Design and Development of a Unified Query Platform as Middleware for NoSQL Data Stores

Hadwin Valentine, Boniface Kabaso
Faculty of Informatics and Design, Cape Peninsula University of Technology, Cape Town 8000, South Africa

***Abstract*—The advancements in technology such as Web 2.0, 3.0, mobile devices and recently IoT devices has given rise to a massive amount of structured, semi-structure and unstructured datasets, i.e. big data. The increasing complexity and diversity of data sources poses significant challenges for stakeholders when extracting meaningful insights. This paper demonstrates how we developed a unified query prototype as middleware using a polyglot technique capable of interrogating and manipulating the four categories of NoSQL data models. This study applied established algorithms to different aspects of the prototype to attain this study's objective. The prototype was subjected to an experiment where varying query workloads were processed. The performance data comprised of application performance index, memory consumption, and execution time and error rates. The results demonstrated that the prototype had a low error rate indicating it's robustness and reliability. In addition, the results showed that the prototype is responsive and able to query the underlying storage system effectively and efficiently. The prototype provides a standardize set of operations abstracting the complexities of each underlying storage system; reducing the need for multiple data retrieval management systems.**

Keywords*—Unified query; polyglot; NoSQL; middleware; query processing; big data

I. Introduction

Information systems in the modern era have shifted the mindset of organizations from application-driven processes to data driven initiatives, i.e. big data. This has led to the creation and adoption variety of NoSQL database technologies, each with its own underlying architectural principles [1, 3]. As a direct result of big data technologies, organizations face the ultimate challenge; how to query structured, semi-structured and unstructured data uniformly? Since numerous NoSQL storage technologies exist; technical consumers have embarked on creating a singular platform for consolidating these heterogeneous data models [17, 27].

The term NoSQL is often confused with "No SQL", the implication being that NoSQL is intended to replace relational SQL database management systems. However, the actual meaning refers to "Not Only SQL" [27]. NoSQL technologies has become the preferred choice for managing big data in this ubiquitous digital realm [1, 3]. The NoSQL philosophy essentially stems from the shortcomings of the relational database management systems. The NoSQL technology stack supports four fundamental data models (1) key-value, (2) column-orientated, (3) document-orientated and (4) graph models [1]. These data models are schema-less in nature, owing to the de-normalize data it holds within the data store [8, 27]. This requires data to be interpreted by the consuming application. A number of challenges start to arise when collating heterogenous NoSQL data schemas from disparate sources since each database system has its respective guidelines and features [17]. This is partly due to the absence of a global schema capable of encompassing the four fundamental data models promoted by NoSQL technologies. As each NoSQL database technology is tailored to serve specific use cases.

In the absence of a global schema for diverse data sets [16, 27], organizations painstakingly develop very specific and rigid implementations to consolidate data from different databases in order to gain valuable and actionable insights from a particular business domain. This activity is traditionally accomplished through data warehousing via ETL's i.e. extract, load and transform [8]. However, the past decade has seen a rise of proposed and propriety unified query solutions to bridge the heterogenous querying gap that exists between database technologies. This data-driven need is inspired by organizations looking to extract key metrics from data to support strategic business initiatives in real time [8, 13, 32]. A common approach used to consolidate disparate data sources is to develop middleware. This is known as a polyglot persistent solution. Polyglot persistent solutions in the context of this paper refer to a system's ability to interact with several database technologies in a multi-faceted way. While there have been numerous successes in these endeavours, the solutions tend to serve very specific use cases and are not easily generalized to the wider IT audience.

A. Aim of Research

The primary objective of this study was to evaluate and validate the effectiveness and efficiency of the developed unified query prototype. It measured the performance of the query process in a holistic manner specifically in terms of query response times, accuracy, reliability and efficacy across different NoSQL storage systems.

B. Significance of Study

This study simplifies the querying process for interrogating multiple categories of NoSQL storage systems. It facilitates seamless data integration, while ensuring data consistency across the supported data models. The prototype segments the boundaries between the varying databases making it easier to extend the family of storage options appropriate for big data [10] applications. Moreover, it assists in outlining the direction for future research initiatives in unified query systems, fostering advancement in the field.

C. Problem Statement

In the absence of a global query instrument, interrogating heterogenous NoSQL storage systems presents complexities when attempting to collate data in a uniformed manner [5, 25]. According to Zhang et al. [9:p.1], the various NoSQL storage models inherently serves by design "different characteristics supported by different database systems and the differences in query syntax rules", thus impeding the pursuit standardization for uniformed query.

Consequently, software engineers spend an inordinate amount of time learning each individual NoSQL database's features. Although a number of research papers have contributed towards developing a unified query model, not many middleware solutions truly encapsulate how key-value, column-orientated, document-orientated and graph data models may be query via a single query mechanism simultaneously. Furthermore, there are unequivocally no standardized data modelling paradigms to the best of our knowledge that exist today able to consolidate the four distinct NoSQL types through normalised methods [15, 9].

An effective and efficient way to overcome this obstacle is to develop a query platform system. This is exactly what this work entailed. Adopting an approach to easily interface with the heterogeneous data models while abstracting the technical details of each storage mechanism. This study provided insights on a developed prototype to determine its feasibility.

D. Contributions

The study contributes to the field of unified query systems in several ways. Firstly, offers a text-based language that's intuitive abstracting the technical barriers of each underlying storage system. Secondly, it presents a novel approach [2] to querying multiple NoSQL systems in a uniform manner by organising established programming design patterns in a unique way. In addition, the modular approach facilitates scalability in terms of extending support for additional storage options without impeding existing supported targeted options. Finally, the prototype's performance results demonstrated that it reduces operational time and costs, considering how it envelops query workloads in a standardized manner.

E. Summary

In this paper, we present the design and development of a unified query platform that acts as middleware for NoSQL datastores. Our research aims to address the challenges associated with querying across heterogeneous NoSQL databases by providing a single query interface that abstracts the underlying complexities. In order to provide clear and concise view of the study, we have organized this paper as follows:

- Section II: Background - Identifies key principles that's required to be present when developing a unified query platform as middleware.
- Section III: Related Works - Discusses related work on existing polyglot solutions within the context of NoSQL databases.
- Section IV: Proposed Architecture - We discuss the architectural and design details of our proposed unified query platform. Furthermore, we describe the composition of the prototype and the software design patterns applied.
- Section V: Experimental Approach - Describes the evaluation method employed to assess the performance of the prototype.
- Section VI: Prototype's Results - We present and analyse the results attained through the experiment.
- Section VII: Discussion – We identified and discussed key findings and repeated themes encountered in the experiment.
- Section VIII: Conclusion and Future Work - Summarizes the key findings and implications of our research. We also outline future work directions.

II. Background

Polyglot query systems generally adheres to layered architectural pattern. However, each layer encompasses a unique class of problems which it aims to resolve [4, 3, 24]. The differences lies within the variety of approaches, methods, principles and technology instantiations to satisfy the intended use cases as shown in Fig. 1. Researchers assessing polyglot systems concur that specific criteria must be met during solution development for it to be deemed acceptable [6, 30]. These criteria form the foundation of unified query resolutions. They are designed to streamline the diversity among various data storage mechanisms [8, 28] and facilitate the abstraction process needed to tackle the complexities inherent in a disparate collection of database technologies.

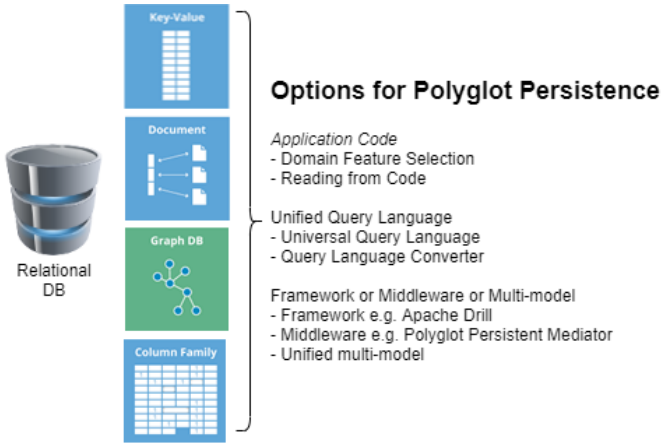


Fig. 1. Approaches to unified query system adopted from [27:p.18].

A. Key Principles

We've identified fives key principles that should be present in these types of systems:

1) Abstract syntax tree: In computer science an Abstract Syntax Tree (AST) acts as a mediator, bridging the gap between conceptualization, design, implementation, and execution, regardless of the underlying technology employed. This concept has found utility across various research domains, including source code compilers, security exploration, anti-plagiarism detection, and code analysis systems [14, 31].

Within the scope of this study, an AST is employed within the query parser to ensure that commands adhere to syntax, semantic, and lexical rules, thereby guaranteeing that the command constitutes a well-formed statement [19].

2) Schema consolodation: A fundamental aspect of developing unified solutions is obtaining a comprehensive understanding of the schema information for each underlying storage mechanism [15]. This is commonly referred to as metamodeling. Despite the promotion of NoSQL as schema-less because of its efficient handling of unstructured data, there indeed exists a schema. Depending on the vendor, schema constraints may be enforced, which the consuming application must adhere to.

3) Query translation: Arguably, the most crucial aspect of any unified solution is to generate native queries capable of interrogating NoSQL storage models [23, 32]. It's important to note that this feature is heavily influenced by the unified approach shown in Fig. 1. However, conceptually, regardless of the approach, it facilitates the generation of native queries that can execute on their respective NoSQL databases.

4) Database integration: In every unified query solution, provision must inevitably be made for communication with the targeted databases [17]. NoSQL databases commonly employ diverse protocols as communication mediums to access the data source [9, 23]. These communication protocols range from HTTP(S) to TCP/IP, typically employing an adaptor or driver that implements a generic interface for database connection. An intriguing observation noted during this study is a direct correlation between the primary communication protocol and query language. Depending on the protocol, the query interrogation mechanism may access the database data via an API endpoint or some form of lower-level network protocol for data exchange.

5) Output management: To present data from various storage systems uniformly, unified query systems typically employ two approaches: Global-as-View (GaV) and Local-as-View (LaV), where data unification is facilitated by a mediator [8, 13]. It's important to note that this also contributes to the aforementioned key features. This feature is categorized as a mediator, an intelligent layer that possesses structural knowledge of the local data stores. GaV integrates schemas of the underlying local data stores, providing a unified view of heterogeneous structures. Conversely, LaV amalgamates local schemas to form a global view.

III. Related Works

Polyglot solutions like BigDawg aims to leverage the relative strengths of underlying DBMSs to effectively process data [30]. This solution embraces three types of data models: key-value, relational, and array stores. The architecture of BigDawg primarily focuses on query processing rather than query construction. Its objective is to utilize key features to achieve optimal performance and produce the most comprehensive result set. To achieve this objective, the architecture incorporates features such as islands, shims, and cast, as illustrated in Fig. 2 [6, 30].

