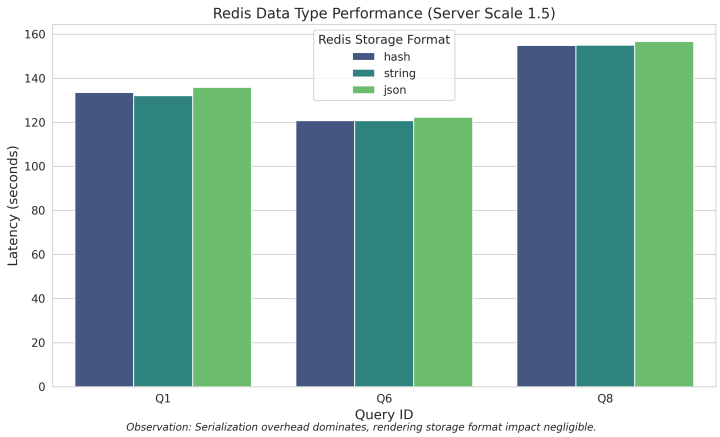


Figure 13. Performance comparison of Redis storage formats at SF 1.5.

4. Discussion

The empirical results expose the dual nature of the PolyFuseQL architecture. The Strategy Pattern effectively isolates low-latency transactional operations from analytical workloads, while the integration of Apache Spark introduces well-defined trade-offs for the latter. This section interprets these findings in the context of the polyglot persistence literature, characterizes the shift in performance bottlenecks, and delineates the operational scenarios in which the artifact is genuinely useful.

4.1. Architectural Trade-Offs

The OLTP results validate the effectiveness of the Strategy Pattern in the routing layer. By dispatching transactional queries directly to native drivers, the middleware achieves sub-millisecond latency and bypasses the Spark initialization overhead that would otherwise dominate small operations. Conversely, the OLAP results expose a familiar

pattern in distributed analytics: initialization costs dominate on small datasets, rendering the system inefficient for micro-analytics, and efficiency emerges only when the dataset size justifies the JVM startup cost. This adheres to a general "Law of Diminishing Overheads" that has been observed in many distributed query engines and confirms that a unified middleware should not attempt to serve both workloads through a single execution path.

4.2. The Bottleneck Shift

A key finding of the study is that the dominant performance bottleneck is no longer I/O but computation, specifically serialization between the Python driver and the JVM executor. This shift, first reported in Section 3.4.4, has practical implications for the design of polyglot middlewares. The Strategy Pattern protects OLTP throughput by isolating it from this bottleneck, as evidenced by the sub-millisecond latencies of Section 3.2. For OLAP workloads, however, the architecture exposes the intrinsic limits of Python–JVM interoperability: the Py4J bridge effectively caps vertical scaling at 1.33× despite a 5.3× increase in cores. The "Spark tax" is an acceptable cost for analytical flexibility, but the system must continue to bypass it for transactional throughput.

4.3. Comparison with Prior Approaches

The contrast with prior unified query interfaces summarized in Table 1 is informative. Canonical-model approaches such as SQLtoKeyNoSQL [9] simplify the addition of new key-based backends but offer limited support for advanced SQL features. Direct-wrapper approaches [17] are simpler to implement but require dedicated maintenance for each backend. PolyFuseQL combines aspects of both: native connectors with Spark integration for Redis and Neo4j (where SQL semantics are non-trivial to map) and lightweight HTTP proxies that wrap pre-existing translators for MongoDB and Cassandra. By validating semantic correctness against PostgreSQL across five engines and reporting measurable performance metrics, the present work also addresses the lack of standardized polyglot benchmarks identified in the survey of Mami et al. [10].

A further question concerns the choice of Apache Spark over out-of-the-box massively parallel processing (MPP) SQL engines such as Presto and Trino, and over native multi-model databases such as ArangoDB. These systems occupy genuinely different design points. Presto [44] and Trino [45] are cluster-oriented federated query engines. However, as pure Java Virtual Machine (JVM) architectures, integrating them directly behind a lightweight, asynchronous Python routing layer introduces severe language barrier complexities. Apache Spark was deliberately selected for this prototype because of PySpark, that aligns with PolyFuseQL's Python based Abstract Syntax Tree (AST) inspecting routing. This alignment made Spark the ideal experimental vehicle for cleanly isolating and measuring the Python–JVM serialization boundary (Py4J) characterized in Section 3.4.4. While Presto and Trino offer high throughput analytical capabilities, they are purely OLAP engines designed for query latencies in the seconds to minutes range. PolyFuseQL does not aim to outperform an MPP cluster on raw analytical throughput; its distinguishing architectural feature is the dual-path design, which protects high-frequency Online Transaction Processing (OLTP) by routing simple statements to native asynchronous drivers for sub-millisecond execution, entirely bypassing the "cold-start tax" of a distributed engine. A head-to-head evaluation of the Spark analytical path against an out-of-the-box Presto or Trino deployment remains a valuable extension for future work. ArangoDB [46], in turn, is a native multi-model engine that stores several models within a single product, rather than a middleware that federates independent best-of-breed engines behind one interface; the two approaches optimize for different goals, single-engine multi-model storage versus many-engine single-interface abstraction, so a direct benchmark would conflate distinct