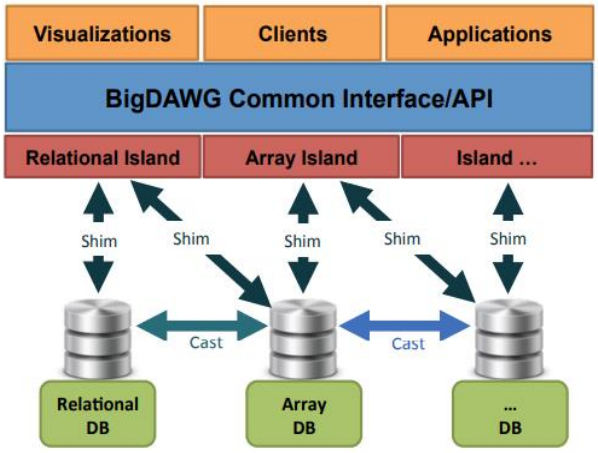


Fig. 2. BigDawg architecture [30].

An island is associated with a specific data model and a set of query language features for the storage engine it intends to support. A shim acts as a communication bridge between the island and the storage engines. A cast facilitates data migration from one storage engine to another. The API directs inquiries to the middleware, which handles query execution and data migration through casts [4]. The middleware comprises various modules, including the query planner, performance monitor, and executor. These modules validate the semantic correctness of queries and route them to the appropriate storage mechanism for execution.

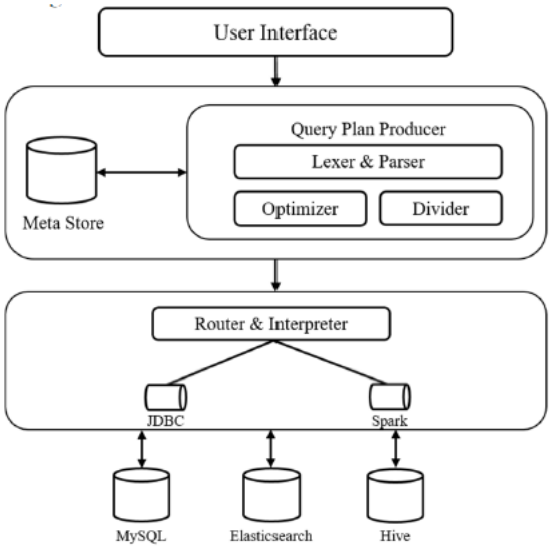


Fig. 3. Unified SQL query middleware architecture [9].

Zhang et al. [9] introduced a solution that employs middleware to execute queries on multiple heterogeneous databases through a unified interface using standard SQL syntax. Their segmented architecture, depicted in Fig. 3, separates the initial query from the targeted queries via an abstract syntax tree. This tree is responsible for verifying if the initial query aligns with the requirements of the respective heterogeneous databases. While the article mentions that the middleware supports a pluggable interface for new data sources,

it does not detail how this would impact the abstract tree and computing layer. The provided middleware comprises three main components: a syntax parsing layer, a computing engine, and a data layer. The syntax layer validates a unified query against a customer abstract syntax tree. Native queries are then generated based on a meta store, which delegates them to the computing engine for execution on the data layer.

NoDA, a lightweight implementation, acts as an intermediary layer between applications and targeted NoSQL databases, including MongoDB, HBase, Redis, and Neo4j [23]. This middleware offers a generic set of operators such as sorting, filtering, and aggregation, aiming to efficiently execute queries using the Apache Spark open-source data analytical framework. Although NoDA is categorized as a polyglot implementation, it simplifies complexity by separating the rule engine, which validates syntax and semantics of the unified query, from the abstract layer using a third-party tool.

Cox et al. [29] introduced the Translator Query Language (TranQL), a solution that federates biomedical ontologies within a framework. Their study is grounded in real-world case studies. TranQL utilizes natural language to map to queries, generating targeted queries on various graph data models. An essential component of the framework is the Translator KGS API, which employs the shared schema RDF concept to express queries as Biolink data model, a hierarchical medical ontology at a high level. This API maps a network of knowledge graphs as a coherent whole, forming the basis for TranQL as a unified query pattern by interconnecting federated knowledge graph data models through curated links across entities.

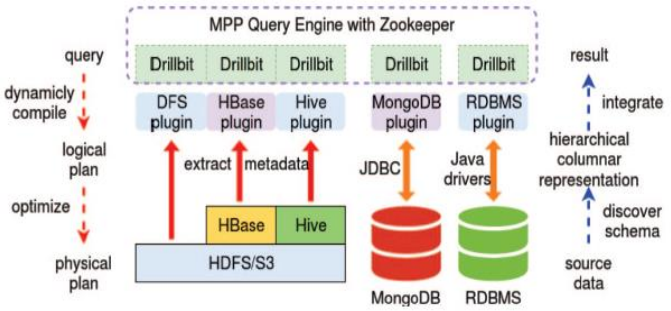


Fig. 4. Unified SQL query middleware architecture [28].

Apache Drill is a fully distributed open-source software framework designed for large-scale analysis in data-intensive applications [16]. It specializes in processing extensive datasets efficiently by executing tasks in parallel. The Apache Drill solution leverages in-memory data representation in JSON and Parquet formats for rapid data manipulation operations. Additionally, its MPP (Massively Parallel Processing) query engine dynamically compiles and recompiles data queries on the fly to maximize performance, relying on parallelism [28]. Similar to BigDawg's implementation, Apache Drill supports various data models accessed through a comparable mechanism as illustrate in Fig. 4. However, instead of islands, it utilizes plugins to connect to different storage engines and file systems via the Drillbit component [6]. Drillbit serves as a background component orchestrating the optimal execution query plan. The query executions are partially rendered on an execution tree and brought into memory.

CloudMdsQL is recognized as a multistore system capable of querying multiple databases through its SQL-Like unified query construct [4, 6]. Supporting relational, NoSQL, and HDFS storage mechanisms, CloudMdsQL is designed to leverage the inherent features of each supported heterogeneous data store [23]. The abstract layer catalogs the semantics rules of the supported data stores, enabling the optimization of native queries. This allows the construction of native queries through a relational query framework for targeted executions. The results of embedded invocations are converted into an intermediary table for distributed processing.

A. Evaluation Approaches of Polyglot Systems

It's important to note that this paper does not encompass all unified solutions, as the objective is not to describe every possible solution. Rather, we aim to introduce readers to the distinguishing components of these solutions and the use cases it aims to satisfy. Research papers proposing unified query solutions understandably prioritize the overall utility of the artifact. Much emphasis is placed on practical considerations such as query workloads, indexing, and partitioning, which are integral to query processing [13, 32].

The described polyglot solutions are tailored to address different use cases. For instance, Apache Drill excels in processing vast amounts of data for analysis, requiring robust hardware as it loads data into memory for rapid retrieval [6]. Conversely, CloudMdsQL and BigDawg aim to leverage the full capabilities of supported databases' native features to process data, thereby providing users with enhanced native capabilities. TranQL serves as a federated query system for Biolink data using a topology of graph stores. Each of these solutions comprises a collection of individual isolated components targeting the supporting databases. These components operate independently, acting as intermediaries between the middleware layer and the database, except for BigDawg, which allows data integration between silos.

Other solutions, such as NoDA, are less intricate, as it follows the basic principles. This prototype primarily focus on the query construct [23, 9], which aligns with the goals of this study. Although the middleware supports the four primary categories of NoSQL data models, it can only query one underlying database at a time. The authors highlight this limitation, underscoring that the prototype primarily emphasizes the system's capability to access data through its connector. Zhang et al. [9] on the other hand, is limited to select queries and does not accommodate evolving schemas. Additionally, the use of wildcards within the middleware may introduce suboptimal practices and potential runtime issues stemming from datatype and schema mismatches.

IV. Proposed Architecture

This section presents the methods employed to design and develop the prototype. The goal of this prototype was to provide a high-level unified query platform that is database-agnostic capable of querying data across the four distinct types of NoSQL storage models simultaneously [17]. The prototype provides a query language that offers a consistent a set of syntax, semantics and data operations to express queries in a generic manner for the targeted storage models.

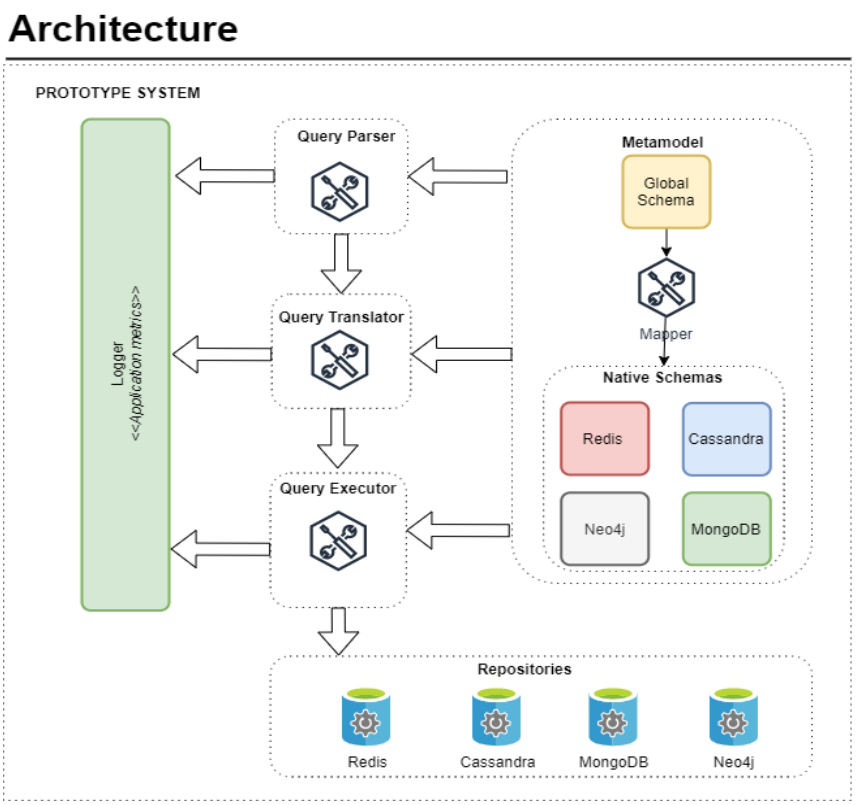


Fig. 5. Prototype: architectural overview.

The prototype for the unified query platform had the following basic requirements, (1) develop a custom parser that accepts a SQL-like query as input, (2) develop a metamodel describing the each of the native schemas as well as the global schema, (3) build a translation engine that accepted the parser's output and generated a native queries, (4) build a an executing layer that accepts the native queries as input and executes it on the supported NoSQL data stores, and finally a (5) logging mechanism to audit performance and functionality of the prototype. This is encapsulated in Fig. 5. showing the overall architecture and the interactions between the various components.

Design Science Research: This paper used DSR methodology to ascertain the necessary knowledge to build the prototype. DSR is a problem-solving architype that creates knowledge on the design process and product concurrently [12]. The study subscribed to the seven guidelines proposed by Hevner et al. [2]. The design and architectural choices made was influenced by existing literature and the empirical insights during the development and evaluation phase of the prototype. The iterative nature facilitated the authors of this study to test and refined the prototype based on ideal approaches and current shortfalls on unified query platforms. The constant feedback loop guided the software development lifecycle [18]. The act of the repeated circumscription process influenced the prototype construction until design requirements in Table I were satisfied. A student database for each instance of the supported NoSQL storage systems was created shown in Appendices A and B to interrogate.

The study employed a mathematical abstraction, wherein *q(n)* symbolizes the native or targeted query for each instance category of a NoSQL database [3]. Furthermore, *DS* represents the data source which consolidates the four supported types of NoSQL storage data models. i.e., *GR* - Graph, *KV* - Key-Value, *DO* – Document-Orientated, *CO* - Column- Orientated data stores. The data source is represented as $DS \rightarrow GR \cup KV \cup DO \cup CO$, indicating which the NoSQL data storage models are supported. The query parser ensures the unified query conforms to the signature of the abstract syntax tree, whereby the unified query is required to prove it conforms to the lexical (*lex*), semantic (*sem*) and syntactic (*syn*) rules of the prototype.

$$S_{lss} = \sum_{i=0}^{n-1} k^i, k < (lex[\mathrm{i}] \wedge sem[\mathrm{i}] \wedge syn[\mathrm{i}]) \quad (1)$$

The query translator verifies if the targeted data model, *dm(k)*, specified in the unified query is an element of the data source:

$$dm(k) = \begin{cases} 1, & if(k \in DS) \\ 0, & otherwise \end{cases} \quad (2)$$

Once the system has established that the data model is supported by one or more elements of the data sources, it is required to generate the targeted or native query, *t(k)*:

$$t(k) = \begin{cases} 1, & if(dm(k) \vdash (GR \mid KV \mid DO \mid CO)) \\ 0, & otherwise \end{cases} \quad (3)$$

The query executor subsequently directs *t(k)* to appropriate NoSQL database instance to be executed. If $q(n) = \prod_{k=1}^{DS_n} k, \exists_n[\emptyset, n].\, t(k).\, dm(k)$ *holds dm(k)*, the native query is executed on the target storage model. Finally, the object mapper

wraps the output of each target query into a result, $r_i = o \in [q(0), \dots q(n)]. (k \geq q(k))$.

A. Design Requirements

A set of requirements were identified to achieve the envisioned design goals shown in Table I. Each requirement was linked to a component responsible for a specific functionality in realising a unified query platform. These components function are akin to "spokes in a wheel," relying on each other to accomplish the functional objectives.

TABLE I. PROTOTYPE DESIGN REQUIREMENTS

Prototype Design	
Components	***Requirements***
Metamodel repository	Create a metadata schema denoting Redis.
	Create a metadata schema denoting Cassandra.
	Create a metadata schema denoting MongoDB.
	Create a metadata schema denoting Neo4j.
	Create a global metadata schema.
Query parser	Build a lexer for input characters.
	Build a query syntax tree.
	Build a semantic engine.
Query translator engine	Build Syntax and Semantic Matching engine.
	Build Feature Mapping engine.
	Build Query Optimization engine.
Query Executor	Build a database adapter for NoSQL databases.
	Map native results to a global view.
Log Mechanism	Build data collection mechanism.

B. Prototype Construction

The first step was to determine how context and meaning can be given to the prototype's intended query language [14, 19]. Therefore, the prototype facilitates three commands: Fetch, Add and Modify (Appendix C). The nature of these commands is intrinsic, as their names suggest. The Fetch command retrieves data, the Add command inserts data, and the Modify command updates data across the supported NoSQL storage system concurrently. Determining the fundamental intent of the query serves as the initial step in shaping the unified query platform.

1) Query parser: To operationalise the commands, an AST was built within the query parser component. A text-based language was the preferred design choice to serve as the prototype's unified query as its familiar to consumers interrogating data and will most likely drive greater adoption [19]. The elements of the query language within the prototype were deconstructed into an organized tree-like structure. The prototype incorporates an embedded lexer feature within the query parser component, which scans the text and generates a stream of tokens, serving as input for the subsequent parsing phase. During the parsing phase, the stream of tokens produced as shown in Table II by the lexer is systematically examined, and the abstract syntax tree (AST) is constructed based on the grammar rules of the unified query language. The keywords and identifiers guided informed the prototypes query intent, path and code generators to executed the appropriate native query. On this basis the necessary tokens is generated are representative of the unified query's meaning and purpose.

TABLE II. PARSER'S LEXICONS

Keywords	**Parser**	
	Lexicons	***Input Text***
	FETCH	FETCH
	MODIFY	MODIFY
	ADD	ADD
	PROPERTIES	PROPERTIES
	DATA_MODEL	DATA_MODEL
	FILTER_ON	FILTER_ON
	ORDER_BY	ORDER_BY
	RESTRICT_TO	RESTRICT_TO
	TARGET	TARGET
	ASC	ASC
	DESC	DESC
	LAND	AND
	LOR	OR
Identifiers	REFERENCE_ALIAS	Identifier preceding 'DOT'; example: ***t.****property*
	REFERENCE_ALIAS_NAME	Identifier succeeding 'AS'; example: *t.property AS* ***alias***
	REFERENCE_MODEL	Identifier succeeding 'AS' in DATA_MODEL; example DATA_MODEL { *data AS* ***dataAlias*** }
	PROPERTY	Referenced column\attribute name
	JSON_PROPERTY	A JSON referenced column\attribute name
	TERM	Identifier succeeding 'FILTER_ON'; example FILTER_ON { ***term*** = *'1'* }
	DATA	Identifier succeeding 'DATA_MODEL'; example DATA_MODEL { ***data*** }
	NAMED_VENDOR	Identifier of database vendor; example ***neo4j, mongodb, cassandra, redis***
	AS	AS
	LEFT_CURLY_BRACKET	{
	RIGHT_CURLY_BRACKET	}
	LEFT_BRACKET	[
	RIGHT_BRACKET	]
	LEFT_PAREN	(
	RIGHT_PAREN	)
	COMMA	,
	DOT	.
	NSUM	Nsum
	NAVG	Navg
	NCOUNT	Ncount
	NMIN	Nmin
	NMAX	Nmax
Operators	EQL	=
	LSS	<
	GTR	>
	GTE	>=
	LTE	<=
Literals	NUMBER	1,2,3,4,5,6,7,8,9,0
	STRING	Aa,Bb,Cc,….Zz

The prototype employs a parser combinator technique, where multiple parsers are accepted as input to create a new parser as output. This technique enables the prototype to modularize sections of the query language by recursively traversing through the token stream and using demarcating locations. These demarcated locations assist the program in indicating where the parser should start and stop. Following a recursive descent strategy, the parser inspects terminal and non-terminal symbols based on the syntactic rules governing the grammar of the unified query. This process results in grouping a disjointed set of nodes [11]. A lightweight library called Superpower was utilised to facilitate the construction of token-driven parsers embedded directly in the source code [21]. This library is an extension of Sprache, a text-based parsing framework that does not require any additional build tools or runtime configurations. According to its documentation "it fits somewhere in between regular expressions and a full-featured toolset like ANTLR" [20]. A demonstration of the lexical activity reveals how the tokens are generated by the prototype as per a given input (Appendix D). Once the unified query has proven to be well-formed by the parse, the prototype delegates the query to the metamodel to determine if the actual properties are defined in the global schema.

2) Metamodel: The function of the metamodel is to bridge the gap between the unified and native schemas [6, 16, 17]. It plays a crucial role in the solution by revealing the physical structures of the native schemas and the conceptual structure of the global schema. The global schema contains instructional configurations to the native schema, indicating the relationship between the models. The prototype's metamodel catalogues each storage mechanism's schematics, data types, and indexes. Additionally, it assists the query parsing mechanism by performing basic validations to ensure that the specified fields are supported by the unified query data model. It aids the query translator in resolving native references at runtime and assists in generating the appropriate native query constructs. To some extent, it informs the query processing engine about the optimal query to create when inspecting relevant native storage mechanism schematic information such as indexes and unique keys.

3) Query translator: The translation engine has several features for the query processing and the creation of executable native queries:

- Syntax and Semantics Matching
- Feature Mapping
- Query Optimization

a) Syntax and semantics matching: Any unified query polyglot system targeting multiple types of databases, will innately have different syntax and semantics compared to the native query languages [5]. Hence, the prototype's query translation engine finds the equivalent meaning and grammar of the supported databases in order to successfully build executable queries. Finding the equivalent match ensures the intended meaning and functionality is preserved during the conversion process of unified query. In addition, the syntactic translation involves converting the unified query's expressions, keywords, identifiers, literals and operators to match the syntax of the native query language [23]. This ensures the adherence to each supported database, safeguarding against unintended results once the generated query is eventually natively executed.

b) Feature mapping: The prototype's query language in some instances does not have the direct equivalent features or constructs in the targeted native query language. It attempts to preserves the anticipated functionality while still creating a converted query that may be executed. In general, features for database management systems are naturally influence by the applicable use cases [1, 3]. In the instance of the key-value database, Redis, aggregation amongst other features are not natively supported in its database management as shown in Table III. Therefore the prototype requires an additional abstraction layer for the Redis data store to circumvent this issue which currently does not support.

TABLE III. PROTOTYPE VERSUS EQUIVALENT NATIVE DATA STORES FEATURES

Prototype	Redis	Cassandra	MongoDB	Neo4j
Aggregation				
NSUM		X	X	X
NAVG		X	X	X
NMIN		X	X	X
NMAX		X	X	X
NCOUNT		X	X	X
Filtering				
WHERE	X	X	X	X
AND		X	X	X
OR		X	X	X
JOIN				X
RESTRICT		X	X	X
Sorting				
ASC		X	X	X
DESC		X	X	X
Projections				
*No explicit command			X	X
Operators				
'=', '+', ' -', '*', '/'	X (only '=')	X	X	X
Comparators				
'<', '<=', '>=', '>'		X	X	X

The translation engine maps these features to appropriate native constructs, ensuring the preservation of the expected functionality. Specialized strategies for each of the inherent data stores was built, thus establishing clear boundaries between the various NoSQL translation layers.

c) Query optimization: The query optimizer plays an key role in the efficiency of the polyglot solution. The prototype employs an approach concerned with delegating the heaving lifting to the targeted database of query filtering, sorting, projections and aggregation where applicable [32]. As a consequence, it aims to shift the I/O, memory and CPU

processing power to the respective DBMS reducing the computational footprint on the prototype. Additionally, pushing operations such as projections and filtering closer to the data source, reduces the network bottleneck when data is transferred between the prototype and the corresponding NoSQL data stores [27].