architectural objectives. We therefore position these systems, together with multi-model benchmarks such as M2Bench [47], as comparison targets for the cross-database federation extension outlined in Section 5, in which tables physically residing in different engines are unified within a single query.

4.4. Operational Viability

Despite the serialization bottleneck, the system remained stable across all scale factors and successfully executed an eight-table TPC-H Q8 over 1.5 GB of federated data in 155 s. This positions PolyFuseQL as a viable solution for medium-scale analytical workloads in which development velocity and architectural flexibility matter more than raw execution speed. The medium-scale ceiling reported here is not a refutation of the SQL interface itself: the difficulty of PolyFuseQL targets is the developer-facing fragmentation of querying five heterogeneous NoSQL engines, each with its own dialect, rather than the raw-volume limits of relational processing. The observed ceiling is an engineering consequence of the single-threaded Py4J serialization bridge (Section 3.4.4), for which a concrete remediation path exists, namely native Spark connectors with predicate pushdown, rather than a fundamental limit of the routing architecture. In particular, the artifact is well suited to business intelligence and ad hoc analytical scenarios in which teams must execute standard SQL across legacy NoSQL silos without engineering a custom Extract, Transform, Load (ETL) pipeline for each query pattern. Conversely, the architecture is unsuitable for high-frequency trading or real-time user-facing applications, where the serialization overhead identified in Section 3.4.4 would prevent the consistent sub-millisecond latency required by complex joins.

5. Conclusions

This paper introduced PolyFuseQL, a Unified Middleware that executes standard ANSI SQL queries across a federated pentaglot environment composed of PostgreSQL, Redis, Neo4j, MongoDB, and Cassandra. Following the Design Science Research methodology and an iterative Scrum-based engineering strategy, the artifact integrates the Strategy, Adapter, and Abstract Factory design patterns with an Apache Spark federation engine. Empirical evaluation through TPC-H Q1, Q6, and Q8 over scale factors from 0.01 to 1.5 supports four central conclusions.

First, the prototype exceeds the original specification by integrating five distinct engines covering the relational, key–value, graph, document, and columnar paradigms, demonstrating that a single SQL interface can effectively abstract storage heterogeneity without mandating rigid schema standardization. Second, the Strategy Pattern proves to be a robust architectural boundary between transactional and analytical workloads: ASTs are inspected at parse time to dispatch simple queries to native drivers (sub-millisecond latency on Redis and PostgreSQL) and to compile complex queries into distributed Spark jobs. Third, shifting ingestion from the driver to the executor yields significant benefits on the Spark path; the optimization analysis in the broader research showed that distributed ingestion reduced Q8 execution time from 688.3 s to 76.15 s, a $\sim 9\times$ speedup, confirming distributed ingestion as a foundational requirement for scalable polyglot federation. Fourth, the semantic-correctness protocol verified that the middleware preserves data integrity to within a 10^{-5} floating-point tolerance when mapping relational logic to NoSQL stores.

Several limitations bound these results. The research design prioritizes the read path; distributed transaction protocols (two-phase commit, Saga) are not yet integrated, so the system offers no guarantees regarding ACID consistency for multi-store updates. By construction, each analytical query is resolved within a single backend: the full TPC-H dataset is loaded into one store at a time, so the federation engine supplies the relational

join and aggregation semantics an individual store lacks rather than joining tables that physically reside in different engines; automated cross-database joins within a single Spark session therefore remain out of scope and are deferred to future work. The unified interface also targets, by design, the standardized relational subset of SQL (selection, projection, joins, and grouped aggregation); model-specific operations such as native graph traversals or vector similarity search fall outside this common denominator and remain reachable only through each engine's native dialect. The OLAP benchmarks focused exclusively on the Spark-enabled Redis and Neo4j connectors, since MongoDB and Cassandra were integrated through HTTP proxies that lack distributed-ingestion capabilities. Finally, the reported latencies and overhead ratios reflect single-run measurements without uncertainty quantification; the isolated SF 1.0 overhead excursion for Neo4j in Figure 11 exemplifies why repeated trials with confidence intervals are warranted, a refinement we leave to future work.

Future work will address each of these limitations along three axes. First, the system will move from a load-then-filter pattern to genuine *predicate pushdown*, translating SQL filters into native Cypher or Redis patterns prior to data transfer in order to minimize network overhead. Second, the proxy adapters for MongoDB and Cassandra will be migrated to native Apache Spark connectors, mirroring the architectural shift that yielded the 9× speedup observed for Redis. Third, a distributed transaction manager, likely based on the Saga pattern, will extend the middleware to the write path, and the orchestration engine will be enhanced to register temporary views from multiple connectors within a single Spark session, enabling true cross-database joins without manual configuration. The Docker Compose deployment will also be replaced by a Kubernetes Operator to support autoscaling of Spark workers in response to query load. Together, these directions outline a credible path to evolve PolyFuseQL from an academic prototype into a production-grade federated database system.

Author Contributions: Conceptualization, A.A.-C. and D.M.-M.; methodology, A.A.-C.; software, A.A.-C.; validation, A.A.-C., D.M.-M., and I.C.; formal analysis, A.A.-C.; investigation, A.A.-C.; resources, D.M.-M.; data curation, A.A.-C.; writing—original draft preparation, A.A.-C.; writing—review and editing, A.A.-C., D.M.-M., and I.C.; visualization, A.A.-C.; supervision, D.M.-M. and I.C.; project administration, D.M.-M. All authors have read and agreed to the published version of the manuscript.

Funding: This research and APC was funded by Escuela Politécnica Nacional.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The TPC-H benchmark dataset used in this study is publicly available through the Transaction Processing Performance Council (https://www.tpc.org/tpch/ (accessed on 5 June 2026)). The source code of the PolyFuseQL middleware and the experimental scripts used to produce the reported results are available from the corresponding author upon reasonable request. The results dataset of the experimentation are available in this repository https://github.com/Middleware-NoSQL/PolyFuseQLResultsDataset (accessed on 5 June 2026).

Acknowledgments: The authors thank the developers of the document-store and columnar-store SQL-translation modules whose components are integrated as black-box services within the federation engine, and gratefully acknowledge the institutional support provided during the development of the artifact.

Conflicts of Interest: The authors declare no conflicts of interest.

Abbreviations

The following abbreviations are used in this manuscript:

ACID	Atomicity, Consistency, Isolation, Durability
API	Application Programming Interface
AST	Abstract Syntax Tree
BASE	Basically Available, Soft state, Eventual consistency
CAP	Consistency, Availability, Partition tolerance
CQL	Cassandra Query Language
CRUD	Create, Read, Update, Delete
DAG	Directed Acyclic Graph
DBMS	Database Management System
DDL	Data Definition Language
DML	Data Manipulation Language
DSR	Design Science Research
JVM	Java Virtual Machine
MQL	MongoDB Query Language
NoSQL	Not Only SQL
OLAP	Online Analytical Processing
OLTP	Online Transaction Processing
RDBMS	Relational Database Management System
RDD	Resilient Distributed Dataset
SF	Scale Factor
SQL	Structured Query Language
TPC-H	Transaction Processing Performance Council Benchmark H
YCSB	Yahoo! Cloud Serving Benchmark