4) Query executor: This component is responsible for natively running queries produced by the query translation engine against the respective NoSQL data sources. It establishes the database connections, the authentication procedures and data transfer between the unified query platform and the data source, similar approaches to BigDawg, NoDA [9, 23]. The prototype's query executor coordinates the concurrent executions of the respective native queries amongst the NoSQL data stores based on the targets specified in the unified query. It splits the executable queries into multiple processing units by creating threads for each one. For each data source, the executor collects the query results. It performs any necessary data mapping to present a consolidated result. Any errors and exceptions that may occur during query execution process provides the appropriate error messages back to the query interface.

5) Logger: The experiment embeds metrics directly into the prototype. Utilizing an open-source library known as App Metrics (app-metrics.io, 2021), the prototype measured various performance aspects of the components within the unified query solution. The report modules provided a set of libraries through which the unified query parser, translator, and executor could be scoped.

C. Design Integration

Ultimately, the prototype needed specific non-functional aspects to finalize the solution. The study identified the (i) query intent, (ii) query path, and (iii) query generator as key elements comprising the non-functional requirements. Each of these elements was implemented using established programming design patterns. Fig. 6 depicts the alignment of the parser, translator, and executor components with the non-functional requirements. It illustrates the path of the unified query through each stage of the query processor and, importantly, how the design programming patterns are encapsulated within this process.

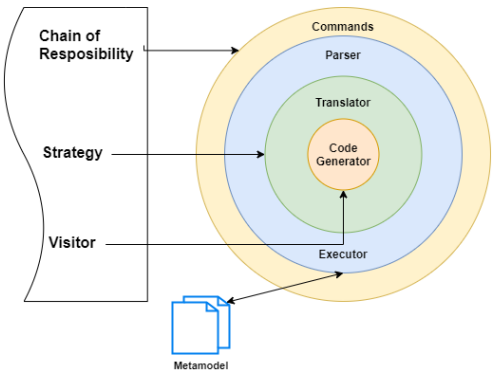


Fig. 6. Prototype design patterns and components.

1) Query intent: Determining the intent of the unified query is crucial as it directly influences the expected outcomes. This necessitates the solution to align the prototype commands with the corresponding features of each native system. Once the query intent is identified, the prototype directs the query to follow the appropriate query path. The chain of responsibility design pattern was selected, wherein the prototype dynamically determines which command to execute at runtime [22]. The prototype defines *Fetch*, *Add*, and *Modify* commands as handlers (see Fig. 7), each responsible for interpreting its respective request. These handlers share a common interface, which is tasked with dispatching client query requests to the appropriate command handler based on the data inquiry [26]. The command handlers contain the query parser and translator logic.

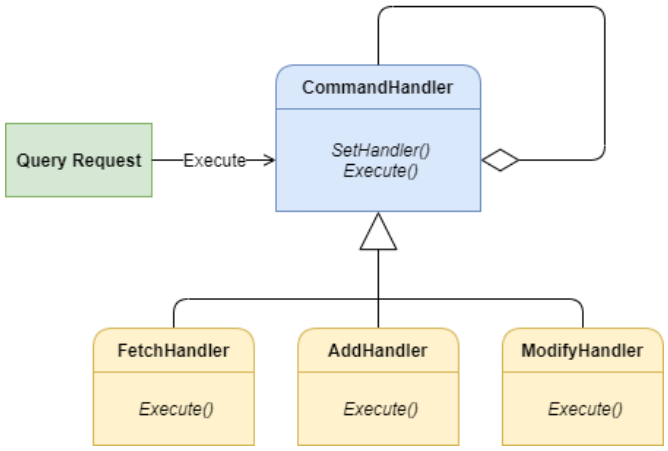


Fig. 7. Query intent: chain of responsible design pattern.

This pattern has found widespread application in scenarios where system messages dictate the execution result [7]. Upon the program's initiation, new instances of each command type are created, resulting in a chain of objects. To enhance the efficiency of the execution processing chain of objects, the collection of concrete handlers, i.e., command handlers, was organized as a dictionary, with the command types serving as unique keys. The query request passed to handlers is tagged with the appropriate command type, which is then used to locate the corresponding handler in the execution chain. In instances where the command is not found in the dictionary, no action is taken, and the unified query request is aborted with an error message. The prototype implements the chain of responsibility in the following manner as shown in Table IV. Q denotes the intent of the query language. Each of the commands within the unified query are denoted as f for *Fetch*, m for *Modify* and a for *Add*. Therefore the command, represented as *cmd*, must always be present in the unified query. Thus one can conclude that the *cmd* is a subset of Q, i.e. $cmd \subseteq Q$. N represents the collection of nodes within the AST, $N \rightarrow \{n_1, \ldots, n_n\}$. The nodes are assigned an array of instantiations expressing the mechanical parts of the query. The prototype is able to discover command instantiations thereby enabling the correct handler to be invoke. This act facilitates the prototype to realise the intent.

TABLE IV. CHAIN OF RESPONSIBILITY PATTERN PSEUDOCODE

Algorithm : Query Intent

```
≔ QueryIntent(q)
if q ∈ Q do
  if q.cmd ∈ f do
    f(N)
  else if q.cmd ∈ m do
    m(N)
  else if q.cmd ∈ a do
    a(N)
  else
    InvokeError
```

2) Query path: In anticipation of the native query generators, the query path determines the supported NoSQL storage systems to target and the components to execute. This guarantees the generation of the correct native query based on the unified query intent. The strategy pattern was employed, ensuring the appropriate algorithm is enforced based on the query elements specified in the target clause within the AST. Each of the supported NoSQL data storage models was defined as descendants within a family of algorithms shared by the same ancestor [22]. In the prototype, each of the supported NoSQL data models is represented as specialized classes responsible for constructing a collection of visitors to be executed by the query generator. The prototype takes the query intent as input and matches the command and storage target to the relevant strategy. During the translation process, the repository metamodel is utilized to identify the equivalent native field for the unified field. If no matches are found, the field is excluded. The prototype intentionally constructs a collection of class instantiations, represented as visitors, to closely mimic the structure of the native query languages it needs to create. Finally, once the native queries are generated by the query generator, the strategy pattern sends the output back to the calling method for execution.

The query path strategy implementation considers the target models specified in the unified query once the command has been established (see Fig. 8). The target models as shown in Table V. where *rs* represents redis, *ms* mongodb, *cs* cassandra and *ns* neo4j. The translator component *T*, accepts the target models as input thus directing the appropriate queries to be generated. The supported NoSQL databases are implemented as concrete classes subscribing to a single collection as they all share a common interface, $SP \rightarrow \{sp_1, \dots, sp_n\}$. The classes inherits from a base strategy class where $sp_n \in (rs \mid ms \mid cs \mid ns)$. The strategies are preloaded within *T*. Therefore, to execute the relevant strategy, it must exist with the translation component $sp_n \ni T$. The data source *DS* is indicative of the underlying NoSQL database categories, *KV* : key-value, *CO* : column orientated, *DO* : document orientated and *GR* : graph data stores. The output *n*, generated by the translator, denotes the native query. This eventually runs on the targeted NoSQL database completing the execution path.

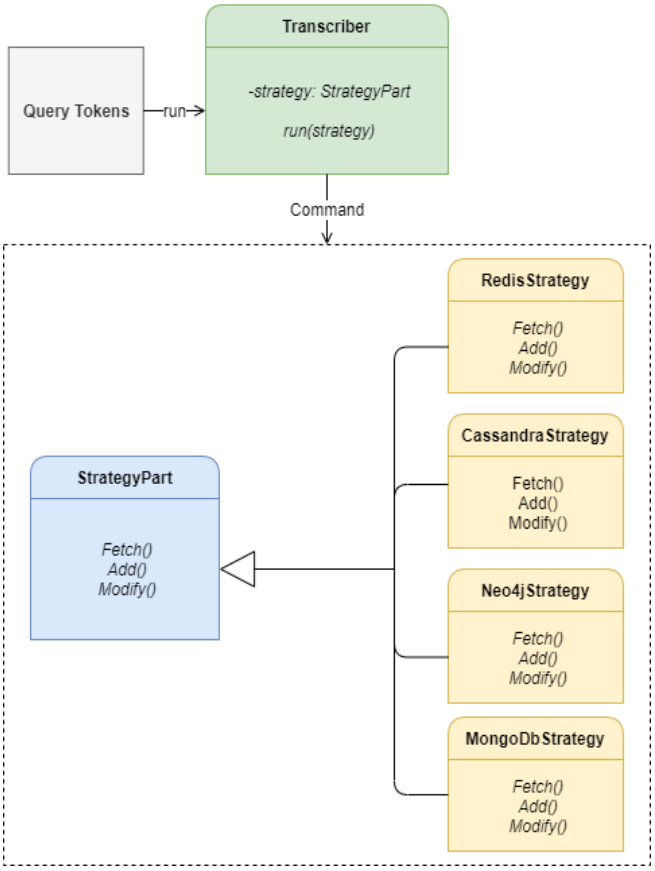


Fig. 8. Query path: strategy design pattern.

TABLE V. STRATEGY PATTERN PSEUDOCODE

Algorithm : Query Path

```
I → QueryIntent(q)
≔ QueryPath(q)
if q ∈ I do
if e → ∃(i.cmd) do
target storage → q.DS
for each sp ∈ SP(target storage) do
if sp ⊆ KV do
n → T.Run(rs)
if sp ⊆ CO do
n → T.Run(cs)
if sp ⊆ DO do
n → T.Run(ms)
if sp ⊆ GR do
n → T.Run(ns)
return n
```

3) Query generator: The query generators separate the processing logic from query components. To generate the native NoSQL queries for the prototype, the visitor pattern was employed. It is invoked by the query translator component. The native query elements are represented as "visitors" which directly correspond to elements of the tokens generated by the query parser. This pattern is highly effective, as it enables class instantiation to add functionality without altering the structure of the class, thereby ensuring scalability [22].

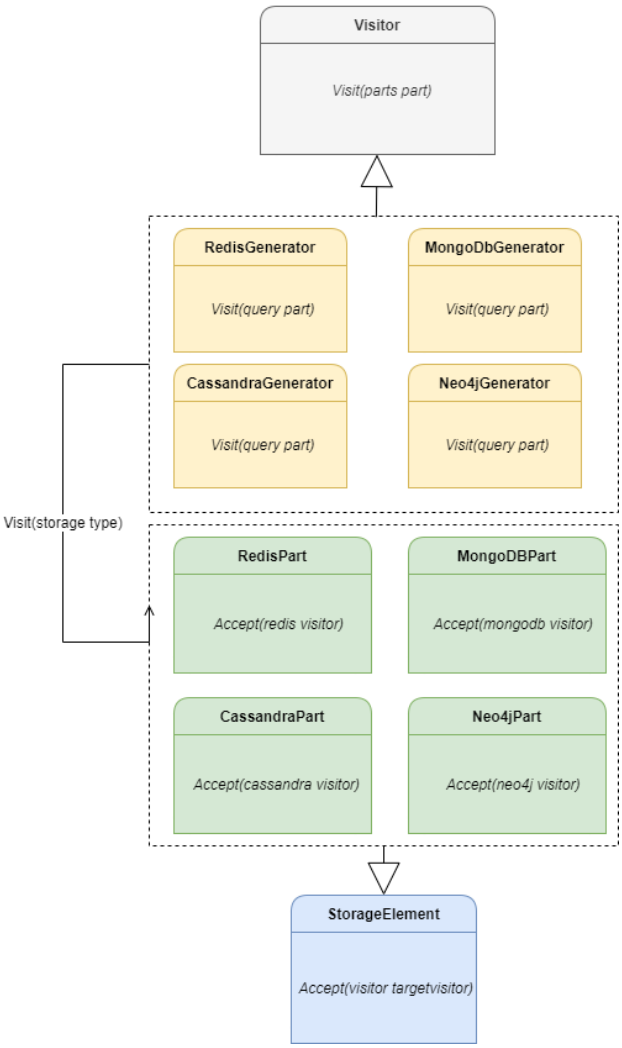


Fig. 9. Query generator: visitor design pattern.

In the context of this study, each supported NoSQL data storage model possesses its own distinct code generating implementation as shown in Fig. 9. This pattern empowers the prototype to traverse through various elements of the query expressions, constructing parts of the native query while retaining its internal state, referred to as the 'whole part' or native query. As the prototype progresses through the organized parts, it invokes other visitors, thereby facilitating the construction of complex query structures in a systematic and controlled manner.

The query generator uniquely encompasses a collection of classes called visitors, each one responsible for generating a part relevant to native query; $VS \rightarrow \{vs_1, \dots, vs_n\}$. In Table VI., *rg* represents redis, *cg* cassandra, *mg* mongodb while *ng* neo4j. The supported NoSQL categories are tied to a storage element which delegates deciding on which code generator to invoke based on the target models in the unpacked in the translation component, $SE \rightarrow \{qe_1, \dots, qe_n\}$. Each visitor represent a specific part within the broader query. The conversion of the unified query requires the visitor to be a specified order. The query generator then proceeds to systematically build each part of the native query, thus returning an executable query.

TABLE VI. VISITOR PATTERN PSEUDOCODE

Algorithm : Query Generator
$I \rightarrow QueryPath(q \rightarrow Query)$
$:= \boldsymbol{QueryGenerator(i)}$
$\boldsymbol{if}\ i \vdash I\ \boldsymbol{do}$
$strategy\ path\ \rightarrow\ i.DS$
$VS \rightarrow\ BuildVisitors(i.query\_elements)$
$\boldsymbol{for}\ each\ part\ in\ VS\ \boldsymbol{do}$
$\boldsymbol{if}\ part\ \in rg\ \boldsymbol{do}$
$rg.Accept(part)$
$\boldsymbol{if}\ part\ \in cg\ \boldsymbol{do}$
$cg.Accept(part)$
$\boldsymbol{if}\ part\ \in mg\ \boldsymbol{do}$
$mg.Accept(part)$
$\boldsymbol{if}\ part\ \in ng\ \boldsymbol{do}$
$ng.Accept(part)$

The prototype components and embedded features in tandem with the programming design patterns aids in producing a working artifact. Careful consideration was given to the middleware as an abstraction layer. By applying a separation of concerns approach, enabled components and features within the prototype to operated independently. Thus, delegating any tasks to the isolated components. Furthermore, compartmentalisation of the intents, the generators and executable paths supplied a clear route for the query processing system.

D. Limitations

The study encountered several limitations and challenges during the research endeavour, including. Initially, the study proposed an automated schema identifier capable of affecting the underlying native schemas through the prototype. However, due to time constraints, this feature was excluded from the scope of the research project. Manual schema updates were necessary, leaving the prototype susceptible to errors. The prototype struggled to handle complex data additions and updates, particularly in the case of nested query processing based on existing data models. Updates couldn't be performed on complex fields within the Cassandra database management system, as it required retrieving the entire object, updating the identified field(s), and then sending the entire field back for modification. The study was restricted to specific versions of the supported NoSQL data storage options. Any changes in versions of the respective NoSQL database management system may render the solution obsolete or cause previously successful unified queries to produce errors. The adaptors developed for the prototype relied on a rudimentary security mechanism for the respective NoSQL databases, requiring connections to be authenticated.

V. EXPERIMENTAL APPROACH

We've conducted an experiment to assess the complexity of key algorithms contained within the overarching design principles. The prototype was tested against varying workloads contained within threads to measure its scalability and robustness. In accordance with Hevner et al. (2004), it is imperative to meticulously demonstrate the effectiveness of an artifact through the appropriate evaluation methods. Therefore the prototype was subjected to ninety-one individual test cases, shown in Table VII. Each test cases were grouped to specific

query intents in order to isolated and identify potential errors or performance degradation. Furthermore, each test case represents a participant or user assigned to a predefined query to leverage control over the experiment. This enable us to effectively automate the testing process.

TABLE VII. TEST CASE SUMMARY

#	Summary	Test Cases
1	Syntax and Sematic Validations.	87, 88, 89, 90, 91
2	Retrieve complete dataset.	1, 9, 28, 45, 66
3	Retrieve dataset where a single filter was applied.	2, 3, 4, 10, 16, 17, 54, 67, 77, 78, 79
4	Retrieve dataset where a multiples filters were applied.	11,12, 15, 29, 30, 55, 56, 68, 69, 70, 80, 81
5	Apply a limit to the dataset retrieval process.	13, 31, 46, 47, 48, 49, 50, 51, 52, 53
6	Apply sorting to the dataset retrieval process.	14, 32, 33, 34, 35, 36, 57, 71
7	Aggregation on a datasets.	18, 19, 20, 21, 22, 37, 38, 39, 40, 41, 58, 59, 60, 61, 62, 72, 73, 74, 75, 76
8	Update existing dataset.	5, 6, 23, 24, 25, 42, 43, 63, 64, 82, 83, 84, 85
9	Data inserts.	7, 8, 26, 27, 44, 65, 86

We conduct the experiment using an Intel(R) Core(TM) i7-10610U CPU running at 1.80GHz with a maximum frequency of 2.30GHz. The device is equipped with 16,0 GB (15,6 GB usable). The system operates on a 64-bit Windows operating system and is based on an x64 processor architecture.

A. Participants

We purposefully embedded a module within the prototype which comprised of participants. The participants within the context of this study served as human stakeholders with specific query intents. Each participant invoked the prototype's query language, consisting of either data retrieval, modification, or insertion commands. The query workloads assisted in automating the experimental process and facilitating the capturing of performance metrics for analysis. In addition, we were able to control the expected outcomes in deterministic manner. Thus playing a crucial role in evaluating the prototype's performance.

B. Metrics

The data collected for each payload encompasses a number of varying metrics which includes the Apdex, CPU usage, memory usage, execution times for each individual component and error rates. The Apdex, CPU and memory usage enveloped the entire query's execution path. While the execution times and error rates were logged at a granular level with respect to each component i.e. the query parser, translator and executor.

1) Application performance index: The Apdex or Application Performance Index score is an industry standard, which was utilised to assess the users or participants satisfaction rate in terms of the responsiveness of the prototype. It's a binary metric whereby 1 represents the best possible outcome, alternatively 0 represents the worst possible outcome. In this study, we've set benchmarks to classify the user experience as follows :

- *Satisfied* - Response time less than 2 seconds
- *Tolerating* - Response time between 2 and 8 seconds
- *Frustrating* - Response time greater than 8 seconds

let's say :

- *sr* is satisfied requests
- *tr* is tolerating requests
- *s* is the total number of requests (i.e. sample size)

$$\therefore Apdex\ Score = \frac{\left(sr+\frac{tr}{2}\right)}{s} \quad (4)$$

2) CPU usage : The prototype's consumption of the Central Processing Unit (CPU) provided a multifaceted perspective on performance, functionality and viability of the solution.

let's say :

- *st* is the start time of CPU utilisation
- *et* is the end time of CPU utilisation
- *pa* is the number of processors available to the current process
- *pt* is the total processing time

$$\therefore\ CPU\ Usage = \frac{(et-st)}{(pa \times pt)} \quad (5)$$

3) Memory usage :The memory consumption of the prototype was evaluated from two perspectives, both the virtual and physical. In both instances the memory expenditure was calculated as follows :

In the case of virtual memory:

- *ivm* is the initial amount of virtual memory allocated.
- *fvm* is the final amount of virtual memory allocated.

$$\therefore\ vm = fvm - ivm \quad (6)$$

In the case of physical memory:

- *ipm* is the initial amount of physical memory allocated.
- *fpm* is the final amount of physical memory allocated.

$$\therefore\ pm = fpm - ipm \quad (7)$$

4) Query execution times : The individual components of the prototype measured the respective execution times in milliseconds. The parser determines the time taken for the global parser to validate the unified query. The translator measures the time taken for the translator to generate the native queries. Whereas the executor measures execution time of the generated native query on the supported storage system.

Elapsed time measurement:

- *st* is start time
- *et* is end time

$$\therefore\ el = et - st \tag{8}$$

5) Error rate: The components, namely the parser, translator and executor reported the number of errors produced by each of the automated participants.

VI. Prototype's Results

The prototype's architecture adheres to established design principles promoting modularity, extensibility, reusability and scalability. This section assesses the efficacy of those applied principles evaluating the varying algorithms employed in the query parsing, translation, and execution processes.

1) Application performance index: In the instance of the Apdex data acquired, the queries executed when viewed from an overall perspective, demonstrates a minimal use of resources within the call stack, leading to an optimal execution path. This efficiency is further corroborated by the Apdex scores in Fig. 10, which consistently indicated that the results were delivered within an acceptable timeframe. Therefore it is plausible to assert that the query parser, translator, and executor worked in harmony to ensure timely query responses from multiple storage mechanisms. However, the experimental results also indicated performance outlier's whereby certain tests exceeded the satisfactory threshold. This was evident in the Neo4j storage system in test group 2, as a large amount of connected nodes degraded performance as observed in Cox et al. (2020) study.

The other notable observation relates to the use of the "*OR*" logical operator. The experiment revealed when applying deepened search criteria, it results in longer execution times, negatively affecting Apdex score. These compounding factors highlights a need to improve the metamodel in terms of enhanced cataloguing which affects the translation feature. Firstly, the metamodel requires an improved awareness of the with each targeted storage systems indexes. Secondly, it need to be aware of the capabilities for the individual storage systems to a certain extent. This should encompass the limitations of the supported models, thus aiding in the translation process to support efficient executable native queries.

2) CPU usage: The objective was to assess whether the prototype excessively consumed the physical machine's resources during the simulated tests. We deliberately overloaded the prototype with threaded workloads to monitor if it caused system instability or crashes during operations. The prototype demonstrated fluctuations in the CPU based on the query activities. Each query of the predefined queries induced, handled by a dedicated thread intentionally loaded the CPU with requests to measure the feasibility of the prototype. It proved to show peak activity during high query loads and effectively releases the processor at the appropriate time revealing the efficient design algorithms applied to the parser, translator and executor (Fig. 11).