References

1. Sadalage, P.J.; Fowler, M. *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*; Always Learning, Addison-Wesley: Upper Saddle River, NJ, USA; Munich, Germany, 2013.
2. Khasawneh, T.N.; AL-Sahlee, M.H.; Safia, A.A. SQL, NewSQL, and NOSQL Databases: A Comparative Survey. In Proceedings of the 11th International Conference on Information and Communication Systems (ICICS), Irbid, Jordan, 7–9 April 2020; pp. 013–021. [CrossRef]
3. Fowler, M. bliki: Polyglot Persistence. 2011. Available online: https://martinfowler.com/bliki/PolyglotPersistence.html (accessed on 1 March 2025).
4. Swift, J. What Is a Non-Relational Database? Understanding Its Key Features. 2025. Available online: https://www.whatisdatabase.com/what-is-a-non-relational-database-understanding-its-key-features (accessed on 11 November 2025).
5. IBM. What Is a NoSQL Database? 2022. Available online: https://www.ibm.com/think/topics/nosql-databases (accessed on 11 November 2025).
6. Holubová, I.; Klettke, M.; Störl, U. Evolution Management of Multi-model Data: (Position Paper). In *Heterogeneous Data Management, Polystores, and Analytics for Healthcare*; Gadepally, V., Mattson, T., Stonebraker, M., Wang, F., Luo, G., Laing, Y., Dubovitskaya, A., Eds.; Springer International Publishing: Cham, Switzerland, 2019; Volume 11721, pp. 139–153. [CrossRef]
7. Bondiombouy, C.; Valduriez, P. Query processing in multistore systems: An overview. *Int. J. Cloud Comput.* **2016**, *5*, 309–346. [CrossRef]
8. Valentine, H.; Kabaso, B. Design and Development of a Unified Query Platform as Middleware for NoSQL Data Stores. *Int. J. Adv. Comput. Sci. Appl.* **2024**, *15*, 628–639. [CrossRef]
9. Schreiner, G.A.; Duarte, D.; Dos Santos Mello, R. SQLtoKeyNoSQL: A layer for relational to key-based NoSQL database mapping. In Proceedings of the 17th International Conference on Information Integration and Web-Based Applications & Services, Brussels, Belgium, 11–13 December 2015; pp. 1–9. [CrossRef]
10. Mami, M.N.; Graux, D.; Thakkar, H.; Scerri, S.; Auer, S.; Lehmann, J. The Query Translation Landscape: A Survey. *arXiv* **2019**, arXiv:1910.03118. [CrossRef]
11. Kolev, B.; Bondiombouy, C.; Valduriez, P.; Jimenez-Peris, R.; Pau, R.; Pereira, J. The CloudMdsQL Multistore System. In Proceedings of the 2016 International Conference on Management of Data (SIGMOD), San Francisco, CA, USA, 26 June–1 July 2016; pp. 2113–2116. [CrossRef]

12. Cooper, B.F.; Silberstein, A.; Tam, E.; Ramakrishnan, R.; Sears, R. Benchmarking cloud serving systems with YCSB. In Proceedings of the 1st ACM Symposium on Cloud Computing, Indianapolis, IN, USA, 10–11 June 2010; pp. 143–154. [CrossRef]
13. Pereira, F.; Guerra, E.; Rosa, R.R. Patterns for Polyglot Persistence Layer. In Proceedings of the 26th Conference on Pattern Languages of Programs, New York, NY, USA, 24–26 October 2022; pp. 1–8.
14. Shvets, A. What's a Design Pattern? 2016. Available online: https://refactoring.guru/design-patterns/what-is-pattern (accessed on 25 January 2026).
15. Peffers, K.; Tuunanen, T.; Rothenberger, M.; Chatterjee, S. A design science research methodology for information systems research. *J. Manag. Inf. Syst.* **2007**, *24*, 45–77. [CrossRef]
16. Holubov'a, I. Unified management of multi-model data: (Vision paper). In *Proceedings of the ADBIS, TPDL and EDA 2019 Common Workshops and Doctoral Consortium*; Springer: Cham, Switzerland, 2019; pp. 312–319. [CrossRef]
17. Atzeni, P.; Bugiotti, F.; Rossi, L. Uniform access to NoSQL systems. *Inf. Syst.* **2014**, *43*, 117–133. [CrossRef]
18. MongoDB. What Is a Database Management System (DBMS)? 2025. Available online: https://www.mongodb.com/resources/basics/database-management-system (accessed on 22 October 2025).
19. Binani, S.; Gutti, A.; Upadhyay, S. SQL vs. NoSQL vs. NewSQL—A Comparative Study. *Commun. Appl. Electron.* **2016**, *6*, 43–46. [CrossRef]
20. IBM. What Is a Relational Database? 2021. Available online: https://www.ibm.com/think/topics/relational-databases (accessed on 23 October 2025).
21. Jonker, A.; Mucci, T. What Is Structured Query Language (SQL)? 2025. Available online: https://www.ibm.com/think/topics/structured-query-language (accessed on 16 October 2025).
22. w3resource. NoSQL Introduction. 2024. Available online: https://www.w3resource.com/mongodb/nosql.php (accessed on 24 December 2025).
23. MongoDB. What Is NoSQL? NoSQL Databases Explained. 2024. Available online: https://www.mongodb.com/nosql-explained (accessed on 23 January 2024).
24. Amazon Web Services. What Is NoSQL? 2025. Available online: https://aws.amazon.com/nosql/ (accessed on 11 November 2025).
25. Ali, W.; Shafique, M.U.; Majeed, M.A.; Raza, A. Comparison between SQL and NoSQL Databases and Their Relationship with Big Data Analytics. *Asian J. Res. Comput. Sci.* **2019**, *4*, 1–10. [CrossRef]
26. MongoDB. Cloud Explained—What It Is and How It Works-CouchDB and MongoDB Compared. 2026. Available online: https://www.mongodb.com/resources/compare/documentdb-vs-mongodb/couchdb-vs-mongodb (accessed on 14 May 2026).
27. Redis. What Is Redis? In-Memory Database, Cache, and Message Broker. 2026. Available online: https://redis.io/tutorials/what-is-redis/ (accessed on 19 May 2026).
28. Amazon Web Services. What Is Amazon DynamoDB? 2026. Available online: https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Introduction.html (accessed on 19 May 2026).
29. Apache Software Foundation. Apache Cassandra Documentation. 2026. Available online: https://cassandra.apache.org/ (accessed on 19 May 2026).
30. IBM. What Is HBase? 2021. Available online: https://www.ibm.com/think/topics/hbase (accessed on 19 May 2026).
31. Neo4j, Inc. What Is Neo4j?-Getting Started. 2026. Available online: https://neo4j.com/docs/getting-started/whats-neo4j/ (accessed on 19 May 2026).
32. Amazon Web Services. What Is Amazon Neptune? 2026. Available online: https://docs.aws.amazon.com/neptune/latest/userguide/intro.html (accessed on 19 May 2026).
33. Shvets, A. Abstract Factory. 2016. Available online: https://refactoring.guru/design-patterns/abstract-factory (accessed on 25 January 2026).
34. Shvets, A. Adapter. 2016. Available online: https://refactoring.guru/design-patterns/adapter (accessed on 25 January 2026).
35. Shvets, A. Strategy. 2016. Available online: https://refactoring.guru/design-patterns/strategy (accessed on 25 January 2026).
36. IBM. What Is Apache Spark? 2021. Available online: https://www.ibm.com/think/topics/apache-spark (accessed on 25 January 2026).
37. Apache Software Foundation. PySpark Overview—PySpark 4.1.0 Documentation. 2026. Available online: https://spark.apache.org/docs/latest/api/python/index.html (accessed on 25 January 2026).
38. Al-hayanni, M.A.N.; Xia, F.; Rafiev, A.; Romanovsky, A.; Shafik, R.; Yakovlev, A. Amdahl's law in the context of heterogeneous many-core systems—A survey. *IET Comput. Digit. Tech.* **2020**, *14*, 133–148. [CrossRef]
39. IBM. What Is MapReduce? 2024. Available online: https://www.ibm.com/topics/mapreduce (accessed on 19 May 2026).
40. IBM. What Is Docker? 2024. Available online: https://www.ibm.com/topics/docker (accessed on 19 May 2026).
41. Transaction Processing Performance Council. TPC Benchmark™ H (Decision Support) Standard Specification, Revision 3.0.1. 2022. Available online: https://www.tpc.org/tpch/ (accessed on 19 May 2026).