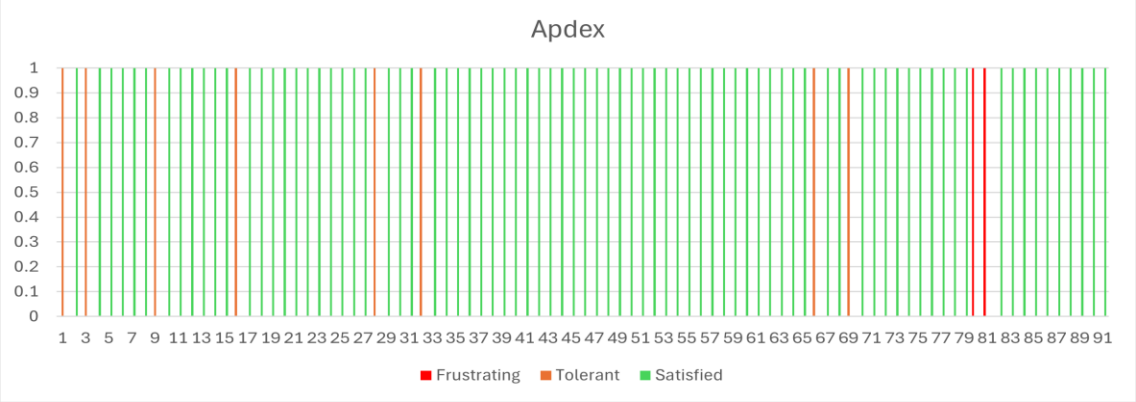


Fig. 10. Apdex scores.

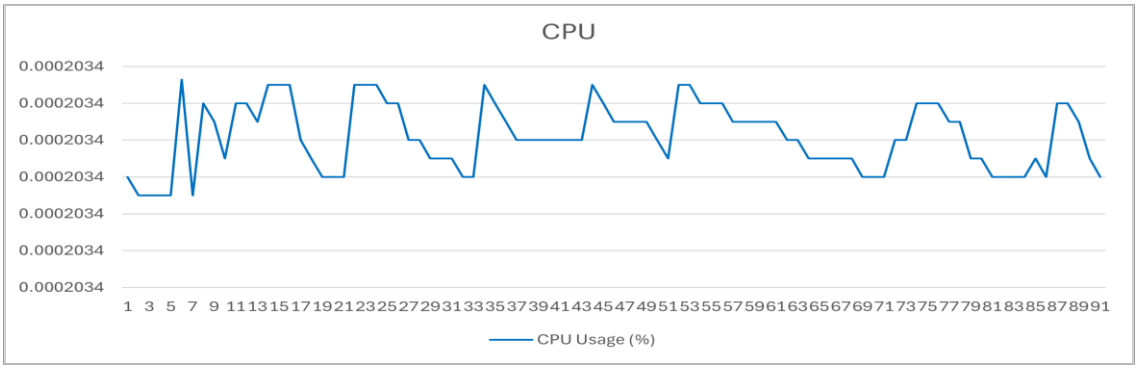


Fig. 11. Central process unit consumption.

3) Memory usage: We've observed the correlation between increased query workloads and memory consumption. This associative behaviour is expected, however more importantly, it was fundamental to ascertain how well the prototype releases memory. On start up, the prototype initially consumed more virtual memory than physical memory. Nonetheless, once the query workloads was imposed, the system virtual memory exceed the physical memory. This indicates that the query parser, translator and executor optimally utilises the available RAM to achieve effective performance rather than relying on slower disk-based memory i.e. *vm*. Furthermore, this implies the system ensured no excessive memory consumption which underscores the robustness of the prototype's architectural design choices (Fig. 12).

4) Query execution times: This applies to the prototype's query parser, translator and executor to determine any bottlenecks in the query execution path illustrated in Fig. 13. The response times of each component generally produced favourable results. The granular results of each component enabled the authors to further assess the pertinency of the design principles applied to each component. Thus strengthening the findings of the CPU, memory and Apdex results. As discovered in the Apdex results, the executor highlighted inefficiencies in the translator component. The query executor performance explicitly depends on how well a native query is generated by the translator. We've observed apply sorting and logical operators has a significant impact on the overall responsiveness of the prototype.

5) Error rates: The number of errors produced during the experiment signifies the stability and reliability of the prototype. In general the prototype exhibited low error rates under the varying workloads. The error rates were evaluated from two perspective, intentional to establish the boundaries of the system and unintentional to assess faults within the system. The data indicated, the prototype was able to distinguish between well-formed queries and non-conforming queries. It also highlighted shortcomings (Fig. 14) in the prototype revealing that the system is not aware of the full compatibilities of certain storage systems and date fields could not be parsed. In demonstrating its robustness, certain unexpected errors produced was isolated to specific targeted storage system, thus not negatively impacting all facets of the unified query.

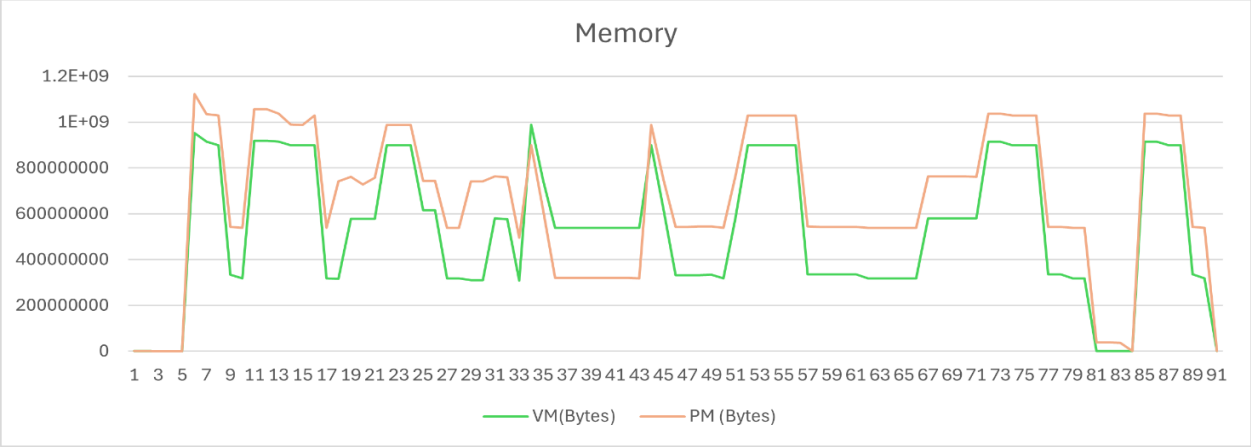


Fig. 12. Physical and memory consumption.

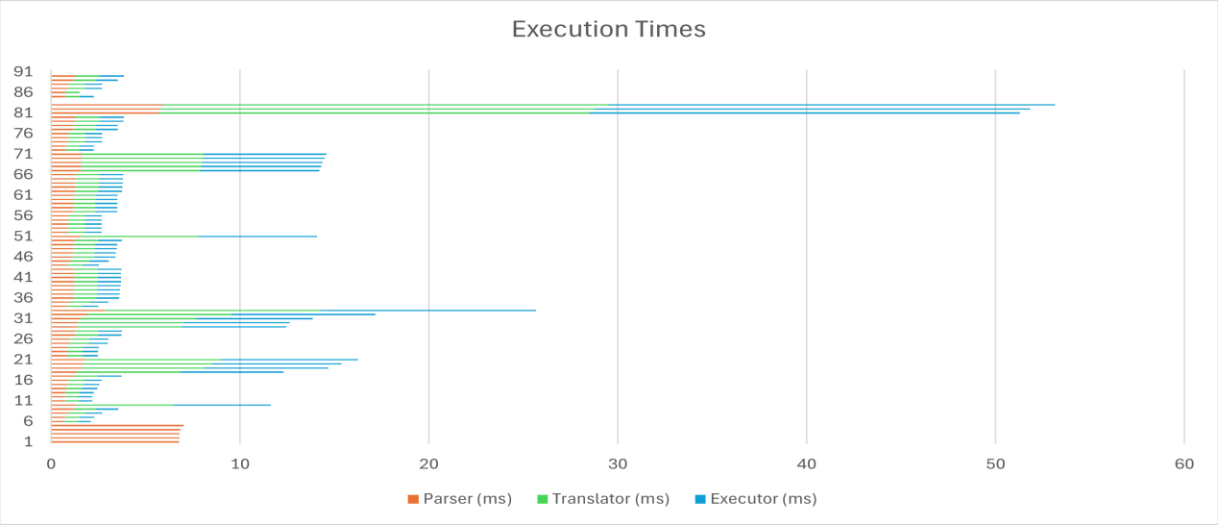


Fig. 13. Component execution times.

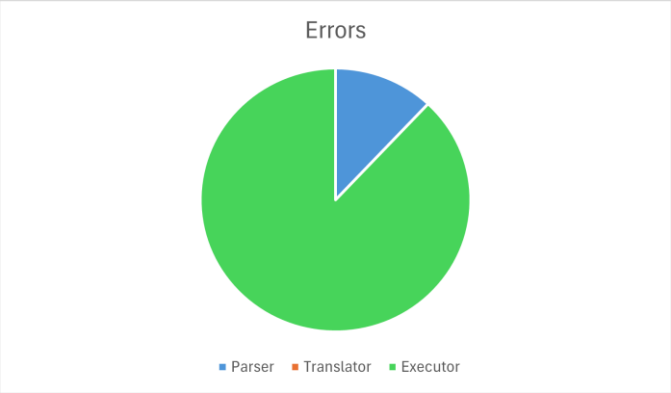


Fig. 14. Errors per component.

VII. Discussion

During our experiment, the performance data revealed that the prototype utilizes the physical machine's resources efficiently, even under load. Since excessive resource consumption can lead to a number significant challenges such system instability and degraded user experience; it important to identify and implement the optimal design patterns at inception. In certain instances, we observed fluctuations of high resource usage by the prototype which could have affected other applications running on the machine. However, the Apdex scores coupled with the query execution times and error rate demonstrated the stability of the prototype within it's environment. Fortunately we could observed that these spikes occurred in short time-bursts, preventing the prototype from monopolizing CPU and memory which could have led to degraded performance and overall user experience. We further attest to these insights as all of the participants were able to execute their respective unified queries to completion without any system interupts or fatal errors.

On reflection of the emperical data produced by the experiment, it is evident that efficiency and robustness must be prioritized from the onset. The experiment highlighted potential inefficiencies in the query translator and executor which heavily relies on the metamodel to produce well-formed native queries. The data suggests that the ineffecienct queries produced by the translator results in longer running times on the executor component. By addressing these potential bottlenecks in the query path at an early stage, it reduces the need for extensive rework later. These findings emphasis the importance of effective and efficient components as an inadequate solution from the start will exponentially increase cost and reduce quality over time i.e. user experience. This is especially pertinent in today's digital era where scalability and cost-effective solutions are at the forefront of innovation. An holistic approach to developing such polyglot systems is essential to demonstrating it's utility.