42. Chalacama-Mueses, E.P. Middleware-NoSQL/Traductor-SQL-NOSQL GitHub Repository. 2025. Available online: https://github.com/ (accessed on 19 May 2026).
43. Castro Paredes, R.S.; Andrade-Cabrera, A.; Martinez-Mosquera, D. A Microservice Based Authentication and Authorization Framework for Column Oriented Databases. In *Proceedings of the 1st International Conference on Big Data Analytics & Applications (BDAA' 2025), Innsbruck, Austria, 25–27 November 2025*; Yurish, S.Y., Ed.; IFSA Publishing: Barcelona, Spain, 2025; pp. 18–23.
44. Sethi, R.; Traverso, M.; Sundstrom, D.; Phillips, D.; Xie, W.; Sun, Y.; Yegitbasi, N.; Jin, H.; Hwang, E.; Shingte, N.; et al. Presto: SQL on Everything. In *Proceedings of the 2019 IEEE 35th International Conference on Data Engineering (ICDE), Macao, China, 8–11 April 2019*; IEEE: Piscataway, NJ, USA, 2019; pp. 1802–1813. [CrossRef]
45. Fuller, M. *Trino: The Definitive Guide: SQL at Any Scale, on Any Storage, in Any Environment*, 2nd ed.; O'Reilly: Beijing, China; Boston, MA, USA; Farnham, UK; Sebastopol, CA, USA; Tokyo, Japan, 2023.
46. Lu, J.; Holubová, I. Multi-model Databases: A New Journey to Handle the Variety of Data. *ACM Comput. Surv.* **2019**, *52*, 55:1–55:38.
47. Kim, B.; Koo, K.; Enkhbat, U.; Kim, S.; Kim, J.; Moon, B. M2Bench: A Database Benchmark for Multi-Model Analytic Workloads. *Proc. Vldb Endow.* **2022**, *16*, 747–759. [CrossRef]

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.