VIII. Conclusion and Future Work

In this article we presented an approach to design and develop a unified query system. The efficiency, scalability and robustness demonstrated by the prototype essentially advocates in favour of the design and architectural patterns applied to the system. A modular approach to the components supports the prototype to be easily extendable and adaptive to change, i.e. new storage systems should be easily added without having adverse effects on the existing integration. The results attained in relation to the query parser, translator and executor worked together to ensure the prototype achieved optimal performance. This is suggested in the Apdex scores achieved by the system as well as the efficient utilisation of the CPU and memory. The low error rates, affirmed the reliability of the developed prototype.

In future, we propose a study that addresses the deficiencies of the prototype. The experiment results revealed it may be beneficial for the metamodel to be partitioned in a fashion that is responsible for different aspects of the unified query system. One such aspect relates to greater schema awareness, therefore an exhaustive catalogue of alternative mappings between unified fields and natives fields including complex data types. This will offer a wider range of query translation permutations are during the native query generation process as well as supporting advanced query parsing methods. Another aspect relates to a context awareness metamodel to identify use cases supporting the accurate interpretation of query intents. Recognising the limitations of the targeted storage models to improve query optimizing algorithms within the prototype. Thus providing improved indexing strategies and query rewriting techniques. The metamodel may also benefit from cataloguing each native storage systems supported operations. This will allow the prototype to delegate unsupported operations to the middleware or at least give context is to why the intent cannot be realised. Finally, the metamodel could benefit from machine learning by either automating the catalogue process, i.e. mapping new native fields to unified model or using historical log information to improve the query optimisation process.

Conflict of Interest

The prototype utilised the Microsoft Visual Studio Community Edition IDE to develop the solution. Microsoft clearly states that this IDE may be used for academic purposes, the study in no way promotes or forms part of any advertising on behalf of the organisation. Furthermore, the study was not funded by Microsoft.

Author's Contribution

Hadwin conceptualized the research study as part of his Master of Information Communication and Technology (MICT) research journey. The student performed the required systematic literature review and built the subsequent prototype. This article represents a chapter within the thesis MICT degree. Dr B. Kabaso supervised this research journey providing invaluable insights and guidance in achieving its goal.

Acknowledgment

The authors wishes to thank the Cape Town University of Technology for providing the opportunity to partake in this study. A special sentiment of appreciation to Dr. B, Kabaso for his patience and support.

References

[1] A. Davoudian, L. Chen and M. Liu, "A survey on NoSQL stores," *ACM Computing Surveys (CSUR)*. vol. 51, no. 2, 2018, pp. 3-36. https://doi.org/10.1145/3158661

[2] A. Hevner, S.T. March, J. Park and S. Ram, "Design science research in information systems," *MIS quarterly*, vol. 28, no. 1, pp. 75-105, 2004.

[3] A. Oussous, F.Z. Benjelloun, A.A. Lahcen and S. Belfkih,. "Big Data technologies: A survey" *Journal of King Saud University-Computer and Information Sciences*, vol. 30, no. 4, 2018, pp. 432-436. https://doi.org/10.1016/j.jksuci.2017.06.001

[4] B. Kolev, C. Bondiombouy, O. Levchenko, P. Valduriez, R. Jimenez-Péris, R. Pau, and J. Pereira, "Design and implementation of the CloudMdsQL multistore system," *CLOSER: Cloud Computing and Services Science*, vol. 1, 2016, pp. 352-359. doi : 10.5220/0005923803520359

[5] C.J.F. Candel, D.S Ruiz, and J.J García-Molina, "A unified metamodel for nosql and relational databases," *ScienceDirect*. 104, p.101898, 2021, pp. 2-25. https://doi.org/10.1016/j.is.2021.101898

[6] D. Glake, F. Kiehn, M. Schmidt, F. Panse and N. Ritter, "Towards Polyglot Data Stores--Overview and Open Research Questions," *arXiv* preprint, 2022, pp. 1-27. https://doi.org/10.48550/arXiv.2204.05779

[7] F. Wedyan and S. Abufakher, "Impact of design patterns on software quality: a systematic literature review," *IET Software*, vol. 14, no.1, 2020, pp. 1-17. https://doi.org/10.1049/iet-sen.2018.5446

[8] H. Ramadhan, F.I. Indikawati, J. Kwon and B. Koo, "MusQ: A Multi-store query system for iot data using a datalog-like language," *IEEE Access*, vol. 8, 2020, pp. 58032-58050. doi: 10.1109/ACCESS.2020.2982472

[9] H. Zhang, C. Zhang, R. Hu, X. Liu and D. Dai, "Unified SQL Query Middleware for Heterogeneous Databases". *In Journal of Physics: Conference Series, IOP Publishing*, p.012065, vol. 1873, no. 1, 2021, pp. 1-6. https://doi.org/10.1007/s11431-020-1666-4

[10] I. Košmerl, K. Rabuzin, and M. Šestak, "Multi-Model Databases-Introducing Polyglot Persistence in the Big Data World," *in 2020 43rd International Convention on Information, Communication and Electronic Technology (MIPRO). IEEE*, 2020, pp. 1724-1728. doi: 10.23919/MIPRO48935.2020.9245178

[11] J. Guo, Q. Liu, J.G. Lou, Z. Li, X. Liu, T. Xie and T. Liu, "Benchmarking meaning representations in neural semantic parsing," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2020, pp. 1520-1528. doi: 10.18653/v1/2020.emnlp-main.118

[12] J. vom Brocke, A. Hevner and A. Maedche, *Introduction to design science research*. In Design Science Research. Cases. Springer, Cham, 2020, pp. 1-17. https://doi.org/10.1007/978-3-030-46781-4_1

[13] K.M. Endris, "Federated Query Processing over Heterogeneous Data Sources in a Semantic Data Lake," Doctoral dissertation, Universitäts-und Landesbibliothek Bonn, 2019, pp. 58-69.

[14] M. Duracik, P. Hrkut, E. Krsak and S. Toth, "Abstract syntax tree based source code antiplagiarism system for large projects set," *IEEE Access*, vol. 8, 2020, pp. 175350-175354. doi: 10.1109/ACCESS.2020.3026422

[15] M. Gobert, "Schema Evolution in Hybrid Databases Systems," in *[Provisoire] Proceedings of the 46th International Conference on Very Large Data Bases (VLDB 2020): PhD workshop track*, ACM Press, 2020, pp. 1-3.

[16] M. Hewasinghage, A. Abelló, J. Varga and E. Zimányi, "Managing polyglot systems metadata with hypergraphs," *Data & Knowledge Engineering, ScienceDirect*, vol. 134, p.101896, 2021, pp. 1-14. https://doi.org/10.1016/j.datak.2021.101896

[17] M. Kolonko and S. Müllenbach, "Polyglot persistence in conceptual modeling for information analysis," *in 2020 10th International Conference on Advanced Computer Information Technologies (ACIT) , IEEE*, 2020, pp. 590-594. doi: 10.1109/ACIT49673.2020.9208928

[18] M. Olsen and M. Raunak, *Quantitative Measurements of Model Credibility*. In Model Engineering for Simulation., Academic Press, 2019, ch 8, pp. 163-175. https://doi.org/10.1016/B978-0-12-813543-3.00008-1

[19] M. Zhang, "A survey of syntactic-semantic parsing based on constituent and dependency structures," *Science China Technological Sciences*, vol. 63, no. 10, 2020, pp. 1898-1920. https://doi.org/10.1007/s11431-020-1666-4

[20] N. Blumhardt. (2021). *Sprache*. [Online]. Available: https://github.com/sprache/Sprache. [Accessed 23th January 2023]

[21] N. Blumhardt. (2022). *Superpower*. [Online]. Available: https://github.com/datalust/superpower. [Accessed 23th January 2023]

[22] N. El Maghawry and A.R. Dawood, "Aspect oriented GoF design patterns," in *2010 The 7th International Conference on Informatics and Systems (INFOS)*, 2010, pp. 1-7.

[23] N. Koutroumanis, N. Kousathanas, C. Doulkeridis and A. Vlachou, "A demonstration of NoDA: unified access to NoSQL stores," Proceedings of the VLDB Endowment, vol. 14, no. 12, 2021, pp. 2851-2854. https://doi.org/10.14778/3476311.3476361

[24] N. Roy-Hubara, P. Shoval and A. Sturm, "Selecting databases for Polyglot Persistence applications," *Data & Knowledge Engineering*, vol. 137, p.101950, 2022, pp. 2-18. https://doi.org/10.1016/j.datak.2021.101950

[25] P. Atzeni, F. Bugiotti,, L. Cabibbo and R Torlone, "Data modeling in the NoSQL world," Computer Standards & Interfaces, 67, p.103149, 2020, pp. 1-10.

[26] P. Gahlyan and S.N. Singh, "Analysis of catalogue of GoF software design patterns," in *2018 8th International Conference on Cloud Computing, Data Science & Engineering (Confluence)*, 2018, pp. 814-818. doi: 10.1109/CONFLUENCE.2018.8442878

[27] P.P. Khine and Z. Wang, "A review of polyglot persistence in the Big Data world," *Information*, vol. 10, no. 4, 2019, pp. 1-19. https://doi.org/10.3390/info10040141

[28] R.Tan, R. Chirkova, V. Gadepally, and T.G. Mattson, "Enabling query processing across heterogeneous data models: A survey," *In 2017 IEEE International Conference on Big Data (Big Data). IEEE*, 2017, pp. 3211-3219. doi: 10.1109/BigData.2017.8258302

[29] S. Cox, S.C. Ahalt, J. Balhoff, C. Bizon, K. Fecho, Y. Kebede, K. Morton, A. Tropsha, P. Wang and H. Xu, "Visualization Environment for Federated Knowledge Graphs: Development of an Interactive Biomedical Query Language and Web Application Interface," *JMIR Medical Informatics*, vol. 8, no. 11, p.e17964, 2020, pp. 1-7. doi:10.2196/17964

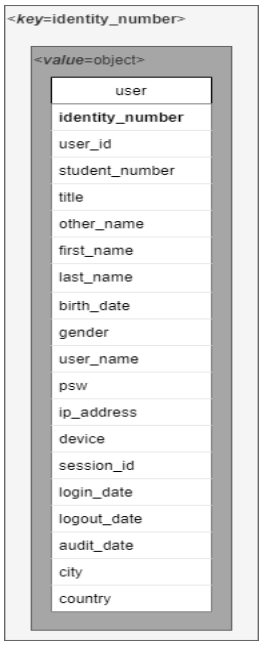
[30] V. Gadepally, P. Chen, J. Duggan, A. Elmore, B. Haynes, J. Kepner, S. Madden, T. Mattson and M. Stonebraker, "The BigDAWG polystore system and architecture," in *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, 2016, pp. 1-6. doi: 10.1109/HPEC.2016.7761636

[31] X. Yang, X. Zhang. and Y. Tong, "Simplified abstract syntax tree based semantic features learning for software change prediction," *Journal of Software: Evolution and Process*, vol. 34, no. 4, p.e2445, 2022, pp. 1-9. https://doi.org/10.1002/smr.2445

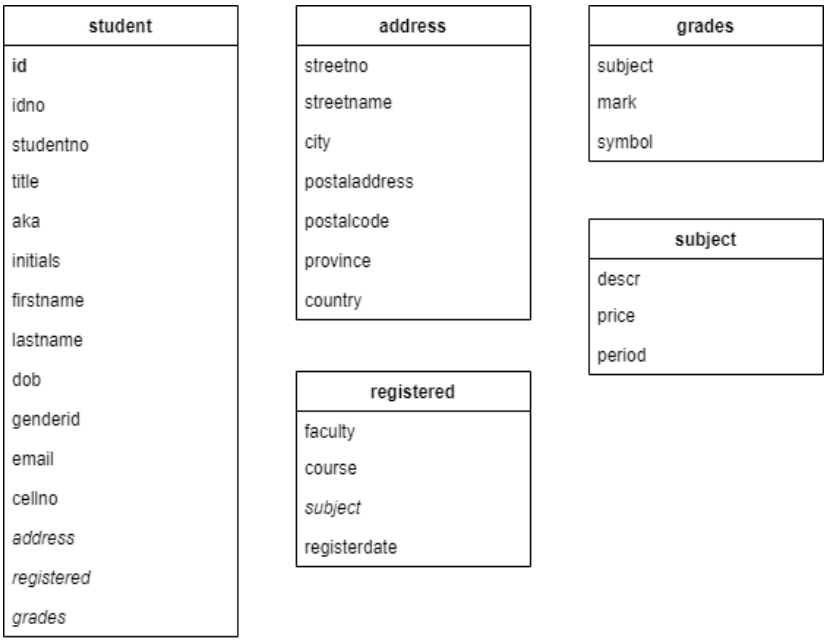
[32] Y. Khan, A. Zimmermann, A. Jha, V. Gadepally, M. D'Aquin, and R. Sahay, "One size does not fit all: Querying web polystores," *IEEE Access*, vol. 7, 2019, pp. 9598-9605. doi: 10.1109/ACCESS.2018.2888601

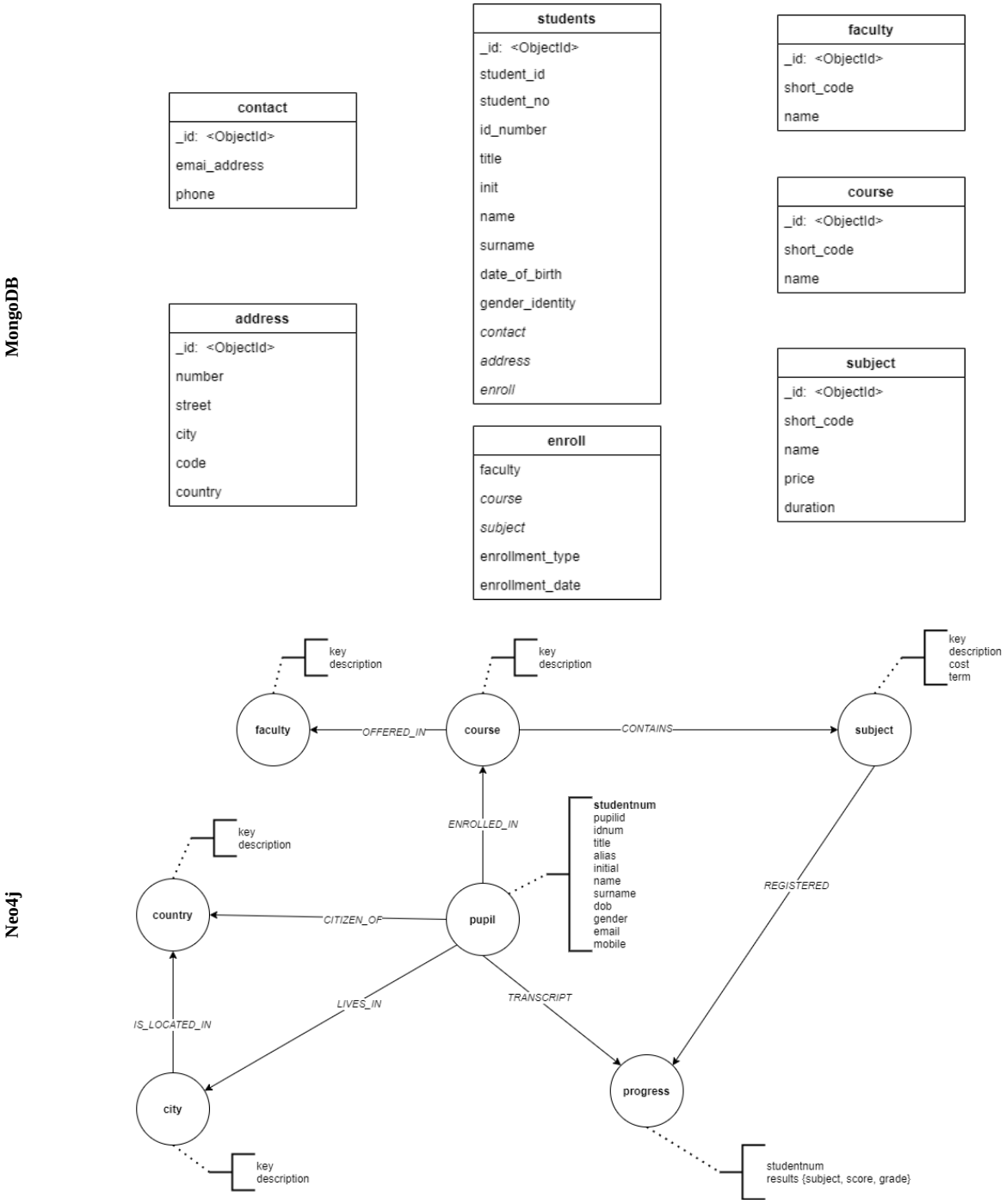
APPENDIX A: NOSQL DATABASE SCHEMAS

Redis



Cassandra





APPENDIX B: REPOSITORY MODEL

	Property	Neo4j	Mongodb	Cassandra	Redis
Models		***pupil***	***students***	***student***	***user***
student	identifier	pupilid	student_id	id	user_id
	idnumber	id	id_number	idno	identity_number
	title	title	title	title	title
	preferredname	alias		aka	other_name
	initial	initial	init	initials	
	name	name	name	firstname	first_name

	surname	surname	surname	lastname	last_name
	dateofbirth	dob	date_of_birth	dob	birth_date
	gender	gender	gender_identity	gendered	gender
	address				X
	contact				X
	register				X
	transcript				X
		faculty	***faculty***		
faculty	code	key	short_code	x	X
	name	description	name	**registered**.faculty	X
		course	**course**		
course	code	key	short_code	x	X
	name	description	name	**registered**.course	X
		subject	***subject***	***subject***	
subject	code	key	short_code	x	X
	name	description	name	descr	x
	cost	cost	price	price	x
	duration	term	duration	period	x
			address	***address***	
address	streetno	x	x	streetno	x
	street	x	street	streetname	x
	postaladdress	x	x	postalcode	x
	postalcode	x	code	postalcode	x
	suburb	x	x	suburb	x
	city	**city**.description	city	city	**user**.city
	province	x	x	province	x
	country				x
			contact		
contact	email	**pupil**.email	email_addrress	**student**.email	x
	mobile	**pupil**.mobile	phone	**student**.cellno	x
register	studentno	**pupil**.studentnum	**student**.student_no	**student**.studentno	**user**.student_number
	faculty	*faculty*	*faculty*	*faculty*	X
	course	*course*	*course*	*course*	X
	subject	*subject*	*subject*	*subject*	X
	username	x	x	x	**user**.user_name
	password	x	x	x	**user**.psw
	type	x	**enroll.**enollment_type	x	X
	ipaddress	x	x	x	**user**.ip_address
	date	x	**enroll.**enrollment_date	**register**.registerdate	X
		progress		***grades***	
transcript	subject	results.subject.description	*x*	subject	X
	result	results.score	x	grades.mark	X
	symbol	results.grade	x	grades.symbol	X

***** *Text in italics or bold denotes a class or complex object*

APPENDIX C: UNIFIED QUERY LANGUAGE TEMPLATE

Fetch Statement:

FETCH { *<property>, <function<property>,…}*

DATA_MODEL { *<data>*}

FILTER_ON { *<term> <operator> <term> <comparator>*}

RESTRICT_TO { *<number>* }

ORDER_BY { *<property>*}

TARGET { *<database vendors>,…* }

Add Statement:

ADD { *<data>*}

PROPERTIES { *<property> <operator> <property>* }

TARGET { *<database vendors>,…* }

Modify Statement:

MODIFY { *<data>*}

PROPERTIES { *<property> <operator> <property>* }

FILTER_ON { *<term> <operator> <term> <comparator >*}

TARGET { *<database vendors>,…* }

APPENDIX D: AST SAMPLE

Command	Input	Tokens
FETCH	*FETCH { id, name, surname, idnumber, dateofbirth }* *DATA_MODEL { student }* *TARGET { cassandra }*	*{FETCH@0 (line 1, column 1): FETCH}* *{PROPERTY@8 (line 1, column 9): id}* *{COMMA@10 (line 1, column 11): ,}* *{PROPERTY@12 (line 1, column 13): name}* *{COMMA@16 (line 1, column 17): ,}* *{PROPERTY@18 (line 1, column 19): surname}* *{COMMA@25 (line 1, column 26): ,}* *{PROPERTY@27 (line 1, column 28): idnumber}* *{COMMA@35 (line 1, column 36): ,}* *{PROPERTY@37 (line 1, column 38): dateofbirth}* *{DATA_MODEL@72 (line 2, column 21): DATA_MODEL}* *{DATA@85 (line 2, column 34): student}* *{TARGET@115 (line 3, column 21): TARGET}* *{NAMED_VENDOR@125 (line 3, column 31): cassandra}*
ADD	*ADD { student }* *PROPERTIES { name = 'Chuck T'}* *TARGET { cassandra }*	*{ADD@0 (line 1, column 1): ADD}* *{DATA@6 (line 1, column 7): student}* *{PROPERTIES@43 (line 2, column 27): PROPERTIES}* *{TERM@56 (line 2, column 40): name}* *{EQL@61 (line 2, column 45): =}* *{STRING@64 (line 2, column 48): Chuck T}* *{TARGET@101 (line 3, column 27): TARGET}* *{NAMED_VENDOR@110 (line 3, column 36): cassandra}*
MODIFY	*MODIFY { student }* *PROPERTIES { name = 'Chuck T'}* *TARGET { cassandra }*	*{MODIFY@0 (line 1, column 1): MODIFY}* *{DATA@9 (line 1, column 10): student}* *{PROPERTIES@48 (line 2, column 29): PROPERTIES}* *{TERM@61 (line 2, column 42): name}* *{EQL@66 (line 2, column 47): =}* *{STRING@69 (line 2, column 50): Chuck T}* *{TARGET@108 (line 3, column 29): TARGET}* *{NAMED_VENDOR@117 (line 3, column 38): cassandra}*